

iOS 9 | Xcode 7 | Swift 2

iOS APPS

for Masterminds



How to take advantage
of Swift to create insanely
great apps for iPhones and iPads

J.D Gauchat

iOS Apps **for Masterminds**

How to take advantage of Swift
to create insanely great apps
for iPhones and iPads

J.D Gauchat

www.jdgauchat.com

Edited by: **Gina Fusco**

www.reword.ca

Cover Illustration by **Patrice Garden**

www.smartcreativz.com

iOS Apps for Masterminds

Copyright © 2016 by John D Gauchat

All Rights Reserved

No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system without the prior written permission of the copyright owner.

Companies, services, or product names used in this book are for identification purposes only. All trademarks and registered trademarks are the property of their respective owners.

Apple™, iPhone™, iPad™ and iPod Touch™ are trademarks of Apple Inc.

For information on translations, please visit www.formasterminds.com

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Mink Books shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available at www.formasterminds.com

Registration Number: 1126232

1st Edition 2016 John D Gauchat

Table of Contents

INTRODUCTION

CHAPTER 1

1.1 Overview

Requirements, Mobile Operative Systems

1.2 Xcode

Running Xcode

1.3 Development

Programming Languages, Cocoa Touch API, Compiler

CHAPTER 2

2.1 Computer Programs

Playground

2.2 Variables

Memory, Primitive Types, Declaration and Initialization, Arithmetic Operators, Constants

2.3 Data Types

Characters, Strings, Booleans, Optionals, Tuples

2.4 Collections

Arrays, Sets, Dictionaries

2.5 Conditionals and Loops

If and Else, Switch, While and Repeat While, For, For In, Control Transfer Statements

2.6 Enumerations

Associated Values

CHAPTER 3

3.1 Programming Paradigms

3.2 Functions

Declaration of Functions, Standard Functions, Scopes, Closures

3.3 Structures

[Definition of Structures, Methods, Initialization, Property Keywords, Computed Properties, Type Properties and Methods, Primitive Type Structures and Casting, String Structures, Array Structures, Set Structures, Dictionary Structures](#)

[3.4 Enumerations](#)

[3.5 Objects](#)

[Definition of Objects, Property Observers, Type Properties and Methods, Optional Chaining, Reference Types, Memory Management, Inheritance, Type Casting, AnyObject, Initialization, Deinitialization](#)

[3.6 Protocols](#)

[Definition of Protocols, Extensions, Delegates](#)

[CHAPTER 4](#)

[4.1 iOS SDK](#)

[Importing Frameworks](#)

[4.2 Foundation](#)

[More Standard Functions, Strings, Ranges, Numbers, Dates](#)

[4.3 Core Graphics](#)

[CGGeometry](#)

[4.4 UIKit](#)

[UIApplication, UIDevice, UIScreen, UIWindow, UIView, UIView Subclasses, UILabel, UIProgressView, UIActivityIndicatorView, UIControl, UIButton, UISegmentedControl, UISwitch, UISlider, UIStepper](#)

[CHAPTER 5](#)

[5.1 The Interface](#)

[Xcode Projects, Xcode Tools, Running the Project](#)

[5.2 Templates](#)

[MVC, Application Delegate, Storyboard, View Controllers](#)

[5.3 Interface Builder](#)

[Object Library, Guidelines, Properties](#)

[5.4 Connections](#)

[Outlets, Connections in the Storyboard, Actions, Outlet Collections, Real-Life Application](#)

[CHAPTER 6](#)

[6.1 Adaptivity](#)

6.2 Auto Layout

[Constraints, Assigning Constraints, Editing Constraints, Standard Values, Resolving Auto Layout Issues, Intrinsic Content Size, Multiple Views Constraints, Relations and Priorities, Constraints in the Document Outline Panel, NSLayoutConstraint, Real-Life Application](#)

6.3 Size Classes

[Grid, Adapting Constraints, Adapting Elements, Adapting Fonts, UITraitCollection, Orientation, Previews](#)

CHAPTER 7

7.1 Input and Output

[UITextField, UITextView, Keyboard](#)

CHAPTER 8

8.1 Images

[UIImage and UIImageView, Visual Effects, Assets Catalog, Icons, Launching Image](#)

CHAPTER 9

9.1 Listing Values

[UIPickerView, UIDatePicker](#)

CHAPTER 10

10.1 NSTimer

CHAPTER 11

11.1 Scroll Views

[Scrolling, Zooming, Pages, Scrolling the Interface](#)

CHAPTER 12

12.1 Gesture Recognizers

CHAPTER 13

13.1 Multiple View Controllers

[Segues, Unwind Segues, Segues in Code](#)

13.2 Navigation Controllers

[Navigation Controllers in the Storyboard, Navigation Bar, Items, Toolbar, Custom Navigation, Orientation, Sharing Data](#)

13.3 Tab Bar Controllers

[Tabs, UITabBarController, Delegate, Real-Life Application](#)

CHAPTER 14

14.1 Tables

[UITableView, UITableViewCell, Table View Protocols, NSIndexPath, Implementing Table Views, Selection, Sections and Indexes, Custom Cells, Self-Sizing Cells](#)

14.2 Table Views and Navigation Controllers

[Adding Rows, Deleting Rows, Moving Rows](#)

14.3 UITableViewController

[Static Tables](#)

14.4 Search

[UISearchBar](#)

CHAPTER 15

15.1 A Collection of Views

[UICollectionView, UICollectionViewCell, UICollectionViewFlowLayout, Collection View Protocols, Layout Protocol, Implementing Collection Views, Scroll Direction, Selection](#)

[Supplementary Views, Flow Layout, Custom Layout, Real-Life Application](#)

CHAPTER 16

16.1 Universal Container

[UISplitViewController, UISplitViewControllerDelegate, Implementing Split View Controllers](#)

16.2 Improving the Interface

[Item by Default, Detail View by Default, Display Mode Button, Split View Controller Delegate](#)

16.3 Expanding the Interface

[Modal Views, UIPresentationController, UIPopoverPresentationController](#)

CHAPTER 17

17.1 Alert Views

[Alerts, Action Sheets](#)

CHAPTER 18

18.1 Notification Center

System Notifications

18.2 Local Notifications

18.3 Key/Value Observing

KVC, KVO

CHAPTER 19

19.1 Errors

Throwing Errors, Handling Errors

CHAPTER 20

20.1 Queues

Operations, Main Queue

CHAPTER 21

21.1 User Preferences

NSUserDefaults

21.2 Files

NSFileManager, URLs and Paths, Files and Directories, Files Attributes, Files Content, NSBundle

21.3 Archiving

Encoding and Decoding, Real-Life Application

CHAPTER 22

22.1 Custom Object Graph

Data Model, Core Data Stack, NSManagedObject, Managing Objects, Counting Objects, Predicates, Sort Descriptors, Delete Objects

22.2 Fetched Results Controller

Sections, Search

22.3 Migration

CHAPTER 23

23.1 Data in the Cloud

Enabling iCloud, Testing Devices

23.2 Key/Value Storage

23.3 UIDocument

[Multiple Documents](#)

22.4 Core Data in iCloud

[Seeding the Store, Duplicates](#)

CHAPTER 24

24.1 Core Graphics

[Destination, Context, Paths, Colors, Configuration, Saving the State, Clipping, Gradients, Shadows, Transformations, Text, Images, Drawing Cycle, Image Context](#)

24.2 Core Animation

[CALayer, 3D Transformations, CABasicAnimation, UIView Animations](#)

CHAPTER 25

25.1 Video

[AVAsset, AVPlayerItem, AVPlayer, AVPlayerLayer, CMTime, Video Player, AVKit](#)

25.2 Audio

[AVAudioPlayerDelegate, Recording](#)

25.3 Media

[Media Controller, Query Media](#)

CHAPTER 26

26.1 Image Picker Controller

[Camera, Photos Library, Adapting the Controller, Storing Pictures, Modifying Pictures](#)

26.2 Custom Controllers

[Camera, Photos, Videos](#)

26.3 Core Image

CHAPTER 27

27.1 Links

27.2 Web Views

[App Transport Security](#)

27.3 Web Content

[JSON, XML](#)

27.4 Social Networks

[UIActivityViewController, SLComposeViewController, SLRequest](#)

CHAPTER 28

28.1 Contacts

[Reading Contacts, Formatters, Working with Contacts, CNContactViewController, CNContactPickerViewController](#)

CHAPTER 29

29.1 Core Motion

[Motion Device, Motion Activity, Pedometer, Altimeter](#)

29.2 Core Location

[Authorization, Getting the Location, Background Location, Region, Geocoding](#)

CHAPTER 30

30.1 Map View

[Configuring the Map, Annotations, User Location, Search, Directions](#)

CHAPTER 31

31.1 Internationalization

[Formatters, Languages, Translation, NSLocalizedString, Resources](#)

CHAPTER 32

32.1 Publishing

[Apple Developer Program, Certificates, Provisioning Profiles, and Identifiers, iTunes Connect, Submitting the Application](#)

Introduction

The first iPhone was regarded as a revolutionary piece of technology, but not for the right reasons. At that moment, the innovation presented by this gadget was clearly changing our understanding of telecommunications and the preconceptions we had of what a phone could do, but nobody, not even Apple, was able to predict the impact that such an amazing piece of engineering would have in the software industry.

At first, Apple's plan was to follow the trend. They wanted to provide a new communication device with enhanced capabilities but under their strict control. The device would be capable of running its own applications, also called *apps* or *native apps*, but only those created by the company would be available. Each user would have the same apps as everybody else and only access third-party applications from the web and through the browser that was delivered with the system. Apple was trying to focus its strategy on web applications and emerging web technologies, like everybody else, but the incredible possibilities offered by the iPhone inevitably altered its plans.

The innovations introduced by this powerful device caught developers' attention all around the world, and the interest to create native applications grew fast. Soon, methods to install unauthorized applications were created and popularized. Dozens of third-party applications were developed and shared through open markets online. The success of these first apps and the willingness of users to install them (under the risk of damaging their devices) made clear that not many people agreed with the company's initial intentions. It was evident that users preferred to tap on an icon and work with the app right away rather than open a browser, remember a domain, type it, and wait for the web application to load. These applications were easy to access, focused on a single purpose, and visually more appealing than a website. Developers wanted to create and users wanted to consume native apps.

This external pressure, along with internal demands from executives in Apple's headquarters, motivated a change in Steve Jobs's mind, and the creation of the App Store was announced. The creation of the App Store not only meant that an official place to sell apps was going to be available, but also that a whole set of tools was going to be provided by the company itself to create them. The App Store and the iOS SDK (Software Development Kit) changed everything for good. Developers now had the right tools to build their apps and users had a safe place to buy them. Soon, thousands of native apps were available, generating an excellent alternative to web applications and creating a whole new market. The new apps attracted more customers for Apple's devices, which motivated the creation of more apps, in a positive spiral that generated billions of dollars. Several applications created by developers working alone from their bedrooms became an overnight success, triggering a gold rush, the effects of which are still visible nowadays.

The new distribution channels and the production tools provided by Apple have initiated a golden era for developers. The possibility and opportunities are infinite. Native applications are here to stay, the market they have created is growing by the day, and now

you can be part of it.

IMPORTANT: Supporting links, examples, projects, videos, and resources are available on our website: www.formasterminds.com. Apple's official documentation is available at **developer.apple.com**.

Chapter 1

App Development

1.1 Overview

The decision to create the App Store and open the platform to developers was not an easy one. Steve Jobs didn't like the idea of exposing his highly praised new creation to the same threats of personal computers. He didn't want the bad reputation his devices could get once affected by viruses or attacked by hackers. He insisted on keeping the platform closed until he realized that they could take advantage of the systems already developed by Apple for the iTunes Store to stay in control of quality and sales. This decision didn't come without a price for developers. The system is open enough to allow full creativity, but it's still under Apple's strict supervision, which means developers can only work according to the company's rules.

Requirements

Apple requires developers to use the software provided by the company to create apps for its mobile devices. This software only works in Apple's computers and requires the developer to have an Apple account. For these reasons, the options are very limited, but the good news is that most of the things we need are provided for free.

Mac Computer—This in theory could be any Intel-based Mac computer, but the development software always requires the latest operative system (OS X), so in practice we need a relatively new computer.

Apple Developer Account—This is a basic account we can get for free; it gives us access to the Member Center where we can watch videos, download programming guides, read the official documentation, get sample codes, and more.

Xcode—This is the software provided by Apple for development. The current version is number 7. It's free and the package comes with everything we need to create our apps, including an editor, the SDK (Software Development Kit), and a simulator to test the applications.

Apple Developer Program—This service is required to publish our apps in the App Store and implement services like iCloud, so we will need to enrol in this program as soon as our first app is ready for distribution. At this moment the membership costs \$99 US dollars per year.

Mobile Device—This could be any of the devices available in the market, including iPads, iPhones, and iPods, but at the time of picking one for testing we have to consider that old models don't support the current versions of Apple's mobile operative system (iOS). Testing our applications in a real device is highly recommended and necessary

before publishing.

The setup is relatively simple. We have to get a Mac Computer capable of running the operative system currently required by Xcode (at the time of this writing, it is OS X 10.9), open an Apple developer account from **developer.apple.com** using our personal Apple ID (the one that is created when we initialized our computer), and install the latest version of Xcode (currently 7). Xcode has everything we need, including a simulator to test mobile applications on the computer. Once the applications are created, we have to enrol to the Apple Developer Program to generate all the certificates and permissions we need to submit our apps to the App Store and put them up for sale.

Mobile Operative Systems

iOS is the name of the operative system delivered by Apple with every mobile device sold. The system is currently in version number 9. Applications programed in the Swift language only work on systems from version 7, but this doesn't reduce or affect the potential market for these applications because most devices are capable of updating their system to the latest one. At the time of this writing, it is estimated that more than 90% of Apple devices in the market are running iOS 8 or 9.

1.2 Xcode

Xcode is a general-purpose IDE (Integrated Development Environment). It includes a very powerful editor with graphic tools to help us write our code, and comes with the SDKs (Software Development Kits) for the creation of software for the iOS and OS X operative systems. The SDKs include compilers for Objective-C and the new Swift language. From Xcode, we can program software for every Apple's platform using any of these programming languages.

Xcode is available as an app on the Mac App Store. To download this application, we have to open the App Store from Launchpad (the application organizer that comes with OS X) or double click the App Store icon inside the Applications folder in Finder (OS X's file explorer). From the App Store, we have to search for the term "Xcode". The window will show the Xcode's icon at the top (Figure 1-1, number 1). We have to click on the *Free* button to download the software.



Figure 1-1: Xcode in the Mac App Store

Running Xcode

Once the downloading process is over, the software is automatically installed. To open Xcode we have to go to Launchpad and click on the icon or search for the program inside the Applications folder in Finder. Figure 1-2 shows Xcode's welcome screen.

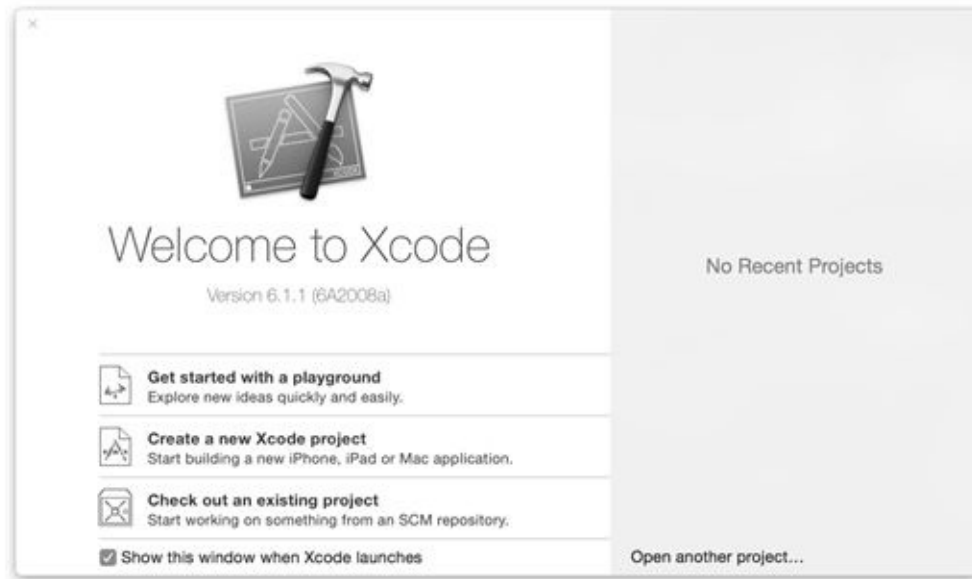


Figure 1-2: Xcode's welcome screen

The welcome screen offers a list of the recent projects on the right and buttons on the left to initiate a new project or check for those already created and stored in a repository. The options currently available are the following:

Get started with a playground—This option creates a single file and provides a simplified interface we can use to experiment and learn about the Swift language and the basic frameworks included in the SDK.

Create a new Xcode project—This is the option we choose when we want to create a new application. The option generates a new Xcode project from a template that provides basic files and settings to start from.

Check out an existing project—This option allows us to open and work on a project stored on a server. It is used to download a project already initiated by other developers or share a project with multiple developers online.

IMPORTANT: The first two options are the ones we are going to use in this book. We will open a Playground to learn elemental programming concepts in the following chapters and work with Xcode projects from Chapter 5 to learn how to create real applications.

1.3 Development

Even though some simple projects could be developed without programming a single line of code, we always have to write our own code if we want to create a useful application.

Programming Languages

Several years ago, Apple adopted and implemented a language called Objective-C to allow developers to create applications for its devices. Due to the technical level required to work with this language, the spectacular success of Apple's mobile devices didn't impress developers the same way as consumers. The demand for more and better applications was growing fast, but the complicated nature of the system didn't appeal to most developers, who were used to working with more traditional tools. To solve this problem, the company introduced some innovations to Xcode and engaged in active work to promote the evolution of Objective-C. But no matter how hard they tried, some complex features of the language, such as the extensive use of pointers and its verbose syntax, were difficult to remove and were still scaring developers away, especially web developers transitioning from simpler languages like HTML, CSS, and Javascript.

In the middle of 2014, with the purpose of putting an end to this situation, the company introduced a new programming language called Swift. Swift presents a simpler syntax that developers find familiar, while at the same time preserves that low-level nature necessary to take advantage of every aspect of Apple's devices. Swift was designed to replace Objective-C and, therefore, is the language recommended to new developers. To program our apps, we have to learn the language's main features, how to organize our code, and how to connect it with the APIs (Application Programming Interfaces) included in the SDK.

IMPORTANT: Because of the recent introduction of the Swift language, a huge part of the SDK is still programmed in Apple's previous languages (Objective-C and C). Although the company's intention is to focus on its new language, the migration of the APIs, documentation, and examples from one language to another will take some time. If you come across examples or tutorials written in Objective-C, there is a guide provided by Apple that explains how this code may be translated to Swift called "Using Swift with Cocoa and Objective-C". Also, you can always learn the basis of Objective-C from Apple's official guide. For more information, visit our website and follow the links for this chapter.

Cocoa Touch API

Programming languages by themselves can't do much. They provide all the elements to interact with the system, but are basic tools for data manipulation. Because of the complexity of the information required to control sophisticated technologies and access every aspect of a system, it could take years to develop an application from scratch working with just the instructions of a programming language. Doing simple things like printing graphics on the screen or storing data in memory would become a nightmare if programmers had to depend on the language alone. For this reason, programming languages are always accompanied by sets of pre-programmed routines that through a simple interface called API (Application programming interface) allow programmers to grant their apps amazing functionality with just a few lines of code. These sets are usually called *libraries*, and a group of libraries is called *framework*.

Xcode comes with an SDK that includes all the frameworks and APIs we need to work with every component of Apple devices. In Apple's environment, this large group of development tools is referred to as Cocoa Touch API (named after the Cocoa API created for Mac computers). We access the Cocoa Touch API from Swift code to create our apps.

IMPORTANT: Frameworks are critical for app development. Learning the APIs (public interfaces) to interact with these frameworks will become the main subject of study in following chapters.

Compiler

Computers don't understand Swift or any other programming language. These languages were created for us to give machines instructions we can understand. Our code has to be converted to elemental orders that work at an electronic level, turning multiple switches on and off to represent the abstraction humans work with. The translation from the language humans understand to the language computers understand is done by a program called *compiler*.

Compilers have specific routines to translate instructions from programming languages to machine code. They are language and platform specific; we need a specific compiler to program in one language and for one particular device. There are a few compilers available for Apple systems, but the one currently used by Xcode is called *LLVM*. LLVM is capable of compiling code written in Swift, C, C++, and Objective-C.

With the compiler, the machinery to build an app is complete. Figure 1-3 shows all the elements involved. There are three main sources of code the compiler uses to build the application: our code in Swift, the frameworks our program requires, and a set of basic

routines necessary for the app to run (called Application Loop in Figure 1-3). The process starts from Xcode. Here we write our code, access frameworks through their APIs, and configure the app to be compiled (built). Combining our code, the codes from the frameworks our app requires and the basic routines (Application Loop), the compiler creates an executable program (builds the app) that may be run in a simulator, a device, or submitted to the App Store for distribution.

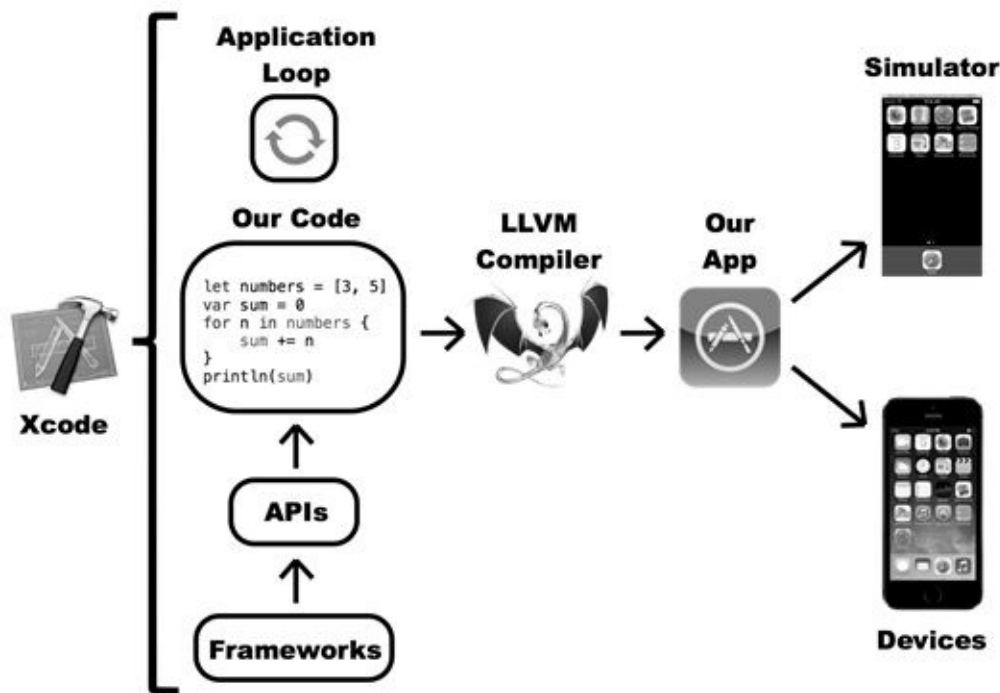


Figure 1-3: Building an App

The Basics: The Application Loop is a group of elemental routines, common to every program, that connects your app to the operative system and provides a loop (a code that executes itself over and over again) to constantly check for events produced by the user's interaction or messages coming from the system. Although you never work directly with these routines, they are connected to your code to inform the state of the program, as we will see in further chapters.

Chapter 2

Introduction to Swift

2.1 Computer Programs

To get a computer to do what we want we have to write a program. A program is a succession of instructions that the computer has to follow. We write the program using the instructions provided by a specific programming language, then a compiler translates these instructions into orders the computer can understand, and when we tell the computer to run the program, the orders are executed sequentially one by one.

The instructions are always listed in sequential order, but programming languages offer different ways to group them together and organize the code and the data that is going to be processed. Although the instructions provided by programming languages are limited, the possible combinations are infinite. Developing an app requires a deep understanding of these instructions and the possible combinations we can apply to get the results we want. Since this may be daunting for those who want to create their first app, Xcode includes a tool called *Playground* to learn how to program.

IMPORTANT: This chapter introduces basic concepts of programming and the Swift language. If you are familiar with this information, feel free to skip the parts you already know or move to the next chapter. Our approach to Swift in this book is introductory. We work with basic and complex features, but only apply those necessary to understand the examples of the book. To expand your knowledge of this language, read the official guide provided by Apple. For more information visit our website and follow the links for this chapter.

Playground

There are two alternatives to work in Xcode: we can initiate a project to create an application or we can open a Playground file to experiment with the code before we include it in our app. Although we could use the first option and create our application right away, it's better to work with a Playground first to learn how to program with the Swift language and how to take advantage of some essential frameworks included in the SDK.

When we create an app, we have to program the code, build it (turn it into code the computer can understand), and execute it. The results of our work are visible only when we run the final product. Playground, on the other hand, is a real-time visualizer that has the purpose of showing the results of the execution of our code as we type. It is a great tool for learning and that's the reason why we are going to use it in the following chapters.

To start with Playground, we have to select the option *Get started with a playground* on the Xcode's welcome window (Figure 1-2). Xcode will ask for the name of the

Playground file and the platform we want to program for (iOS or OSX). In the next step, a window opens up to select where on our hard drive we want to store the file. When the initialization process is over, Xcode shows the Playground's interface on the screen (Figure 2-1).



Figure 2-1: Playground interface

The Basics: The Xcode's menu at the top of the screen also includes the option to create the Playground's file. If you can't see Xcode's welcome window, open the *File* menu from the menu bar, select the option *New* and click on *Playground*.

Playground presents a simple interface with only two main areas: the *Editor Area* where we write our code and the *Results Side Bar* where the results produced by our code are printed. In the Editor Area we can see the code we have programmed so far. When a new Playground file is created, Xcode offers a template that includes a few basic lines of code to start from. Listing 2-1 shows the code currently generated for iOS applications.

Listing 2-1: *Playground template*

```
//: Playground - noun: a place where people can play
import UIKit
var str = "Hello, playground"
```

A computer program is just text written in a specific format. Each line of text represents an instruction. Sometimes one line of code includes several instructions, and this is why each line is usually called *statement*. Every statement is an order for the computer to perform a task. In the code of Listing 2-1, the first statement is just a comment for the programmer, the second statement uses the instruction **import** to include in our code the pre-programmed codes from the UIKit framework, and the last statement uses the

instruction `var` to store the text “Hello, playground” in memory.

Inside the Results Side Bar, on the same vertical line of the last statement, there is an indication of what the code does (in this case, it shows the text stored in memory by the `var` instruction). When we move the mouse over this indication, two small buttons show up (Figure 2-2).



Figure 2-2: Quick Look and Show Result buttons

The button on the left is called *Quick Look* button, and it shows a popup window with a visual representation of the final result of the execution of the code, such as formatted text or an image (in this case, no visual effect is associated with the code so only the plain text is shown on that window).



Figure 2-3: Quick Look window

The button on the right is called *Show Result* button, and what it does is open a window within our code with a visual representation of the results of the execution of the code over time (in this case, nothing changes, so only the “Hello, playground” text is shown).

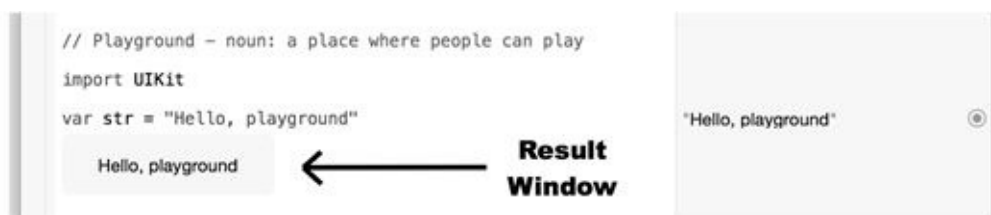


Figure 2-4: Result window

The code provided by the Xcode's template for the new Playground's file is useless, but it shows the basic syntax of the Swift language and how to do elemental things in a program such as writing comments, importing frameworks for additional functionality, and storing data. The reason why one of the statements is storing data in memory is because this is the most important task of a program. A program's main functions are storing, retrieving, and processing data. Working with data in the computer's memory is a delicate process that demands meticulous organization. If we are not careful, data may be accidentally deleted, corrupted, or completely overwritten. To make sure this doesn't happen, programming languages introduce the concept of variables.

2.2 Variables

Variables are values in memory represented by a name. Once a variable is defined, its name remains the same but the value in memory they represent may change. This allows us to store and retrieve a value from memory without the need of remembering where in the memory the value was stored. The system takes care of memory management in the background. With just mentioning the name of the variable created to store a value, we can get it back or replace it with a new one. Let's see how all of this works.

Memory

A computer's memory is like a huge honeycomb, with consecutive cells that can be in two possible states: activated or deactivated. They are actually electronic switches with on and off positions established by low and high energy levels.



Figure 2-5: Memory cells

Because of their two possible states, each cell is a small unit of information. One cell may represent two possible states (switch on or off), but by combining a sequence of cells we can represent more states. For example, if we combine two cells, there are four states possible, as shown below.

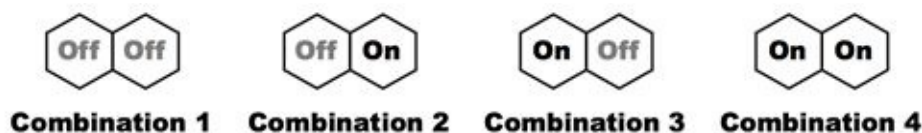


Figure 2-6: Combining two cells

With these two cells we can now represent up to 4 states (4 possible combinations). If we had used three cells instead, then the possible combinations would have been 8 (eight states). The number of combinations is duplicated every time we add another cell to the group. This can be extended to represent any number of states we want.

This system of switches was actually created to represent binary numbers (numbers

expressed by only two digits: 0 and 1). An on switch corresponds to the value 1 and an off switch corresponds to the value 0. For practical purposes, basic units were determined with the purpose to identify parts of this endless series of digits. One cell was called a *bit*, and a group of 8 bits was called a *byte*. The possible combinations of 8 bits is 256, therefore, a byte can represent binary numbers of 8 digits, which in turn can be converted to numbers humans can understand, such as decimal numbers. With its 256 possible combinations, a byte can represent decimal numbers from 0 to 255. Figure 2-7 shows how a byte looks like in memory, with some of its switches on representing the binary number 00011101.

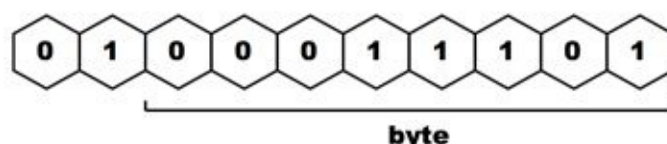


Figure 2-7: Representation of one byte in memory

Because of their nature, computers are incapable of storing decimal values like 255; they can only represent binary numbers stored as a sequence of switches on and off, but a binary number may be converted to a number of any other numerical system. For instance, when the value of the byte in this example is converted to the decimal system, we get the number 29.

IMPORTANT: To learn how to transform a binary number into a decimal or a hexadecimal number, visit our website and follow the links for this chapter.

To represent larger numbers, bytes are grouped together. For example, if we take two bytes from memory, we get a binary number composed by a total of 16 bits (16 zeros and ones). A binary number of 16 bits can represent decimal numbers from 0 to 65535 (a total of 65536 possible combinations). For coherence and to establish clearly defined data structures, each programming language declares its own units of data of a predetermined size. These units are language specific, but there are some elemental units that we will find in almost every language because they are the building blocks of more complex types of data. These units are usually called *primitive types*.

Primitive Types

Primitive types are types of units of data defined by the language. They have always the same size, so when we store a value of one of these types the computer knows exactly

how much memory to use and how to read it later. Of all the primitive types, probably the most useful is `Int`:

Int—This type is for integer numbers (numbers with no fractional component). In Swift, the size of this type is variable; it is adjusted to the system the app is compiled for. In 32 bits systems, the size of the `Int` type is 4 bytes and it can store values from -2,147,483,648 to 2,147,483,647, while in 64 bits systems the size is 8 bytes and it can store values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Although it is recommended to always use the `Int` type to store integer numbers, the automatic adaptation to the system makes it unreliable when working with frameworks that require a very specific type of integer. Because of this, Swift also defines the following types:

Int8—This type is for integer numbers of a size of 1 byte (8 bits). Because of its size, it can store values from -128 to 127.

Int16—This type is for integer numbers of a size of 2 bytes (16 bits). Because of its size, it can store values from -32,768 to 32,767.

Int32—This type is for integer numbers of a size of 4 bytes (32 bits). Because of its size, it can store values from -2,147,483,648 to 2,147,483,647.

Int64—This type is for integer numbers of a size of 8 bytes (64 bits). Because of its size, it can store values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

If we check the size of each type presented so far and calculate the possible combinations of bits, we will discover that the maximum values are not right. For example, an `int8` uses 1 byte, which means it is made by 8 bits, and for this reason it should be able to store numbers from 0 to 255 (256 possible combinations). The reason why an `int8` has a positive limit of 127 is because it only uses 7 bits to store its value, the first bit at the left side is reserved to indicate the sign (positive or negative). Although these limits are not restrictive, the language also provides the unsigned versions of these types in case we need to store larger positive values:

UInt—This is the same as `Int` but for unsigned values. Because it doesn't reserve a bit for the sign, it can store values from 0 to 4,294,967,295 in 32 bits systems and 0 to 18,446,744,073,709,551,615 in 64 bits systems.

The specific types for `UInt` are: `UInt8`, `UInt16`, `UInt32`, and `UInt64`. These work exactly like their equivalents for `Int`, but in this case they are intended to store unsigned values (positive numbers).

Although all these types are very useful, they are only good for storing binary values that can be used to represent integer numbers. Arithmetic operations also require the use of real numbers (e.g., 3.14 or 10.543). Computers can't reproduce this type of number, but they can work with an approximation called floating-point numbers. In old computers, integers were directly converted to binary and stored in memory, while floating-point numbers were simulated by software. Because of performance issues, computers incorporated an extra processor to manage floating-point numbers. Instead of reading the binary number and calculating the real number from that value, these processors assign a specific purpose for different parts of the binary value. There is one bit reserved for the sign, as with signed integers, and the rest is divided into two parts: the *significand* and the *exponent*. The exponent is multiplied by the significand to get the final number. This is called *floating-point processing*. From this process, new primitive types were defined. The following are the types defined in the Swift language.

Float—This type is for 32 bits floating-point numbers. It has a precision of 6 decimal digits.

Double—This type is for 64 bits floating-point numbers. It has a precision of at least 15 decimal digits.

Floating-point types can handle huge numbers using scientific notation, but because of their precision, it is recommended to declare a variable of type `Double` when performing calculations and use `Float` for minor tasks.

IMPORTANT: Although all of these types are useful, `Int`, `Float`, and `Double` are the most commonly used. The `Int` type is the one recommended by Apple when working with integer numbers because of its adaptability to the system. The `Float` and `Double` types are both equally useful, but because of its higher precision level the `Double` type is recommended when performing calculations while the `Float` type is reserved to store coordinate values to position graphics on the screen.

Declaration and Initialization

If we want to store data in memory, the first thing we need to do is to select the right type from the data types provided by the language and then create a variable of that type. This action is called *Declaration*, and it is done using the `var` instruction and the syntax `var name`

: type.

Listing 2-2: Declaring variables

```
var mynumber : Int
```

The example of Listing 2-2 creates a variable called `mynumber` of type `Int`. When the system reads this statement, it reserves a space in memory of 4 bytes long (8 bytes in a 64 bits system) and assigns the name `mynumber` to that space. After the execution of this statement, we can use the variable `mynumber` to store in memory any integer value from -2,147,483,648 to 2,147,483,647 (or larger if it is a 64 bits system).

IMPORTANT: According to the official specification, you can use almost any character you want to declare the name of a variable, except for spaces, mathematical symbols, and a few special Unicode characters. Also, they can't start with a number. If you declare a variable with an illegal name, Xcode will show you an error. There are two things you should always remember: Swift distinguishes lower and upper case characters (`MyInt` is considered a different variable than `myint`), and also that to avoid conflict with reserved words we should declare complex names or define the names with a combination of two or more words (for example, `myint` instead of `int`).

The Basics: The spaces between the name, the type, and the colon are recommended but not required. You can write the instruction of Listing 2-2 without any space (e.g., `var myvariable:Int`) or just one (e.g., `var myvariable: Int`).

Memory is a reusable resource. The space we reserve for a variable may have been used before by another variable that was deleted, or pieces of functional code could have been stored in that place, leaving incomprehensible data in the same space of memory we are now trying to use for the new variable. For this reason, after the declaration of a variable we always have to store a value in it. This action is called *Initialization*.

Listing 2-3: Initializing variables

```
var mynumber: Int  
mynumber = 5
```

In the new example of Listing 2-3, we first declare the variable as we did before, and then initialize it with the value 5 (we store the number 5 in the space of memory that was reserved for this variable). To store the value we use the = (equal) symbol and the syntax

name = **value**, where **name** is the name of the variable and **value** is the value we want to store (once the variable was already declared, we don't have to use the **var** instruction or specify its type anymore).

Although this code doesn't perform any other task than storing the value 5 in memory, we can use a Playground's feature to check the current value stored in the variable. To do this, we just have to write the name of the variable in a new statement and the current value will be shown on the Results Side Bar.

Listing 2-4: Reading variables

```
var mynumber: Int
mynumber = 5
mynumber // 5
```

Do It Yourself: If you didn't do it yet, create a new Playground file. When the interface is opened, replace all the statements of the Xcode's template by the code of Listing 2-4. You should see the value 5 on the Results Side Bar. Repeat this process for the following examples. Each example in this chapter replaces the previous one.

The Basics: The text added at the end of the last statement of Listing 2-4 is a comment used to show the value produced by the statement. Comments are ignored by the compiler but useful for programmers to remember vital information that could help in further reviews of the code. They are introduced after the characters `//` (e.g., `// comment`) or in between the characters `/*` / (e.g., `/* comment */`). The compiler ignores everything written on the right side of `//`, and everything between `/*` and `*/`. You can also use these characters to comment lines of code you don't want to use at the moment. Xcode offers a shortcut to comment code. Select the lines of code you want to comment and press the keys `Command` and `/`.

Most of the time we know what the initial value for the variable will be right away. In cases like this, Swift allows us to declare and initialize the variable in just one line of code.

Listing 2-5: Declaring and initializing variables in the same statement

```
var mynumber: Int = 5
```

Variables are called variables because they are not constant. We can change their values any time we want. To store a new value in the space of memory reserved for a variable, we have to use the same syntax we use for initialization.

Listing 2-6: Assigning a new value to the variable

```
var mynumber: Int = 5  
mynumber = 87
```

The process of storing a new value is called *assign*. In these terms, we can say that in the example of Listing 2-6 we “assign the value 87 to the variable **mynumber**“. This new value replaces the old one in memory. After that second statement, every time we read the variable it will always return the value 87 (unless another value is assigned to the variable later).

IMPORTANT: Once a variable is declared, the values stored in that variable have to be of the same type. If we declare a variable of type **Int**, we can’t store floating-point values in it (e.g. 14.129).

Of course, we can create all the variables we want and of the types we need.

Listing 2-7: Declaring variables of different types

```
var mynumber: Int = 5  
var myfavorite: Float = 14.129
```

The first statement of Listing 2-7 declares an integer variable and initialize it with the value 5, and the second statement does the same but for a floating-point variable. When the values are of a clear type, Swift can infer them. If this is the case, the syntax may be simplified, as in the following example.

Listing 2-8: Declaring variables without specifying the type

```
var mynumber = 5  
var myfavorite = 14.129
```

Swift infers the variable’s type from the value we are trying to assign to it. In this last example, the value 5 is clearly an integer and the value 14.129 is clearly a floating-point value, so Swift will create the variable **mynumber** of the type **Int** and the variable **myfavorite** of the type **Double** (it selects the most comprehensive type).

IMPORTANT: When the type is not clear, we have to declare it explicitly. For

example, if we are initializing a variable with the value 8, but we want to assign floating-point numbers to this variable in the future, then we have to declare it as **Float** or **Double** (as we did in Listing 2-7), otherwise Swift will infer an **Int**.

The Basics: There is a simple tool we can use to confirm the data type assigned to a variable and get additional information. All you have to do is click on the name of the variable while holding down the Option key on your keyboard. This opens a popup window with the full declaration of the variable, including its data type, and any information we may need to identify the function of the code. As we will see later, this not only works with variables, but also constants and any type of instruction, including functions, properties, methods, etc.

The value of one variable may be assigned to another.

Listing 2-9: Assigning variables to variables

```
var mynumber = 5  
var myfavorite = mynumber
```

The second statement of Listing 2-9 reads the value of the **mynumber** variable and assigns it to the **myfavorite** variable. The type of **myfavorite** is inferred to be **Int** (the same of **mynumber**).

Arithmetic Operators

Storing values in memory is what variables allow us to do, but those values don't have to be declared explicitly, they can also be the result of arithmetic operations. Swift supports the five basic arithmetic operations: + (addition), - (subtraction), * (multiplication), / (division) and % (remainder).

Listing 2-10: Assigning the result of an operation to a variable

```
var mynumber = 5 + 10 // 15
```

When the system reads the statement of Listing 2-10, it adds 10 to 5 and assigns the result to **mynumber** (15). Of course, we can use variables of any data type we need and perform any of the operations available.

Listing 2-11: Performing operations in variables of different type

```
var mynumber = 2 * 25 // 50
var anothernumber = 8 - 40 * 2 // -72
var myfraction = 5.0 / 2.0 // 2.5
```

The first two statements of Listing 2-11 are easy to read. They perform arithmetic operations over integer numbers that produce an integer value as result, so the variables **mynumber** and **anothernumber** will be of type **Int**. The problem arises when we work with operations that may produce floating-point numbers. That is why for the third statement we specifically declared the values as floating-point numbers (adding the .0 at the end). This forces Swift to infer the variable's type as **Double**.

The Basics: Arithmetic operations are executed following an order of precedence determined by the operators. For example, multiplication and division have more precedence over addition and subtraction. This means that the multiplications and divisions will be performed before the additions and subtractions. If you want to change the order, you have to include the values between parentheses (e.g., **var number = (8 - 40) * 2**).

When the compiler finds an operation with two or more numbers and has to infer the type of the result, it converts the number of the less specific type to the more specific type. For example, when we declare an **Int** and a **Double** in the same operation (e.g., **5 + 2.0**), the **Int** value will be converted and processed as **Double**, and the result will also be **Double**. On the other hand, if all the values are integers, then the result will be an integer (unless the variable is explicitly declared as **Float** or **Double**). Here is an example:

Listing 2-12: Type inference

```
var myfraction1 = 5.0 / 2.0 // 2.5
var myfraction2 = 5 / 2.0 // 2.5
var myfraction3 = 5 / 2 // 2
```

Listing 2-12 declares and initializes three variables. The first statement uses the same operation performed for one of the variables in Listing 2-11. In this case, both numbers were declared as floating-point values, so the compiler infers a **Double** and creates the **myfraction1** variable of that type. In the second statement we have an integer value and a floating-point value. Because of the floating-point value, the compiler interprets the integer (5) as a **Double** (5.0) and creates the **myfraction2** variable of type **Double**. But in the last

statement there is no clear floating-point value. Both numbers were declared as integers (with no decimals). In this case, the compiler doesn't know what we want, so it interprets both numbers as integers and creates the `myfraction3` variable of type `Int`. When an operation produces a result that is expected to be an integer, any fractional part is discarded. In this example, the system gets rid of the decimal 5 from the result and only assigns the integer 2 to the variable. If we don't want to lose the fractional part, we don't have to let the compiler infer the type of the variable and declare it ourselves as `Float` or `Double` (e.g., `var myfraction3: Double = 5 / 2`).

IMPORTANT: When we work with numbers, we can declare the type explicitly by adding the decimal part, but when working with variables, they are already of a specific type so they must be converted. There is a process called *casting* that allows us to convert a variable of one data type to another. We will study this process in Chapter 3.

Dividing integer numbers may be pointless most of the time, except in some circumstances when we need to know the remainder. The remainder is the amount left over by a division between two numbers, and is calculated using the `%` symbol.

Listing 2-13: Calculating the remainders

```
var remainder1 = 11 % 3 // 2
var remainder2 = 20 % 8 // 4
var remainder3 = 5 % 2 // 1
```

Listing 2-13 shows three examples of how to calculate the remainder of a division. Each statement calculates the remainder of the division of the first number by the second one, and assigns the result to the variable. For instance, the first statement produces the remainder 2. The system divides 11 by 3 and finds a quotient of 3. Then, to get the remainder, it calculates 11 minus the multiplication of 3 times the quotient ($11 - (3 * 3) = 2$).

The second statement produces a remainder of 4 and the third statement produces a remainder of 1. This last statement is particularly useful because it allows us to determine whether a value is odd or even. When we calculate the remainder of an integer number divided by 2, we get a result according to its parity (odd or even). If the number is even, the remainder is 0, and if the number is odd, the remainder is 1 (or -1 for negative numbers).

Performing arithmetic operations becomes useful when instead of numbers we use variables.

Listing 2-14: Adding numbers and variables

```
var mynumber = 5  
var total = mynumber + 10 // 15
```

The example of Listing 2-14 declares the **mynumber** variable and initializes it with the value 5. In the following statement, the **total** variable is declared and initialized with the result of the addition of the number 10 plus the current value of **mynumber**. When the system processes this instruction, it reads the current value of the **mynumber** variable first, adds it to 10, and assigns the result to **total** (15).

In Listing 2-14 we use a new variable to store the result of the operation, but when the old value is not important anymore, we can store the result back into the same variable.

Listing 2-15: Performing operations on the variable's current value

```
var mynumber = 5  
mynumber = mynumber + 10 // 15
```

Here the current value of **mynumber** is added to 10 and the result is assigned to the same variable. After the execution of the second statement, the value of **mynumber** will be 15.

IMPORTANT: As in mathematics, the - sign may be used before the name of a variable to toggle its sign and get the opposite result. If the current value is positive, the result will be negative, and vice versa. For example, assuming that the current value of **mynumber** is 20, the statement **var result = -mynumber** will assign the value -20 to **result**. Another useful application of this technique is changing the current sign of the value of a variable assigning to the variable its own value preceded by the - sign (e.g., **mynumber = -mynumber**).

Working with values previously stored in a variable allows our program to evolve and adapt to new circumstances. For example, this is a good way to create a counter. We could add 1 to the current value of a variable and store the result in the same variable. Every time a statement like this is executed, the value of the variable is incremented by one unit. Recurrent increments and decrements of the values of variables are actually very important in computer programming. Because of this, Swift supports operators that were specifically designed for this purpose.

- **++** increments the value of the variable by 1. It produces the same result as **variable = variable + 1**.

- `—` decrements the value of the variable by 1. It produces the same result as `variable = variable - 1`.
- `+=` is a shorthand for `variable = variable + number`, where `number` is the value we want to add to the variable's current value.
- `-=` is a shorthand for `variable = variable - number`, where `number` is the value we want to subtract to the variable's current value.

The difference between the operators `++` and `+=` is that the first one always adds 1 to the current value of the variable, while the last operator allows us to add whatever value we want. Here is an example.

Listing 2-16: Modifying the variable's value using different incremental operators

```
var mynumber = 5
mynumber++
mynumber += 4 // 10
```

The process generated by the code of Listing 2-16 is straightforward. After the value 5 is assigned to the `mynumber` variable, the system reads the second statement, gets the current value of the variable, adds 1 to that value, and stores the result back to `mynumber`. The same process is repeated for the third statement, although this time we use the `+=` shorthand to be able to add a value different than 1.

The `++` and `—` operators not only increment or decrement the value of a variable, they also return the result. This is useful when we want to check the value after the operation or store the result in another variable.

Listing 2-17: Assigning the current value to another variable

```
var mynumber = 5
var control = 0
control = mynumber— // 5
```

After initializing the variables `mynumber` and `control`, the code of Listing 2-17 decrements the value of `mynumber` by 1 and assigns the value returned to `control`. If we check the values of these two variables at the end, we will see that `mynumber` is equal to 4 and `control` is equal to 5. This might sound strange, since we subtracted 1 to the value of `mynumber` and then stored the result in `control`, the value of both variables should be 4. The reason why we didn't get the expected result is because the value returned by the operation is different according to the position of the operator. If the operator is placed after the name of the

variable, like in the example of Listing 2-17 (`mynumber—`), the current value is returned first and then decremented by 1, but if the operator is placed before the name of the variable (`—mynumber`), the current value is decremented first and the new value is returned, as in the following example.

Listing 2-18: Assigning the result to another variable

```
var mynumber = 5
var control = 0
control = —mynumber // 4
```

Do It Yourself: To test these examples, replace the code in your Playground by the one you want to try and add statements at the end with the name of the variables to be able to see their final values on the Results Side Bar (as shown in Listing 2-4).

As we explained before, the possible values to store in an integer variable or constant are limited by their data type. If we perform an operation that assigns to an integer variable a result out of the range of values it can handle, the application crashes. This is called *overflow* and Swift offers additional operators to avoid it. These operators are called *Overflow Operators* and are declared as normal operators but prefixed by the pound character (`&+`, `&-`, `&*`, `&/` and `&%`).

Listing 2-19: Operations with Overflow Operators

```
var mynumber: Int8 = 120
mynumber = mynumber &+ 10 // -126
```

An overflow operator performs the calculation until it reaches the limit of the possible range and then keeps calculating from the other side of the range. For example, the maximum and minimum values a variable of type `Int8` can handle are -128 and 127. If we assign to a variable of this type an operation that produces a result higher than 127, the operator adds the rest of the operation to the number -128, keeping the result inside the range of values -128 and 127. The code of Listing 2-19 creates a value of this type. The variable is initialized with a value of 120 and then an operation adds 10 to that number. For a normal operator, the result would be 130, which is out of the range of the variable and would crash the app, but since we are working with an overflow operator, the addition returns the value -126. The operator adds units to the value until it reaches the maximum limit of the range (127) and then adds the rest to the minimum value the variable can take (-128), getting the final result (-126).

IMPORTANT: Overflow operators are useful when we think that an operation could overflow a variable, but most of the time is more practical to just check the value and perform the operation only when the result is within the possible range. In Chapter 3 we will study properties available for integer types that report the minimum and maximum values allowed in real time to prevent overflow.

Constants

As mentioned before, the memory of a computer is a sequence of switches. There are millions and millions of switches, one after another, with no clear delimitations. To be able to know where the space occupied by a variable starts or ends, the system has to use addresses. These addresses are just consecutive numbers that correspond to each byte of memory. For example, if one byte is at the address 000000, the next byte will be at the address 000001, the next one at 000002, and so on. If we declare a variable of 4 bytes, the system has to reserve the four consecutive bytes and remember where they are so as not to overwrite them with the value of another variable. The task is easy when working with primitive types because their sizes are constant, but the size of variables of more complex or custom data types depends on the values we assign to them. For example, the space in memory required to store the text “Hello” is smaller than the space required for the text “Hello World”. A variable will occupy less space with the first value and more with the second one. Managing the memory for data of inconsistent size takes more time and consumes more resources than working with fixed sizes. This is one of the reasons why Swift includes the concept of constants.

Constants are exactly the same as variables, but their values can’t change. Once a constant is declared and initialized, we can’t change its value. In fact, we will get an error if our code tries to do it, so we know we made a mistake even before running our app. Constants provide a safe way to store a value but also help the system to manage the memory, improving the performance of our application. These advantages make constants suitable for primitive and custom types alike. We may use constants to help the system manage the memory for complex data types and also create constants of primitive types to make sure a value is not changed during the execution of our program.

To declare a constant we have to apply the same syntax used for a variable but replace the `var` keyword by the `let` keyword.

Listing 2-20: Declaring and initializing a constant

```
let mynumber = 5
```

All the rules for variables also apply to constants; with the exception that we can’t assign a new value after the constant was already initialized. We can declare the constant

first, specifying its type, and initialize it later, but we can't assign new values to it. The `mynumber` constant declared in Listing 2-20 will always have the value 5.

IMPORTANT: When to use constants or variables depends on your application. As guidance, you can follow what Apple recommends: if a stored value in your code is not going to change, always declare it as a constant with the `let` keyword. Use variables only for storing values that need to be able to change.

2.3 Data Types

Besides the primitive types already studied, Swift defines an additional data type to allow us to work not only with numbers, but also other values such as logical values (true or false), characters, or text.

Characters

Because of their nature, computers can't store decimal numbers, characters, or text. As we have seen in the previous section, computer memory is only capable of storing 1s and 0s (switches on and off), but they can represent more complex values using tables that contain the information necessary to represent those values (numbers, letters, symbols, etc.). What the system stores in memory is not the character but the value corresponding to the index of the character on the table. For example, if we use the letter A, the value stored in memory will be the decimal number 65 (in its binary representation) because that's the position of the letter A on the table used by Swift. When the value is stored, the system not only stores the value but also a reference to indicate that this is a character and not just a number. This way, when we get the value back, the system reads the number 65, searches for the character in that position on the table and returns the letter or shows it on the screen.

There are several standard tables available. Swift is compliant with a table called *Unicode*. This is a very comprehensive table that includes every character from almost any language in the world, and also special characters, such as emoticons. Due to its broad range, the space in memory required to store a single character varies from one character to another. For this reason, Swift provides the **Character** type to store these values.

Listing 2-21: Declaring and initializing a Character variable

```
var myletter: Character = "A"
```

A character is declared using the **Character** type and initialized with the value between double quotes. In Listing 2-21 we declared a variable called **myletter** with the character A as its value.

The Basics: To be able to interact with a computer, we create layers of abstraction, one over another, until we obtain something similar to what we are used to working with in the real world. Represent a character through an index stored in memory as a series of switches on and off may be considered the first layer. Organizing those

characters to create instructions the computer can follow would be the second layer. As we will see in further chapters, the organization of these instructions into structures and objects and the association of elements in more complex structures represent new layers of abstraction. Each layer adds more functionality with the final purpose of creating something useful for the real world. To learn how to program an app, we have to understand the tools created by these layers and how to use them. We will go through this process in the rest of the book.

Strings

In a program we barely work with individual characters, we usually need to store a string of characters. The `String` type was created for this purpose.

Listing 2-22: Declaring and initializing a `String` variable

```
var mytext: String = "My name is John"
```

A string is actually a sequence of `Character` values. It is declared with the `String` type and its value between double quotes. These types of variables are very flexible. We may replace the string by another one of different length, concatenate two or more strings, or even modify parts of them. Concatenation is a common operation and it is done with the `+` and `+=` operators.

Listing 2-23: Concatenating strings

```
var mytext = "My name is "  
mytext = mytext + "John" // "My name is John"
```

In Listing 2-23, the `mytext` variable is created with the value `"My name is "` and then the string `"John"` is added at the end of the current value to get the string `"My name is John"`. The `+=` operator works in a similar way. We can even combine them to get the string we want.

Listing 2-24: Concatenating strings with the `+` and `+=` operators

```
let exclamation = "!"  
var mytext = "My name is "  
mytext += "John" + exclamation // "My name is John!"
```

The example of Listing 2-24 uses the `+=` operator to add a string at the end of the current value of the `mytext` variable. The final value will be “My name is John!”. Notice that the `exclamation` variable is considered by Swift to be a `String` value, not a `Character` value.

With the `+` and `+=` operators we can only concatenate strings with strings. To concatenate strings with characters and numbers we have to use a procedure called *String interpolation*. The variables have to be included inside the string and wrapped by parentheses prefixed with a backslash.

Listing 2-25: Including variables in strings

```
var age = 44
var mytext = "I am \(age) years old" // "I am 44 years old"
```

In the code of Listing 2-25, the `age` variable is read and its current value is added to the string. The string “I am 44 years old” is then assigned to `mytext`. Using this method we can insert inside a string any variable we want, including `Character` and `String` variables, as well as the result of mathematic operations.

Listing 2-26: Performing operations in strings

```
var age = 44
var mytext = "I am \(age * 12) months old" // "I am 528 months old"
```

In this last example, the value of `age` is multiplied by 12 and the result is included in the string. The value of `mytext` is “I am 528 months old”.

IMPORTANT: Processing strings of characters is probably the most important task of any program. Text is the tool used by the application to communicate with the user. For this reason, Swift includes a series of functions we can use to manipulate and work with strings. We will study these functions in Chapter 3.

Booleans

Boolean variables are a type of variable that can only have two possible values: `true` or `false`. These variables are particularly useful when we want to execute an instruction or a set of instructions only in certain conditions. To declare a Boolean variable we can specify the type as `Bool` or let Swift infer the type from its value, as in the following example.

Listing 2-27: Declaring a Boolean variable

```
var valid = true
```

The purpose of these variables is to simplify the process. If we use an integer variable to indicate the state of a condition, we have to remember which numbers we decided to use for the valid and the invalid states. Using a Boolean variable instead we just have to check whether the value is equal to **true** or **false**.

IMPORTANT: Booleans become useful when we use them along with instructions that allow us to perform a task or repetitive tasks according to a condition. We will study Conditionals and Loops later in this chapter.

Optionals

After a variable is declared we have to provide its initial value. We can't use a variable if it was not initialized. This means that a variable has a valid value all the time. But this is not always appropriate. Sometimes we don't have an initial value to assign to the variable or we need to indicate the absence of a value because the current one becomes invalid. For these situations, Swift defines a modifier that turns every data type into an optional type. This means that the variable marked as optional may have a value or be empty. To declare an optional, we just have to add a question mark after the type's name.

Listing 2-28: Declaring an optional variable of type Int

```
var mynumber: Int?
```

The empty state is represented by the keyword **nil**. Thus, when an optional variable is declared but not initialized, as in the example of Listing 2-28, Swift assigns to the variable the **nil** keyword to indicate the absence of a value. New values are assigned as we do with normal variables.

Listing 2-29: Assigning new values to optional variables

```
var mynumber: Int?  
mynumber = 5
```

If later we need to empty the variable, we have to assign the keyword **nil**.

Listing 2-30: Using `nil` to empty the variable

```
var mynumber: Int?  
mynumber = 5  
mynumber = nil
```

The code of Listing 2-30 declares an optional integer, assigns the value 5 to it and then declares the variable as empty with the keyword `nil`. Although optionals seem to work like regular variables, they don't expose their values. To read the value of an optional, we have to unwrap it by adding an exclamation mark at the end of the name.

Listing 2-31: Unwrapping an optional

```
var mynumber: Int?  
mynumber = 5  
var total = mynumber! * 10 // 50
```

The last statement of Listing 2-31 unwraps `mynumber` to get its value, multiplies this value by 10 and assigns the result to the `total` variable. This is only necessary when we need to use the value. If we just want to assign the optional to another optional, the assignation is as always.

Listing 2-32: Assigning an optional to another optional

```
var mynumber: Int?  
mynumber = 5  
var total = mynumber
```

In this last example, the system infers the type of the `total` variable to be an optional of type `Int` and assigns the value of `mynumber` to it. If we want to read the value of `total` later, we have to unwrap it as we did with `mynumber` before.

IMPORTANT: Before unwrapping an optional, we have to be sure it contains a value (it is not `nil`). If we try to unwrap an empty optional, the app will return an error and crash. Later in this chapter we will learn how to use conditional statements to check this condition.

There are times when we know that an optional will always have a value but we don't

know what the initial value is. For example, we could have a variable that receives a value from the system as soon as the application is executed. When the variable is declared in our code, we don't have a value to assign to it, but we know that the variable will have a value as soon as the user runs the app. For these situations, Swift provides the possibility to declare *implicitly unwrapped optionals*. These are optional variables declared with the exclamation mark instead of the question mark. The system reads these variables as optionals until they are assigned a value and therefore we don't have to unwrap them anymore.

Listing 2-33: Declaring implicitly unwrapped optionals

```
var mynumber: Int!  
mynumber = 5  
var total = mynumber * 10 // 50
```

In Listing 2-33, the **mynumber** variable was declared as an implicitly unwrapped optional and it was later initialized with the value 5. Notice that it was not necessary to write the exclamation mark when reading its value anymore. The implicitly unwrapped optionals are automatically unwrapped.

Tuples

A tuple is a type of variable that contains a group of one or more values of equal or different type. It is useful when we need to store values that are somehow related to each other. Tuples are declared with their values and types separated by a comma and enclosed between parentheses, as shown in Listing 2-34.

Listing 2-34: Declaring a tuple with two values

```
var myname: (String, String) = ("John", "Doe")
```

In this example, the **myname** variable is declared to be a tuple that contains two **String** values. The values of this tuple are of the same type, but we can use any combination of values we want and also let Swift infer their types.

Listing 2-35: Declaring a tuple with values of different type

```
var myname = ("John", "Doe", 44)
```

For further reading, an index is automatically assigned to each of the values of the tuple. Indexes start at the value 0, so the first value will be at index 0, the second at index 1, and so on. Using the corresponding index and dot notation we can access the value we want.

Listing 2-36: Reading a tuple

```
var myname = ("John", "Doe", 44)
var mytext = "\(myname.0) is \(myname.2) years old"
```

In Listing 2-36 we read the values of the `myname` tuple at index 0 and 2 to include them in a new string. The result is assigned to `mytext` ("John is 44 years old"). The same syntax may be used to modify a value.

Listing 2-37: Changing the value of a tuple

```
var myname = ("John", "Doe", 44)
myname.0 = "George"
var mytext = "\(myname.0) is \(myname.2) years old"
```

The second statement of Listing 2-37 assigns a new string to the first value of the tuple `myname`. The new value has to be of the same type as the old one. If we had tried to assign an integer instead, we would have received an error. At the end of this code, the value stored in `mytext` is "George is 44 years old".

Indexes are a quick way to access the values of a tuple, but they don't help us remember what the values represent. To facilitate the identification of the values in a tuple, we can assign a name to each one of them. The name has to include a colon at the end and be declared before the value.

Listing 2-38: Declaring names for the values of a tuple

```
var myname = (name: "John", surname: "Doe", age: 44)
var mytext = "\(myname.name) is \(myname.age) years old"
```

Sometimes the values are required more than once in our code and the dot notation may not be practical. For these situations, Swift provides a way to copy the values of the tuple into independent variables.

Listing 2-39: Creating multiple variables from the values of a tuple

```
var myname = ("John", "Doe", 44)
var (name, surname, age) = myname
var mytext = "\ (name) \ (surname) is \ (age) years old"
```

The names of the variables are declared in the same statement between parentheses and the tuple is assigned to this construction. The values are assigned to the variables in the same order they are presented. These variables may be used later as we would do with any other variable. If only some of the values are required, the rest may be ignored writing an underscore in their place.

Listing 2-40: Reading some of the values of a tuple

```
var myname = ("John", "Doe", 44)
var (name, _, age) = myname
var mytext = "\ (name) is \ (age) years old"
```

Only the variables **name** and **age** are used in this last example (notice the underscore in the place of the second variable). The string assigned to **mytext** is "John is 44 years old".

IMPORTANT: Tuples are particularly useful when working with functions, as we will see later. The use of tuples is only recommended when we need to share information between different pieces of code, but not to store persistent data. To store complex data structures you should use Collections (such as arrays or dictionaries). We will study Collections in the next section of this chapter and functions in Chapter 3.

2.4 Collections

Most of the time, instead of independent units of information, programs require lists of related values. These lists are called *Collections*. From the options of a menu to the books in a library, any useful information in the real world may be organized as a collection of values. Swift provides three types of collections: Arrays, Sets and Dictionaries.

Arrays

Arrays are variables that contain an ordered list of values. They have the capacity to store all the values we want, but they all have to be of the same type. For example, if we create an array of type `Int`, we will only be able to store values of type `Int`. The declaration of an array includes the type and the values enclosed in square brackets.

Listing 2-41: Declaring arrays

```
var list: [Int] = [15, 25, 35]
```

Again, Swift may infer the type from the values.

Listing 2-42: Declaring arrays with type inference

```
var list = [15, 25, 35]
```

The `list` array declared in these examples was initialized with three integer values, 15, 25 and 35. The values of an array are usually called *elements* or *items*. Following these terms we can say that the code of Listing 2-42 declares an array of three elements of type `Int`.

An index is assigned to each of the values automatically, starting from 0, but arrays don't use the dot notation applied for tuples, the index of the element we want to read has to be declared after the array's name, enclosed in square brackets.

Listing 2-43: Reading the array's elements

```
var list = [15, 25, 35]
```

```
list[1] // 25
```

The last statement of Listing 2-43 shows the value of the second element of the `list` array

(the element at index 1) on the Results Side Bar. We can also use indexes to modify the values of an array, as we did with tuples.

Listing 2-44: Assigning a new value to an element

```
var list = [15, 25, 35]
list[0] = 400
list // [400, 25, 35]
```

Assigning new values is only possible for elements that already exist in the array. To add a new element, or several, we can use the `+=` operator.

Listing 2-45: Adding new elements to an array

```
var list = [15, 25, 35]
list += [45, 55]
list // [15, 25, 35, 45, 55]
```

The `+=` operator adds an array at the end of another array. In Listing 2-45 we use it to add two more elements to the array declared in the first statement. The final array contains the values 15, 25, 35, 45, and 55, in that order.

IMPORTANT: The elements added to the array have to be of the same type of the original array (in this last example, they have to be of type `Int`). If you try to add elements of different type, Xcode will show you an error.

What the `+=` operator actually does is to concatenate two arrays and assign the result back to the same variable. If we want to use two or more arrays to create a new one, we can apply the `+` operator.

Listing 2-46: Concatenating two arrays

```
var list1 = [15, 25, 35]
var list2 = [45, 55, 65]
var final = list1 + list2 // [15, 25, 35, 45, 55, 65]
```

It is possible to declare arrays of arrays. These types of arrays are called multidimensional arrays. Arrays inside arrays are listed separated by comma, as any other value.

Listing 2-47: Creating multidimensional arrays

```
var list: [[Int]] = [[2, 45, 31], [5, 10], [81, 12]]
```

The example of Listing 2-47 creates an array of arrays of integers (notice the declaration of the array `[Int]` inside another array `[]`). To access the values we have to declare the indexes of each level between square brackets, one after another. The following example returns the first value (index 0) of the second array (index 1). The instruction looks for the array at index 1 and then looks for the number at index 0.

Listing 2-48: Reading values from a multidimensional array

```
var list: [[Int]] = [[2, 45, 31], [5, 10], [81, 12]]  
list[1][0] // 5
```

To create a new empty array or remove all its elements, we have to assign square brackets with no values to the variable.

Listing 2-49: Removing the elements of an array

```
var list = [15, 25, 35]  
list = []
```

If we use empty square brackets to declare a new array, we have to specify the element's data type (e.g., `var list: [Int] = []`). Another way to create an empty array is to declare the value of the variable as the data type followed by parentheses, as in `var list = [Int]()`. Elements may be added later using the `+=` operator (Listing 2-45) or applying any of the functions available for this purpose (we will study functions and array functions in Chapter 3).

The Basics: There is a low-level definition for array structures that requires the keyword `Array` and the data type between angle brackets. The syntax to create an array is `Array<Type>`, where `Type` is the data type of the elements of the array. For example, to create an empty array of integers we write `var myarray = Array<Int>()`. If the types were already defined, we can just use the empty parentheses, as in `var myarray: [Int] = Array()`.

Sets

Like arrays, sets are also collections, but the difference is that the order of the elements in a set is not defined. For example, if we store two elements in the array, one element will automatically receive the index 0 and the other the index 1. This correlation between indexes and values never change, allowing elements to be listed always in the right order and also have elements with the same value in different positions (indexes). Sets are similar to arrays, but they don't assign an index to their values and therefore there is no order and no duplicated values. This makes accessing values in sets much faster than arrays, but also makes them only useful in very specific circumstances.

The declaration of sets is similar to arrays, but we have to specify the type as `Set`.

Listing 2-50: Creating a set of integers

```
var ages: Set = [15, 25, 35, 45]
```

The elements' data type is inferred by Swift from the values assigned to the variable. If we need to declare the type, for example, to create an empty set, we have to use the set's low-level definition that includes the `Set` keyword followed by the data type between angle brackets.

Listing 2-51: Creating an empty set of integers

```
var ages: Set<Int> = []
```

IMPORTANT: Because sets' elements are not associated to an index, there is not much we can do with the set alone. Later in this chapter we will see how to interact through a collection with loops, and in Chapter 3 we will learn more about sets and how we can read and process their values.

The Basics: Sets are also defined as structures and they include their own initializer. To create an empty set we have to write parentheses after the set's declaration (e.g., `var myset = Set<Int>()`).

Dictionaries

There is only one way to access elements of an array independently and that is through their automatic indexes. Dictionaries offer a better alternative. With dictionaries we can define the indexes ourselves using any custom value we want. Each index has to be

explicitly declared along with the value that it is associated to. In the declaration, we have to specify the data type of both the indexes and their values. Listing 2-52 shows the corresponding syntax.

Listing 2-52: Declaring a dictionary

```
var list: [String: String] = ["First": "Apple", "Second": "Orange"]
```

Of course, if the indexes and the values are of a clear type, Swift can infer the types.

Listing 2-53: Declaring a dictionary with type inference

```
var list = ["First": "Apple", "Second": "Orange"]
```

If we want to change a value, we just have to declare its index in square brackets after the name of the dictionary.

Listing 2-54: Assigning a new value to a dictionary's element

```
var list = ["First": "Apple", "Second": "Orange"]  
list["Second"] = "Banana"
```

The second statement of Listing 2-54 assigns a new value to the element identified with the keyword "Second". Now the dictionary contains two elements with the values "Apple" and "Banana". If the keyword used to assign the new value exists, the system updates the value, but if the keyword doesn't exist, a new element is created.

Listing 2-55: Adding a new element to a dictionary

```
var list = ["First": "Apple", "Second": "Orange"]  
list["Third"] = "Banana"
```

In this last example, the second statement assigns the value "Banana" to a keyword that doesn't exist. The system creates the new element with the specified keyword and value. The result is a dictionary containing three values "Apple", "Orange", and "Banana".

Dictionaries return optional values. If we try to read an element with a keyword that doesn't exist, the value returned is `nil`.

Listing 2-56: Reading an element that doesn't exist

```
var list = ["First": "Apple", "Second": "Orange"]  
list["Third"] // nil
```

The code of Listing 2-56 tries to read a value with the keyword “Third” that doesn’t exist in the `list` dictionary. As a result, the value `nil` is shown on the Results Side Bar.

Dictionaries may also contain array as values. The declaration is simple, the key is declared as always, and the single value is replaced by an array.

Listing 2-57: Combining dictionaries and arrays

```
var fruits: [String: [String]] = ["A": ["Apple", "Apricot"], "B": ["Banana", "Blueberries"]]
```

Reading the values of a dictionary like this is a little bit more complicated. Because dictionaries return optionals, we can’t just specify the key and index as we do for multidimensional arrays (see Listing 2-48). The value returned by the dictionary has to be unwrapped before accessing its values.

Listing 2-58: Reading dictionaries with arrays

```
var fruits: [String: [String]] = ["A": ["Apple", "Apricot"], "B": ["Banana", "Blueberries"]]  
let list = fruits["A"] // ["Apple", "Apricot"]  
list?[0] // "Apple"
```

In Listing 2-58 we create a dictionary with two values. The values are arrays of strings with a string as key. The code gets the array corresponding to the “A” key, stores it in the `list` constant and then unwraps it to read its values. The value returned by the last statement is “Apple”.

Since dictionary values are treated as optionals, we can also assign the value `nil` to remove an element. The following example removes the element with the keyword “First”.

Listing 2-59: Removing an element

```
var list = ["First": "Apple", "Second": "Orange"]  
list["First"] = nil
```

As with arrays, to declare a new empty dictionary or remove all its elements, we have to assign to the variable the square brackets with no values (the colon is required for dictionaries).

Listing 2-60: Removing all the elements of a dictionary

```
var list = [{"First": "Apple", "Second": "Orange"}]  
list = []
```

If we use empty square brackets to declare a new dictionary, we have to specify the data types for its keywords and values (e.g., `var list: [String: String] = []`). Another way to create an empty dictionary is to declare the value of the variable as the data types followed by parentheses, as in `var list = [String: String]()`. Elements may be added later by declaring a new keyword (Listing 2-55) or by applying any of the functions available for this purpose (we will study functions and dictionary functions in Chapter 3).

The Basics: As arrays and sets, dictionaries are also defined as structures. There is a low-level definition for dictionary structures that requires the keyword **Dictionary** and the data types between angle brackets. The syntax to create a dictionary is **Dictionary<Type1, Type2>**, where **Type1** is the data type for the indexes and **Type2** is the data type for the values. For example, to create an empty dictionary of strings we write `var list = Dictionary<String, String>()`. If the types were already defined, we can just use the empty parentheses, as in `var list: [String: String] = Dictionary()`.

2.5 Conditionals and Loops

Up to this point we have been writing instructions in a sequence, one after another. In this kind of program, the system executes each statement once, and in the order they were presented. It starts with the first one and finishes with the last one on the list. The purpose of Conditionals and Loops is to break this sequential flow. Conditionals allow us to execute one or more instructions only in certain conditions, and Loops let us execute a group of instructions over and over again until a condition is satisfied.

If and Else

The most simple but handful conditional statement available in Swift is `if`. With `if` we can check a condition and execute a group of instructions only when the condition is true. The instructions that are going to be executed have to be declared between braces after the condition.

Listing 2-61: Comparing two values with if

```
var age = 19
var message = "John is old"
if age < 21 {
    message = "John is young"
}
message // "John is young"
```

Two variables are declared in the code of Listing 2-61. The `age` variable contains the value we want to check and the `message` variable is the one we modify depending on the state of the condition. The `if` statement compares the value of `age` with the number 21. This comparison returns the state of the condition (true or false). If the condition is true (`age` is less than 21), the instruction between braces is executed, assigning a new value to the `message` variable, otherwise the instruction is ignored and the execution continues with the instructions that follow the `if` statement.

The Basics: The instruction between braces in the last example is displaced to the right. The whitespace on the left is used to help us differentiate the statements between braces from the rest of the statements. This whitespace is automatically generated for you by Xcode, but you can do it yourself when necessary pressing the Tab key on your keyboard.

The < symbol we used in the last example to compare values is part of a group of operators called *comparison operators*. Here is a list with the most frequently used:

- `==` checks whether the value on the left is equal to the value on the right.
- `!=` checks whether the value on the left is different than the value on the right.
- `>` checks whether the value on the left is greater than the value on the right.
- `<` checks whether the value on the left is less than the value on the right.
- `>=` checks whether the value on the left is greater or equal to the value on the right.
- `<=` checks whether the value on the left is less or equal to the value on the right.

Any of these operators may be applied the same way we did in the previous example. For instance, the following code changes the value of the `message` variable when the value of `age` is less or equal to 21.

Listing 2-62: Comparing two values with the <= operator

```
var age = 19
var message = "John is old"
if age <= 21 {
    message = "John is young"
}
message // "John is young"
```

The Basics: Braces are applied in many constructions on Swift. They allow us to create independent processing units, with their own instructions and variables. These constructions are called *blocks*. We will learn more about blocks in this and following chapters.

When only two conditions are expected, we may define them by using a Boolean value. These values don't need to be compared; they already return a state (true or false).

Listing 2-63: Conditions with Boolean values

```
var underage = true
var message = "John is allowed"
if underage {
    message = "John is under age"
}
message // "John is under age"
```

The code of Listing 2-63 checks whether the value of the **underage** variable is **true** or **false**. If it is **true**, a new string of characters is assigned to the **message** variable. If, on the other hand, we want to execute the statements when the value is **false**, we have to write an exclamation mark before the condition to toggle its state. The value of the **message** variable in the following example is not modified.

Listing 2-64: Using logical operators

```
var underage = true
var message = "John is under age"
if !underage {
    message = "John is allowed"
}
message // "John is under age"
```

The exclamation mark is part of a group of logical operators provided by Swift.

- **!** (logical NOT) toggles the state of the condition. If the condition is true it returns false, and vice versa.
- **&&** (logical AND) checks two conditions and returns true if both are true.
- **||** (logical OR) checks two conditions and returns true if one or both are true.

Logical operators work with any kind of conditions, not only Booleans. To work with complex conditions, it is recommended to enclose the condition between parentheses.

Listing 2-65: Using logical operators to check several conditions

```
var smart = true
var age = 19
```

```

var message = "John is under age or dumb"
if (age < 21) && smart {
    message = "John is allowed"
}
message // "John is allowed"

```

The `if` statement of Listing 2-65 compares the value of the `age` variable to 21 and checks the value of the `smart` variable. If `age` is less than 21 and `smart` is `true`, then the overall condition is true and a new string is assigned to `message`. If any of the individual conditions are false, the overall condition will be false and the block of instructions won't be executed.

The Basics: Using `&&` (AND) and `||` (OR) you can create a logical sequence. The system evaluates one condition at a time and compares the results sequentially. For example, a set of conditions that evaluate as `true && false || true`, return the value `true`. The first expression is evaluated as `false` (`true && false = false`), but the last value is evaluated against the first result, so the final result is `true` (`false || true = true`). If you want to make sure that the expressions are evaluated in the correct order, you can enclose them between parentheses, as in `(true && false) || true`. The expression between parentheses is evaluated first, and the result is then evaluated against the rest of the expression.

Although we can use comparison operators and logical operators in most of the basic data types available, optionals are slightly different. Their values are wrapped, so we can't compare them with other values or check for their state, as we do with Booleans. Optionals have to be compared against the keyword `nil` first and then unwrapped before to work with their values.

Listing 2-66: *Checking whether an optional contains a value or not*

```

var count = 0
var myoptional: Int? = 5
if myoptional != nil {
    let uvalue = myoptional!
    count = count + uvalue
}
count // 5

```

Listing 2-66 shows the process we have to follow to read the value of an optional variable. The optional is checked first against `nil`. If it is different than `nil` (which means it

contains a value), we unwrap the optional inside the block of statements using an exclamation mark, assign its value to a constant, and use the constant to perform any operation necessary (in this example, the value of the `count` variable is incremented).

We always have to check if an optional has a value before unwrapping it. Because of this, Swift introduces a better syntax that checks the optional and unwraps its value at the same time. It is called *optional binding*.

Listing 2-67: Using optional binding to unwrap an optional variable

```
var count = 0
var myoptional: Int? = 5
if let uvalue = myoptional {
    count = count + uvalue
}
count // 5
```

This new code is cleaner and easy to read. The optional is unwrapped as part of the condition checked by the `if` statement. If it is different than `nil`, its value is assigned to the `uvalue` constant and the statements in the block are executed.

IMPORTANT: If we want to unwrap several optionals at the same time using optional binding, we have to declare the expressions separated by a comma (e.g., `let value1 = optional1, let value2 = optional2`).

There are times in which instructions have to be executed for either state of the condition (true or false). Swift includes the `if else` construction to help in this situation. The instructions are presented in two blocks. The first block will be executed when the condition is true, and the second block will be executed otherwise.

Listing 2-68: Using if else to respond to both states of the condition

```
var mynumber = 6
if mynumber % 2 == 0 {
    mynumber = mynumber + 2
} else {
    mynumber = mynumber + 1
}
mynumber // 8
```

The code of Listing 2-68 is a simple example that checks whether a value is odd or even using the remainder operator. The condition gets the remainder of the division between the value of the `mynumber` variable and 2, and compares the result against 0. If true, it means that the value of `mynumber` is even, so the first block is executed. On the other hand, if the value is different than 0 it means that it is odd, so the block corresponding to the `else` instruction is executed.

The statements `if` and `else` may be concatenated to check for as many conditions as we need. In the following example we check the age of a customer. The first condition checks whether its age is less than 21. If this is not true, the second condition checks whether the age is over 21. If this is not true, the final `else` block is executed.

Listing 2-69: Concatenating if else instructions

```
var age = 19
var message = "The customer is "
if age < 21 {
    message += "under age"
} else if age > 21 {
    message += "allowed"
} else {
    message += "21 years old"
}
message // "The customer is under age"
```

Switch

The `if` and `else` statements may include as many combinations as necessary, but soon this gets to a point where the code is impossible to read and maintain. When several conditions have to be checked it is better to use `switch` instead. The `switch` statement compares a value against a list of values and executes the instructions corresponding to the value that matches. The possible values are listed between braces using the `case` keyword, as in the following example.

Listing 2-70: Checking conditions with switch

```
var age = 19
var message = ""
switch age {
    case 13:
        message = "Happy Bar Mitzvah!"
```

```

case 16:
    message = "Sweet Sixteen!"
case 21:
    message = "Welcome to Adulthood!"
default:
    message = "Happy Birthday!"
}
message // "Happy Birthday!"

```

The cases have to be exhaustive; every possible value of the variable being checked has to be contemplated. If we don't include a **case** statement for every possible value, we have to add a **default** statement at the end that will be executed when there is no match.

In Listing 2-70 we compare the value of the **age** variable against a small set of values corresponding to special birthdays. If no **case** matches the value of the variable, the **default** statement is executed and the string "Happy Birthday!" is assigned to **message**.

When we have to execute the same set of instructions for more than one value, we can declare the values separated by comma.

Listing 2-71: Checking multiple conditions per case

```

var age = 6
var message = "You go to "
switch age {
case 1, 2, 3, 4:
    message += "Kindergarden"
case 5, 6, 7, 8, 9, 10, 11:
    message += "Elementary School"
case 12, 13, 14, 15, 16, 17:
    message += "High School"
case 18, 19, 20, 21, 22:
    message += "College"
default:
    message += "Work"
}
message // "You go to Elementary School"

```

The use of consecutive values is very common in computer programming. Swift includes two new operators to generate ranges of values and save us time.

- `...` (three dots) creates a range from the value on the left to the value on the right, including both values in the range (e.g., `1...4` creates a range that includes the values 1, 2, 3 and 4).
- `..<` (two dots and the less than character) creates a range from the value on the left to the value on the right, but only the value on the left is included (e.g., `1..<4` creates a range that includes the values 1, 2 and 3).

The following example applies range operators to the code of Listing 2-71.

Listing 2-72: *Using range operators in a switch statement*

```
var age = 6
var message = "You go to "
switch age {
  case 1...4:
    message += "Kindergarden"
  case 5...11:
    message += "Elementary School"
  case 12...17:
    message += "High School"
  case 18...22:
    message += "College"
  default:
    message += "Work"
}
message // "You go to Elementary School"
```

Ranges like `1...4` are the same as declaring the values 1, 2, 3, 4 for the case. If the value of `age` is within one of the ranges, the instructions for that `case` are executed.

IMPORTANT: Ranges can be made from different types of values, not only integers, and are especially useful when working with strings of characters. We will see more practical applications of ranges in Chapter 3.

While and Repeat While

The conditionals studied so far execute the statements corresponding to each of the states

of the condition only once. Sometimes the program requires executing a block of instructions several times until the condition is satisfied. This is called *loop*. The most simple alternative offered by Swift to create loops is the **while** statement (and its sibling **repeat while**).

The **while** statement checks a condition and executes the statements in its block while the condition is true. The following example initializes a variable with the value 0 and then checks its value in a **while** statement. If the value of the variable is less than 5, the statements inside the block are executed. After this, the condition is checked again. The loop keeps running until the condition becomes false (the value of the **counter** variable is equal or greater than 5).

Listing 2-73: *Using while to create a loop*

```
var counter = 0
while counter < 5 {
    counter++
}
counter // 5
```

If the condition returns false when it is checked for the first time, the statements in the block are never executed. If we want to execute the statements at least once, we have to use the **repeat while** construction.

Listing 2-74: *Using repeat while to create a loop*

```
var counter = 10
repeat {
    counter++
} while counter < 5
counter // 11
```

In this case, the initial value of the **counter** variable is declared as 10. This is greater than 5, but since we are using **repeat while**, the statements in the block are executed before the condition is checked, so the final value of counter will be 11 (its value is incremented once and then the condition returns false, ending the loop).

For

As well as **while**, **for** executes the statements in its block while the condition is true, but it

includes all the elements for the construction of the loop in just one sentence. The syntax is **for initialization; condition; increment {}**, where **initialization** declares the initial values for the loop, **condition** is the condition to be checked, and **increment** is an instruction that determines how the initial values will evolve in every cycle.

Listing 2-75: Using for to create a loop

```
var square = 2
for var counter = 0; counter < 5; counter++ {
    square = square * 2
}
square // 64
```

When the system executes the **for** loop of Listing 2-75 for the first time, it initializes the **counter** variable to 0, checks that its value is less than 5, and executes the statements in the block. After that first cycle is over, the last parameter is executed incrementing the value of **counter** by 1, and the condition is checked again. The loop continues until **counter** reaches a value of 5, which makes the condition false (5 is not less than 5).

For In

This is a more specific version of the **for** loop called **for in**. Its purpose is to iterate over collections of elements such as arrays, dictionaries, and also strings of characters. During the execution of a **for in** loop, the system reads the elements of the collection one by one and assigns their values to a constant that may be read and processed by the statements inside the block (similar to how optional binding works). The condition that has to be satisfied for this loop to be over is reaching the end of the list.

Listing 2-76: Using for in to iterate through an array

```
var fruits = ["Banana", "Orange", "Apple"]
var message = "My fruits:"
for myfruit in fruits {
    message += " " + myfruit
}
message // "My fruits: Banana Orange Apple"
```

The syntax of a **for in** loop is **for constant in collection {}**, where **constant** is the name of the constant that we are going to use to capture the value of each element, and **collection** is the

name of the collection (array, dictionary, or set) that we want to iterate through. The code of Listing 2-76 uses an array called `fruits` with three elements and the `myfruit` constant to capture their values. The statement inside the block adds the value of each element to the `message` variable to get the final string “My fruits: Banana Orange Apple”.

The Basics: The constants used in the loops are not updated with new values; they are recreated in every cycle. That is why we are able to use constants instead of variables. If you need to change their values, you can create new variables inside the block and assign the constants to them or declare the constants for the loops as variables using the `var` keyword enclosed in parentheses (e.g., `for (var mycharacter) in mytext`).

When the constant is not required inside the block we can replace it by an underscore.

Listing 2-77: Iterating through an array without reading its values

```
var fruits = ["Banana", "Orange", "Apple"]
var counter = 0
for _ in fruits {
    counter++
}
var message = "I have \(counter) fruits" // "I have 3 fruits"
```

The iteration through dictionaries is a little bit different than arrays. Dictionaries return a tuple containing the index and the value of each element.

Listing 2-78: Using for in to iterate through a dictionary

```
var fruits = ["First": "Apple", "Second": "Orange"]
var message = "My fruits:"
for (myindex, myfruit) in fruits {
    message += " \(myindex)-\(myfruit)"
}
message // "My fruits: Second-Orange First-Apple"
```

The `for in` loop in Listing 2-78 reads the elements of the `fruits` dictionary one by one. In each cycle, the index and the value of the current element are returned as a tuple and assigned to the `myindex` and `myfruit` constants (the syntax used to create these constants is the same applied to create multiple values from a tuple in Listing 2-39). The values of these constants are added to the `message` variable, getting the final string “My fruits: First-

Apple Second-Orange”.

Control Transfer Statements

Loops sometimes must be interrupted, independently of the state of the condition. Swift offers the following instructions to break the execution flow of loops and conditionals.

continue—This instruction interrupts the current cycle and moves to the next. The system ignores the rest of the statements in the loop after the instruction is executed.

break—This instruction interrupts the loop. All the statements and any pending cycles are ignored after the instruction is executed. It is also useful to ignore cases in **switch** statements, as we will see later.

The **continue** instruction is applied when we don’t want to execute the rest of the statements in the loop but we want to keep the loop running.

Listing 2-79: Jumping to the next cycle of the loop

```
let mynumbers = [2, 4, 6, 8]
var total = 0
for number in mynumbers {
    if number == 6 {
        continue
    }
    total += number
}
print("The total is: \(total)") // "The total is: 14"
```

The **if** statement inside the **for in** loop of Listing 2-79 compares the value of **number** with the value 6. If the number currently returned by the loop is 6, the **continue** instruction is executed, the last statement inside the loop is ignored, and the loop moves on to the next element in the **mynumbers** array. In consequence, all the numbers of the array are added to the **total** variable except the number 6.

Unlike the **continue** instruction, the **break** instruction interrupts the loop completely, moving the execution flow to the statement after the loop.

Listing 2-80: Interrupting the loop

```
let mynumbers = [2, 4, 6, 8]
var total = 0
for number in mynumbers {
    if number == 6 {
        break
    }
    total += number
}
print("The total is: \ (total)") // "The total is: 6"
```

Again, the **if** statement of Listing 2-80 compares the value of **number** with the value 6, but this time it executes the **break** instruction when it finds a match. If the number currently returned by the loop is 6, the **break** instruction is executed and the loop is over, no matter how many elements are left in the array. In consequence, only the values located before the number 6 are added to the **total** variable.

The **break** instruction is also useful to cancel the execution of the **switch** statement. The problem with the **switch** statement in Swift is that the cases have to be exhaustive, which means every possible value has to be considered and every case has to contain at least one statement to perform the corresponding task. But most of the time this is not possible or necessary. Using the **break** instruction we can ignore the statements of the cases that are not applicable. For example, we can declare the cases for the values we need and then break the execution in the **default** case for the rest of the values we don't care about.

Listing 2-81: Ignoring values in a switch statement

```
var fruits = ["Banana", "Orange", "Apple"]
let myfruit = fruits[1]

switch myfruit {
    case "Banana", "Apple":
        print("You have the right fruit")
    default:
        break
}
```

The **for in** loop of Listing 2-81 includes a **switch** statement to check the value of the **myfruit** constant. If the value doesn't match any of the values in the **case** statement, the **break** statement stops the execution of **switch** and returns the control to the next statement.

Executing the **break** instruction in the **default** case of a **switch** statement is a very

common practice because it allows us to only consider the values we need and ignore the rest.

2.6 Enumerations

The primitive types studied in this chapter were designed to create variables capable of storing multiple values. For example, as long as we respect the limits of its type, we can store any consecutive integer in an `Int` variable, or any character in a `Character` variable, or any text of any length we want in a `String` variable. But Booleans taught us there is also the possibility to have data types with a restricted number of values allowed (in the case of Booleans, only `true` and `false`). Enumerations are a way to create custom data types with their own limited set of values. An enumeration type is like a Boolean type but with the possible values defined by the programmer.

Enumeration types are defined by the keyword `enum` and the values between braces. Each value is declared using the `case` keyword.

Listing 2-82: Defining an enumeration type

```
enum Number {  
    case One  
    case Two  
    case Three  
}
```

The example of Listing 2-82 defines an enumeration type called `Number` with three possible values: `One`, `Two`, and `Three`. The values may also be declared in just one case separated by a comma.

Listing 2-83: Declaring the values in one statement

```
enum Number {  
    case One, Two, Three  
}
```

An enumeration is a custom data type. We can't use this construction by itself, we have to create a variable of this type and assign to that variable one of the possible values.

Listing 2-84: Declaring a variable of type Number

```
enum Number {  
    case One, Two, Three  
}  
  
var mynumber: Number = Number.One
```

The word **Number** is like the word **Int**, just a name for the data type. Variables declared of this type may only have the values allowed by the type (**One**, **Two**, or **Three**). To assign a value we have to use the name of the enumeration and dot notation. The **mynumber** variable declared in Listing 2-84 is of type **Number** and has the value **One**.

Once the type of the variable was already defined, only the dot and the value are necessary to modify its value.

Listing 2-85: *Assigning a new value to mynumber*

```
enum Number {  
    case One, Two, Three  
}  
var mynumber = Number.One  
mynumber = .Two
```

In the example of Listing 2-85, we assign a new value to **mynumber** at the end. The value **.Two** may have been written as **Number.Two**. Both syntaxes are valid, but Swift infers that the value provided is of the same type as the variable's type, so it is not necessary to declare the data type anymore. Notice that we can also let Swift infer the type when the variable is declared.

Like Booleans, enumeration types may be used as signals to indicate a state that can be checked later to decide whether or not to perform a certain task. Therefore, they are frequently used with conditional statements. The following example checks the value of an enumeration variable with a **switch** statement. These statements are particularly handy when working with enumerations. Enumeration types have a limited set of values, making it easy to define a **case** for each one of them.

Listing 2-86: *Using switch with an enumeration type*

```
enum Number {  
    case One  
    case Two  
    case Three  
}  
var mynumber = Number.Two  
var message = ""  
  
switch mynumber {  
    case .One:  
        message = "The number is One"
```

```

case .Two:
    message = "The number is Two"
case .Three:
    message = "The number is Three"
}
print(message) // "The number is Two"

```

In Listing 2-86, the enumeration type **Number** is defined first. The **mynumber** variable is declared next to be of this type and the value **.Two** is assigned to it. Next, a **switch** statement compares the value of this variable with the three possible values of its type and assigns the corresponding string to **message**. Notice again that Swift is capable of inferring the type of the enumeration inside the switch statement, so it doesn't require us to specify the type **Number** every time we declare its values.

Associated Values

The members of an enumeration type can also have associated values. These values are assigned to each member as if they were normal variables. The values must be of a type designated during the declaration of the enumeration, as shown next.

Listing 2-87: Assigning raw values to enumeration members

```

enum Number: Int {
    case One = 1
    case Two = 2
    case Three = 3
}
var mynumber = Number.One

```

If we don't assign the values ourselves, Swift automatically assigns an integer to each member. The numbers start from 0, like the indexes of an array. Of course, values are not restricted to integers. The following example associates the values of **Number** to strings.

Listing 2-88: Assigning strings to enumeration members

```

enum Number: String {
    case One = "Number One"
    case Two = "Number Two"
    case Three = "Number Three"
}

```



```
}
```

```
var mynumber = Number.One
```

The Basics: Enumerations work in a similar way to structures. They provide additional functionality, including a property to read associated values, and can also include custom properties and methods. We will learn more about enumerations and how to read their values in Chapter 3.

Chapter 3

Swift Paradigm

3.1 Programming Paradigms

Programs wouldn't be very useful if we were only able to write them as a consecutive set of instructions or statements. At first this was actually the only way to write a program, but soon languages incorporated tools to allow programmers to group instructions together and execute them every time necessary, expanding the possibilities and improving code efficiency. The way instructions are organized is called *paradigm*. Different paradigms are now available, with the most common being the Object Oriented Programming paradigm, or OOP. This paradigm emerges from the construction and integration of independent processing units called *objects*. Swift adopts OOP but it doesn't focus as much on objects as other languages do. Instead, it expands other types of processing units called functions, structures, and enumerations, to conform a new paradigm called *Protocol Oriented Programming*, where all these units become equally important in the construction of an application and may have a relationship established by blueprints called *protocols*.

3.2 Functions

Functions are blocks of code identified by a name. They are similar to the loops studied in Chapter 2, with the difference that there is no condition to satisfy; the statements inside a function are executed every time the function is called. To call (execute) a function, we have to write its name followed by parentheses. Functions may be called from anywhere in the code and any time required. This completely breaks the sequential processing of a program. Once a function is called, the execution of the program continues with the statements inside the function (no matter where it is located in the code), and only returns back to the main program when the execution of the instructions is over.

Declaration of Functions

Functions are declared using the **func** keyword, a name, parentheses, and the code between braces.

Listing 3-1: Declaring and calling functions

```
var mynumber = 5
func myfunction() {
    mynumber = mynumber * 2
}
myfunction()
mynumber // 10
```

The code in Listing 3-1 declares the **mynumber** variable, initializes it with the value 5, and then declares a function called **myfunction()**. The statements in a function are only processed when the function is called, so after its declaration we call our function with the instruction **myfunction()**. Every time **myfunction()** is called, it multiplies the current value of **mynumber** times 2 and assigns the result back to the variable. When the processing of the statements inside the function is over, the execution returns to the statement after the call (in this case, the last statement of the code that shows the final value of **mynumber** on the Results Side Bar).

Do It Yourself: Replace the code in your Playground's file with the code of Listing 3-1 to see how it works. Repeat the same for the rest of the examples in this chapter.

As we already mentioned, once the function is declared, we can call it every time necessary and from everywhere in the program the function is accessible. For example, the following code runs a **for** loop that calls **myfunction()** a total of 5 times (while **counter** is less than 5). Every time the function is executed, **mynumber**'s current value is multiplied times 2, getting a final result of 160.

Listing 3-2: Calling functions from a loop

```
var mynumber = 5
func myfunction() {
    mynumber = mynumber * 2
}
for var counter = 0; counter < 5; counter++ {
    myfunction()
}
mynumber // 160
```

The functions from these examples are modifying the value of an external variable (a variable that was not declared inside the function). Creating a function that works with values and variables that don't belong to the function itself could be dangerous; some variables may be modified by accident from other functions, the function may be called before the variables were even declared or initialized, or the variables that the function tries to modify may not be accessible by the function (functions have limited scope, as we will see later). To make sure that a function processes the right values, they have to be sent to the function when it is called. The type of values the function can receive and the names they are going to take are specified within the function's parentheses separated by a comma. When the function is executed, these parameters are turned into constants that we can read inside the function to access their values.

Listing 3-3: Sending values to a function

```
func duplicate(number: Int) {
    let total = number * 2
    let message = "Result: \"(total)\" // \"Result: 10\"
}
duplicate(5)
```

In the example of Listing 3-3 we don't use external variables anymore. The value to be processed is sent when the function is called and then it is received by the function through its parameters. The parameters are declared within the function's parentheses with the same syntax used for constants or variables. We have to write the name and the data type separated by a colon. In this example, the function is declared with one parameter of

type `Int` called `number`. When the function is called later, the value between the parentheses of the call (5) is assigned to the `number` constant and then the value of the constant is multiplied times 2 and a string is generated with the result.

Functions may not only be called every time we need them, but also the values we provide to the function during the call (called *attributes*) may be different each time. This is important because it makes functions reusable. We could, for example, add a second call with another value at the end of the code to execute the `duplicate()` function once more and get a different result.

Listing 3-4: Sending different values to a function

```
func duplicate(number: Int) {  
    let total = number * 2  
    let message = "Result: \"(total)\""  
}  
  
duplicate(5)  
  
duplicate(25)
```

IMPORTANT: By default, parameters are declared as constants, but they can be declared as variables adding the `var` keyword before the parameter's name. This allows us to read and also assign new values to the parameter from the function's statements (e.g., `func name(var number: Int)`).

The constants and variables declared inside a function, like `total` and `message`, are not accessible from other parts of the code (they have different scopes, as we will study in the next section of this chapter). Using parameters we can allow a function to receive values, but the result produced by the processing of these values is trapped inside the function. To communicate this result to the rest of the code, functions can return a value using a special instruction called **return**.

The **return** instruction finishes the processing of the function, so we have to declare it at the end of the function or after all the statements required have been processed. The next example returns the result of the multiplication.

Listing 3-5: Returning the result

```
func duplicate(number: Int) -> Int {  
    let total = number * 2  
    return total  
}  
  
let result = duplicate(25)  
  
let message = "The result is \"(result)\" // \"The results is 50\""
```

When we create a function that returns a value, the type of the value returned has to be specified in the declaration after the parentheses with the syntax `-> Type`, where **Type** is just the data type of the value that is going to be returned by the function (the function can only return values of the type indicated in its definition).

In Listing 3-5 we modified the `duplicate()` function to be able to return an integer containing the result of the multiplication. The value returned takes the place of the call, so we can assign it to a variable or use it in a statement. This example declares the function first, then declares the `result` variable, and finally assigns to this variable a call to the `duplicate()` function with the value 25. The system doesn't really assign the call of the function to the `result` variable; what it does is execute the function right away and assign to the variable the value returned by the function instead.

The values received and returned by a function may be of any available type. Here is a similar example that takes an array of integers, adds them to another variable, and returns the result. Notice that we use a `for in` loop inside the function to iterate through the array, and this time the call to the function is included directly in the final string.

Listing 3-6: Sending an array to the function

```
func sumup(list: [Int]) -> Int {  
    var total = 0  
    for number in list {  
        total = total + number  
    }  
    return total  
}  
  
var numbers = [1, 2, 3, 4, 5]  
  
var message = "The result is: \$(sumup(numbers))" // "The result is: 15"
```

The `sumup()` function receives the `numbers` array, assigns it to the `list` constant (declared as an array of integers) and then it iterates through the array to add the values of its items to the `total` variable. At the end, the value of `total` is returned and included in the string.

The last example is expanded next to return a tuple containing two integers that represent the total amount of elements in the array and the result of the addition of their values.

Listing 3-7: Returning a tuple

```
func totalelements(list: [Int]) -> (Int, Int) {  
    var counter = 0  
    var total = 0
```



```

for number in list {
    total = total + number
    counter++
}
return (counter, total)
}

var numbers = [1, 2, 3, 4, 5]
var (amount, result) = totalelements(numbers)
var message = "Addition of \ (amount) elements: \ (result)"

```

Similar to the `sumup()` function declared before, the `totalelements()` function of Listing 3-7 receives an array (`list: [Int]`), but this time it returns a tuple containing two integers (`(Int, Int)`). The function not only adds the value of the elements in the array but also counts how many elements are added. The values inside the tuple returned are assigned to the variables `amount` and `result` and then incorporated in a string.

Sometimes the values we send to a function are not descriptive enough. For instance, a function may require a number, but that number could be anything from the total of employees to individual data like how many years they have been working for the company. Swift provides the alternative of assigning names to the parameters of a function to make their purpose explicitly clear. These names are called *external names*, and are declared before the name of the parameter, separated by a space.

Listing 3-8: Declaring external names

```

func duplicate(years number: Int) -> Int {
    let total = number * 2
    return total
}

let result = duplicate(years: 8)
let message = "The result is \ (result)" // "The results is 16"

```

The `duplicate()` function of Listing 3-8 declares an external name called `years` for the `number` parameter. From now on, every time we call the function, Xcode forces us to write the external name in front of the value, helping us remember what the purpose of that value is.

The name of the parameter (`number` in our example) is the one used by the statements of the function to access the value received, while the external name (`years`) is used when calling the function to identify the value.

The Basics: External names have to be included in every call made to the function. If

you don't do it, Xcode shows you an error. It may look hard to remember, but Xcode simplifies our work providing hints for the names and the types of values you should include in every call. All you have to do is press the Enter key to accept the suggestion and the Tab key to jump to the next argument to insert its value.

Swift automatically generates external names for every parameter using the parameters' names. If, for example, we have a parameter called **number**, Swift generates the external name **number** for that parameter. Like those declared explicitly, external names generated automatically also have to be specified in the call. But there is a catch. To preserve compatibility with Objective-C, the first external name is removed. That is why we didn't have to specify the external name until we declared it explicitly in the example of Listing 3-8. Things change when we define a function with two or more parameters. The external name for the first parameter is still removed, but we can't omit the name for the rest of the values.

Listing 3-9: Multiple parameters

```
func multiply(number1: Int, number2: Int) -> Int {  
    let total = number1 * number2  
    return total  
}  
  
let result = multiply(25, number2: 3)  
let message = "The result is \ \(result)" // "The result is 75"
```

Listing 3-9 defines a function called **multiply()** with two parameters: **number1** and **number2**. Both parameters are turned into external names, but Swift removes the first one, leaving only **number2** as required (**multiply(25, number2: 3)**). If we want the first parameter to also have an external name, we have to include it in the function's declaration as we did in Listing 3-8.

Listing 3-10: External names for every parameter

```
func multiply(number1 number1: Int, number2: Int) -> Int {  
    let total = number1 * number2  
    return total  
}  
  
let result = multiply(number1: 25, number2: 3)  
let message = "The result is \ \(result)" // "The result is 75"
```

The new **multiply()** function of Listing 3-10 declares the external name **number1** for the **number1** parameter. Now the call has to include both names, **number1** and **number2**, before the

values (**multiply(number1: 25, number2: 3)**). If, on the other hand, what we want is to remove an external name we have to define the external name as an underscore.

Listing 3-11: Removing external names

```
func multiply(number1: Int, _ number2: Int) -> Int {  
    let total = number1 * number2  
    return total  
}  
  
let result = multiply(25, 3)  
  
let message = "The result is \ \(result)" // "The result is 75"
```

In this last example, we preserved the behaviour by default for the first parameter and removed the external name for the second parameter. Now the call only has to include the values (**multiply(25, 3)**).

By default, Swift uses the parameter's name to define its external name, but we can use any name we want. For instance, in the following example we declare the first parameter's external name as **number** and the second parameter's external name as **times**. Now the function's call reads **multiply-number-times**, which is a more human readable instruction.

Listing 3-12: Declaring different external names

```
func multiply(number number1: Int, times number2: Int) -> Int {  
    let total = number1 * number2  
    return total  
}  
  
let result = multiply(number: 25, times: 3)  
  
let message = "The result is \ \(result)" // "The result is 75"
```

Although we can't create two or more functions with the same name, we can do it as long as their parameters are not the same. Functions with equal name but different parameters are considered by Swift to be different functions.

Listing 3-13: Declaring different functions with the same name

```
func multiply(number1: Int, number2: Int) -> Int {  
    let total = number1 * number2  
    return total  
}  
  
func multiply(number1: Double, number2: Double) -> Double {  
    let total = number1 * number2
```

```
    return total
}
let result = multiply(5.0, number2: 10.5)
let message = "The result is \(result)" // "The result is 52.5"
```

The two functions in Listing 3-13 have the same name but are considered to be different because their parameters are of different type. When we call the `multiply()` function with integer values, the first function is executed, but when the values provided are of type `Double`, the second function is the one that gets called. This is a simple way to create different versions of the same function to process values of any type or even different amount of values.

The Basics: As we mentioned before, the constants declared for the function to receive values are called *parameters*. On the other hand, the values we send to the function when we call it are called *attributes*. In these terms, we can say that a function's call has arguments, values of which are sent to the function and received by its parameters.

Every function we have defined so far requires the values to be specified in the call. We can't omit any of the values that the function expects to receive, but Swift let us declare a default value for any of the function's parameters and avoid this requirement.

Listing 3-14: Declaring default values for parameters

```
func sayhello(name: String = "Undefined") -> String {
    let text = "Your name is " + name
    return text
}
let message = sayhello("John") // "Your name is John"
```

The code of Listing 3-14 declares the function `sayhello()` with one parameter of type `String` called `name` and with the string "Undefined" as its default value. When the function is called without a value, the string "Undefined" is assigned to `name`.

Standard Functions

The main advantage of functions is that we can call them from any part of the program that has access to them and they will always perform the same operations. We don't even

need to know how the function does it, we just have to send to the function the values we want to be processed and read the result. Their statements can be modified, improved, or we can even completely change the operations they perform, without affecting the rest of the code. The only thing the rest of the code needs to know is what values the function can receive and what kind of values it returns, but how those values are processed inside the function becomes irrelevant. This characteristic allows programmers to share functions and also use pre-programmed functions provided by frameworks to incorporate additional functionality that could take them too much time to develop themselves.

Every feature of the Swift language we have implemented so far is included in a library called *Standard Library*. The standard library includes everything from operators to primitive types and even functions. These functions are accessible from everywhere in our code and perform basic but useful operations. The following is a list of the most frequently used.

print(string)—This function prints a string on the console. Xcode provides a small console at the bottom of its interface (also called command-line interface) to display text messages, errors, and provide a simple way to produce an output for small applications or testing code. It may be used during development to print temporary values of variables and fix bugs.

abs(value)—This function returns the absolute value of an integer (inverts negative values to always get a positive number). The value may be a number of any primitive data type.

max(value1, value2, ...)—This function compares two or more values and returns the largest.

min(value1, value2, ...)—This function compares two or more values and returns the smallest.

assert(condition, message)—This function stops the execution of the application if the condition defined by the **condition** attribute is false, and prints the message provided by the **message** attribute on the console. The **condition** attribute is any condition capable of returning true or false (or the Boolean values **true** or **false**), and the **message** attribute is a **String** value.

IMPORTANT: The list of functions included in Swift standard library is extensive. For more information and a complete list of the current standard functions, visit our website and follow the links for this chapter.

From the functions on our list, **print()** is probably the most useful. Up to this point we have relied on the Results Side Bar to see the values of constants and variables. But this

bar is only available on Playground; when we develop a real application, we have to use the Xcode console instead. And this is the exclusive purpose of the `print()` function. The function takes a variable or a formatted string and prints it on the console.

Listing 3-15: Printing values with `print()`

```
var absolutenumber = abs(-25)
print("The number is: \ (absolutenumber)") // "The number is: 25"
```

The code of Listing 3-15 uses the `abs()` function to get the absolute value of the number provided between parentheses. The last statement prints a string on the console with the result (if we try this code on Playground, the string is shown on the Results Side Bar).

The Basics: The absolute value, also called *modulus*, may be thought of as the distance between the value and 0. Therefore, the value returned by the `abs()` function is always a positive number equal to the value of its attribute without considering the sign.

The rest of the functions play a role in the construction of the application itself. For example, the `min()` and `max()` functions evaluate two or more values (including numbers and strings) and return the smallest or the largest, respectively.

Listing 3-16: Comparing values

```
var minimum = min(10, 15, 5) // minimum = 5
var maximum = max(10, 15) // maximum = 15
print("Minimum is: \ (minimum) and Maximum is: \ (maximum)")
```

Scopes

The conditionals, loops, and functions we have studied in this chapter have a thing in common: they all use blocks of code (statements between braces) to enclose their functionality. Blocks are independent processing units; they contain their own statements and variables. To preserve their independence and avoid conflicts between these units and the rest of the code, their variables and constants are isolated. Variables and constants declared inside a block are not accessible by other parts of the code; they can only be used inside the block they were created.

The space in the code where a variable is accessible is called *scope*. Swift defines two

types of scopes: the global scope and the local scope. The variables and constants outside a block have global scope, while those declared inside a block have local scope. The variables and constants with global scope are accessible from any part of the code, while those with local scope are only accessible from the statements inside the block they were created (and also the statements from blocks created inside their block). For better understanding, here is a practical example.

Listing 3-17: Using variables and constants of different scopes

```
var sum = 0.0
var myarray = [10.0, 20.0, 30.0, 40.0, 50.0]

func first() {
    let multi = 10.0
    sum += myarray[1] * multi
}

func second() {
    var myarray = [1.0, 2.0, 3.0, 4.0, 5.0]
    let multi = 3.5
    sum += myarray[2] * multi
}

first()
second()

print("Total: \$(sum)") // "Total: 210.5"
```

This example declares two variables in the global space, **sum** and **myarray**, and then two functions with some local constants. The constants and variables declared in the global space have global scope, and therefore they are accessible from everywhere in the code, but those declared inside functions or blocks of code have local scope and they are only accessible from the space where they were created. Therefore, the **multi** constant declared inside the **first()** function is only accessible from this function (neither the statements in the global space nor other blocks outside **first()** have access to it), but the global variable **sum** used in the next statement to store the result of the multiplication of the array's second value times the value of **multi** have global scope, so its value will be available everywhere.

The next function, **second()**, declares a new variable called **myarray**. This variable has the same name as **myarray** declared before in the global space, but they have different scopes and therefore they are two different variables. When we read the **myarray** variable in the **second()** function to add a new value to the **sum** variable, the variable accessed is the one declared inside the function because in that space this new variable has precedence over the global variable with the same name.

Do It Yourself: Replace the code in your Playground’s file with the code of Listing 3-17. Check the values on the Results Side Bar to understand how the variables are accessed and which variable is used each time.

As demonstrated by the last example, we can declare variables with the same name as long as they have different scopes. Another example is the `multi` constant declared inside the `second()` function. This constant has the same name as the one declared in the `first()` function, but it has a different scope and therefore it is a different constant.

Blocks of code, including those used to create functions, conditionals, and loops, have their own scope and know the variables that are available to them. Because of this, we can generate independent processing units that don’t interfere with the operations of other units but are still connected to the space in which they were declared. This characteristic is so important in computer programming that an additional tool called *closures* was included in Swift to take advantage of it.

Closures

As well as conditionals and loops, functions may also be declared inside other functions. Functions declared inside functions are called *nested functions*. Here is an example.

Listing 3-18: Declaring nested functions

```
func first() -> Int {  
    var counter = 0  
  
    func second() -> Int {  
        counter++  
        return counter  
    }  
    return second()  
}  
  
let total = first()  
print("The count is \"(total)\"") // "The count is 1"
```

There are two functions in Listing 3-18, but the `second()` function was declared inside the `first()` function. The `counter` variable is declared first, and then the `second()` function is declared and executed. The statements in the `second()` function increment the value of `counter` by 1 and return the result. The `first()` function takes this value and also returns it. In the last statement on the global space, the value returned by `first()` is assigned to the `total` variable

and printed.

The variables and constants declared inside a function are created and then destroyed when the execution of the function is over. The values returned by functions are just copies of the values of the variables. In our example, the value returned by both functions is just a copy of the final value of the `counter` variable. The variable is destroyed as soon as the last statement of the `first()` function is executed. And here is when nested functions become useful. If instead of the value returned by the function we return the function itself, what is returned is a reference to the function. This means that the function is not destroyed. Since the function is not destroyed, it preserves all the variables on its scope, and therefore we can keep accessing these variables, modify their values, etc.

The syntax required to return a function from another function is a little bit different from what we have seen so far. The possibility of a function to receive or return another function has to be specified in its parameters and return type. The syntax of a function type is `(parameters) -> Type`. In the following example we create a function called `first()` that returns another function (`second()`). The function returned doesn't receive any value and returns an integer, so the return type of the `first()` function is declared as `() -> Int` (this construction replaces the simple `Int` returned by the same function in the previous example).

Listing 3-19: Returning functions

```
func first() -> () -> Int {  
    var counter = 0  
    func second() -> Int {  
        counter++  
        return counter  
    }  
    return second  
}  
  
let incrementor = first()  
print("The count is \$(incrementor())") // "The count is 1"
```

A function is returned using its name without the parentheses. In the code of Listing 3-19, the `second()` function is declared inside the `first()` function as we did before, but instead of executing the `second()` function and returning the value it produces, we return the function itself declaring only its name without parentheses (`second`). In the global space, the function returned by `first()` is assigned to the `incrementor` variable, and this variable is used later to call the function.

The function returned preserves a reference to the `counter` variable. Because of this, the `counter` variable still exists and is accessible by the function assigned to `incrementor`. Every time we call the function using the instruction `incrementor()`, the `counter` variable is

incremented by 1. After the `second()` function is returned, the variables of the `first()` function are not accessible anymore (because the execution of the `first()` function is over), but the `second()` function can remember these variables and still have access to them. It is said that the variables are closed inside the function returned.

IMPORTANT: Every time the `first()` function is executed, it creates a new `counter` variable and a new `second()` function. This means you can use the `first()` function to create all the `second` functions you need. Each function returned by the execution of `first()` will have its own scope and its own copy of the `counter` variable.

The Basics: Functions are called using their names and the attributes between parentheses. We always have to write the parentheses when we want to execute a function and execute its statements, even when no attributes are required, but when we want to reference a function only the name is necessary.

The process of closing the variables and constants of a function is so useful that Swift offers a specific construction called *closure* to use instead of functions for this purpose. Closures are simple blocks of code with the syntax `{ (parameters) -> Type in statements }`. They are similar to functions (functions are actually closures with a name), but everything goes between the braces and the `in` instruction is included to separate the closure's data types from the statements.

Listing 3-20: Using closures

```
func first() -> () -> Int {  
    var counter = 0  
    return { () -> Int in return ++counter }  
}  
  
let incrementor = first()  
print("The count is \(incrementor())") // "The count is 1"
```

The example of Listing 3-20 demonstrates how easy it is to create closures and how they can simplify our code. There is no `second()` function declared inside the `first()` function anymore; instead, we declare a closure as the value to be returned by the function. Since functions and closures are blocks, they behave the same way, only the syntax changes. The rest of the code follows the same pattern as before: the closure returned is assigned to `incrementor` and it is executed through this variable every time we want to increment the value of `counter` by 1.

IMPORTANT: As well as with functions, you can call the `first()` function over and over again to create all the closures you need. Each one will have its own scope and its own copy of the `counter` variable.

Of course, a closure can also receive values. As functions, the closure's attributes are declared between the parentheses.

Listing 3-21: Defining closures with attributes

```
func first() -> (Int) -> Int {  
    var counter = 0  
  
    return { (number: Int) -> Int in  
        counter += number  
        return counter  
    }  
}  
  
let incrementor = first()  
print("The count is \$(incrementor(10))") // "The count is 10"
```

The new closure of Listing 3-21 receives an integer value and adds it to the `counter` variable. Thanks to this, now `counter` is not incremented by 1 unit but by the number provided every time the closure is executed (`incrementor(10)`). The attribute is declared in the closure's definition, but it also has to be specified in the function's return type (`(Int) -> Int`).

In this and previous examples we assigned the function or closure returned by the `first()` function to the `incrementor` variable. What is assigned to the variable is actually a reference to the function or closure, and this is why we can execute the function or the closure later from the variable. We are using the variable as the name for that specific block of code. This is common practice and can be applied anywhere in the code. For example, we could have assigned the closure to a variable inside the function and then return the value of that variable instead.

Listing 3-22: Assigning closures to constants and variables

```
func first() -> (Int) -> Int {  
    var counter = 0  
  
    let process = { (number: Int) -> Int in  
        counter += number  
        return counter  
    }  
  
    return process  
}
```

```
let incrementor = first()
print("The count is \n(incrementor(10))") // "The count is 10"
```

A great advantage of being able to assign closures to constants or variables is the possibility to initialize them with the result of complex processes or operations. The closure is assigned to the constant or variable and executed right away adding parentheses at the end of the declaration. When the system reads the statement, it executes the closure and assigns the value returned to the constant or variable.

Listing 3-23: Initializing constants and variables with closures

```
let fruits = ["Apple", "Orange", "Banana", "Apple"]

let process = { () -> Int in
    var counter = 0
    for fruit in fruits {
        if fruit == "Apple" {
            counter++
        }
    }
    return counter
}()

print("There are \n(process) apples") // "There are 2 apples"
```

The closure declared in Listing 3-23 counts the number of values in the array that are equal to the string “Apple” and returns the result. This is a simple task but the technique is usually applied to more complex processes such as loading content from a file or opening a database.

If the closure doesn’t receive any parameter, as in this last example, we can simplify its syntax declaring the type of the constant or variable and letting Swift infer the type of the value returned.

Listing 3-24: Simplifying the syntax of closures

```
let fruits = ["Apple", "Orange", "Banana", "Apple"]

let process: Int = {
    var counter = 0
    for fruit in fruits {
        if fruit == "Apple" {
            counter++
        }
    }
}
```

```
}  
    return counter  
}  
}  
print("There are \(process) apples") // "There are 2 apples"
```

IMPORTANT: Closures can be sent or returned from functions and used as any other value. In the next section of this chapter, we will study some of these alternatives and implement them in practical situations.

3.3 Structures

Structures are the essential part of the organizational paradigm proposed by Swift. They are custom data types that include not only the data but also the code in charge of processing that data. When we define a structure, what we are doing is declaring a data type that may contain variables and constants (called *properties*) and also functions (called *methods*). Later we can declare variables and constants of this type to store information with the characteristics defined by the structure. These values (called *instances*) are unique, each one with its own properties and methods.

Definition of Structures

To define a new structure we have to use the `struct` keyword and enclose the data and functionality between braces.

Listing 3-25: Declaring a structure type

```
struct Item {  
    var quantity: Double = 0  
    var name: String = "Not defined"  
    var price: Double = 0  
}
```

Listing 3-25 defines a structure called **Item** with three properties: **quantity**, **name**, and **price**. The definition by itself doesn't create anything; it is just delineating the elements of the type (also called *members*), like a blueprint used to create the real structures. What we have to do to store data of this new type in memory, as we would do with any other data type, is to declare a variable or constant of this type. The declaration is done as before, but the type is the name of the structure and the initialization value is a special constructor, usually called *initializer*, with the syntax `Name()` (where **Name** is, again, the name of the structure).

Listing 3-26: Declaring a variable of type Item

```
struct Item {  
    var quantity: Double = 0  
    var name: String = "Not defined"  
    var price: Double = 0  
}
```

```
var purchase: Item = Item()
```

The code of Listing 3-26 creates a variable of type `Item` that stores an instance of the `Item` structure containing the properties `quantity`, `name`, and `price`. The instance is created by the `Item()` initializer and then assigned to the `purchase` variable.

The Basics: It is recommendable to write the names of custom data types, such as structures and objects, in lower case letters but with every word beginning with a capital letter (e.g., `Item`, `TheItem`, `TheRedItem`, etc.). This is called Camel Case, and helps us differentiate the name of data types from other elements such as functions and variables.

The properties of a new instance always take the values declared in the structure's definition (0, "Not Defined" and 0), but we can change them as we do with any other variable. The only difference is that the properties are inside a structure, so every time we want to access them we have to mention the structure they belong to. The syntax is `variable.property`, where `variable` is the name of the variable that contains the structure and `property` is the name of the property we want to access.

Listing 3-27: Assigning new values to the properties

```
struct Item {  
    var quantity = 0.0  
    var name = "Not defined"  
    var price = 0.0  
}  
  
var purchase = Item()  
  
purchase.quantity = 2  
purchase.name = "Lamps"  
purchase.price = 10.50  
  
var total = purchase.quantity * purchase.price  
print("Invoice: \(purchase.quantity) \(purchase.name) $ \(total)")
```

In Listing 3-27 the structure and `purchase` variable are declared as before, but this time we let Swift infer their types. Once the instance is created, new values are assigned to its properties using dot notation. Dot notation is not only used to assign new values but also to read the current ones. In the last statements the value of the `quantity` property is multiplied by the value of `price` to get the total money spent (21.0), and then everything is printed on the screen ("Invoice: 2.0 Lamps \$ 21.0").

Do It Yourself: Replace the code in your Playground’s file with the code of Listing 3-27. Create another variable after **total** and assign to this variable the initializer of the **Item** structure to create another instance. Assign different values to the properties of each instance to learn how to work with different copies of the same structure.

Structures may be instantiated inside other structures, as many times as necessary, in an unlimited chain. The dot notation is extended in these cases to reach every element in the hierarchy.

Listing 3-28: Structures inside structures

```
struct Price {  
    var USD = 0.0  
    var CAD = 0.0  
}  
struct Item {  
    var quantity = 0.0  
    var name = “Not defined”  
    var price = Price()  
}  
var purchase = Item()  
  
purchase.quantity = 2  
purchase.name = “Lamps”  
purchase.price.USD = 10.50
```

Listing 3-28 defines two structures: **Price** and **Item**. The **price** property of the **Item** structure is now a property of type **Price**. Instead of storing a single value, now it can store a structure with two properties, **USD** and **CAD**, for American and Canadian dollars. When the **Item** structure is created and assigned to the **purchase** variable, the **Price** structure for the **price** property is also created with its values by default. By concatenating the names of the variables and properties containing these structures we can reach and modify any value we want. For example, the last statement in the code of Listing 3-28 access the **USD** property of the **price** structure inside the **purchase** structure to assign a new price to the item in American Dollars (**purchase.price.USD = 10.50**).

Methods

If we could only store properties, structures would be just a complex data type, like a tuple, but as we mentioned before, code may also be defined as part of the structure. This is done through functions. Functions inside structures are called *methods*, but their syntax is exactly the same.

Listing 3-29: Defining methods

```
struct Item {  
    var quantity = 0.0  
    var name = "Not defined"  
    var price = 0.0  
    func total() -> Double {  
        let cost = quantity * price  
        return cost  
    }  
}  
  
var purchase = Item()  
purchase.quantity = 2  
purchase.name = "Lamp"  
purchase.price = 10.50  
  
print("Total cost: \"(purchase.total())\"") // "Total cost: 21.0"
```

In Listing 3-29, a method was declared as part of the definition of the **Item** structure. The method calculates the total money spent in the transaction multiplying the value of the **quantity** property times the value of **price**. This is the same operation we have done before through an independent statement, but now it is part of the structure, ensuring that it is always accessible and we always get the right result (we don't have to write the operation over and over again, it is always part of the instance we are working with).

The dot notation also applies to methods. The syntax to execute a method is **variable.method()**, where **variable** is the name of the variable that contains the structure and **method** is the name of the method we want to call.

Since methods are functions, they can receive and return values. In the example of Listing 3-29, the **total()** method returns a value of type **Double** corresponding to the total cost of the transaction.

A method can read the values of the instance's properties, but cannot assign new values to them. If we want a method to be able to modify the values of the properties of its own instance, we have to use a special keyword called **mutating**. The **mutating** keyword has to be placed before the **func** keyword in the method's definition.

Listing 3-30: Assigning new values to properties from inside the structure

```

struct Item {
    var quantity = 0.0
    var name = "Not defined"
    var price = 0.0
    mutating func total() {
        let cost = quantity * price
        name += " $ \(cost)"
    }
}

var purchase = Item()
purchase.quantity = 2
purchase.name = "Lamp"
purchase.price = 10.50

purchase.total()
print("Your purchase: \(purchase.name)") // "Your purchase: Lamp $ 21.0"

```

The `total()` method of the `Item` structure in Listing 3-30 is declared as a mutating method to be able to assign a new value to the `name` property from its statements. Now the structure not only calculates the total amount of the transaction but also adds it to the name of the item. The last statements of this example call the `total()` method to prepare the string for `name` and then print the result on the screen.

Initialization

Every instance we create from the structure's definition has the purpose to store and process specific data. For example, we can create multiple instances of the `Item` structure defined in previous examples to store information about several products. Each product will have its own name and price, so the properties of each instance have to be initialized with the proper values. The initialization of an instance is a very common process, and is difficult to do if we have to assign the values one by one after the instance is created. For this reason, Swift provides different alternatives to initialize values. The most convenient is called *memberwise initializer*.

Memberwise initializers detect the properties of the structure and declare their names as external names. By declaring these external names we can provide the values for initialization between the parentheses of the initializer. The following code uses memberwise initializers to initialize an instance of the `Item` structure declared in previous examples.

Listing 3-31: Initializing properties

```
struct Item {  
    var quantity = 0.0  
    var name = "Not defined"  
    var price = 0.0  
}  
  
var purchase = Item(quantity: 2, name: "Lamp", price: 10.50)  
  
var total = purchase.quantity * purchase.price  
print("Invoice: \(purchase.quantity) \(purchase.name) $ \(total)")
```

The structure's definition doesn't change when we use memberwise initializers, but the values for the properties have to be declared between parentheses. This saves a lot of code and simplifies initialization. Also, if we use the memberwise initializer, we can ignore the values by default declared in the definition.

Listing 3-32: Using memberwise initializers to provide the initial values

```
struct Item {  
    var quantity: Double  
    var name: String  
    var price: Double  
}  
  
var purchase = Item(quantity: 2, name: "Lamp", price: 10.50)  
  
var total = purchase.quantity * purchase.price  
print("Invoice: \(purchase.quantity) \(purchase.name) $ \(total)")
```

The two forms of initialization we have seen so far (providing the values for the properties in the definition or using the memberwise initializer) are not customizable enough. Some structures may have multiple properties to initialize or even methods that have to be executed right away to get the proper values for the instance to be ready. To add more alternatives, Swift provides a specific method called `init()`. The `init()` method is called as soon as the instance is created, so we can use it to assign values to the properties or perform some operations and assign the results instead.

Listing 3-33: Initializing properties from the `init()` method

```
struct Price {  
    var USD: Double  
    var CAD: Double  
    init() {
```

```

    USD = 5
    CAD = USD * 1.09
}
}
var myprice = Price()

```

When the instance is generated by the initializer, the properties are created first and then the `init()` method is executed. Inside this method we can perform any operation we need to get the properties' initial values. In the example of Listing 3-33, we assign an initial value of 5 to the `USD` property and then multiply this value by the corresponding exchange rate to get the value for the `CAD` property (the same price in Canadian dollars).

As well as any other method or function, the `init()` method may include parameters. These parameters are added to be able to specify initial values from the initializer.

Listing 3-34: Declaring the `init()` method with parameters

```

struct Price {
    var USD: Double
    var CAD: Double
    init(americans: Double) {
        USD = americans
        CAD = USD * 1.09
    }
}
var myprice = Price(americans: 5)

```

The parameters of the `init()` method are automatically turned into external names, and therefore we have to mention their names when we declare the initializer's arguments. This is what is actually generated for us when we use memberwise initializers. Swift creates an `init()` method in the background with a parameter for every property available. The advantage of declaring the `init()` method ourselves is that we can specify only the parameters we need (as in the last example) or even declare multiple `init()` methods to present several alternatives for initialization.

Listing 3-35: Declaring multiple `init()` methods

```

struct Price {
    var USD: Double
    var CAD: Double
    init(americans: Double) {
        USD = americans
    }
}

```

```

    CAD = USD * 1.09
}
init(canadians: Double) {
    CAD = canadians
    USD = CAD * 0.91744
}
}
var myprice = Price(canadians: 5)

```

As we mentioned when we were studying functions, Swift identifies each function by its name and parameters, so we can declare several functions with the same name as long as they have different parameters. This, of course, also applies to methods, including the `init()` method. In the example of Listing 3-35, two `init()` methods were declared to initialize the instance of the structure. The first method receives a `Double` value with the name `americans` and the second method also receives a `Double` value but with the name `canadians`. The right method will be executed according to the attribute provided to the initializer. In this example, we use the attribute `canadians` with the value 5, so the instance is initialized by the second method.

Property Keywords

Swift defines keywords (also called *modifiers*) that can be applied to properties to confer them special attributes. There are several keywords available, but two are particularly useful when working with properties: `lazy` and `private`. The `lazy` keyword defines a property whose initial value is not assigned until the property is used for the first time, and the `private` keyword defines a property that is only accessible from the file it was created.

The `lazy` keyword is used when the property's value may take time to be determined and we don't want the initialization of the structure to be delayed. For example, we may have a property that stores a currency rate retrieved from a server. This is an expensive task that we should perform when the value is required and not every time a structure from this definition is initialized.

Listing 3-36: *Defining lazy properties*

```

struct Price {
    var USD: Double
    lazy var rate: Double = {
        return 1.09
    }()
}

```

```

init(money: Double) {
    USD = money
}
}
var purchase = Price(money: 3.5)

```

The **Price** structure of Listing 3-36 defines two properties: **USD** and **rate**. The **USD** property is a **Double** value that we use to store the price, and **rate** is a lazy property that returns the current currency rate. A closure is assigned to the **rate** property to get the current rate and return it, but because we declared the property as **lazy** the closure will only be executed when we try to read the property's value (in this example the value 1.09 is not assigned to the **rate** property because the property has not been used).

IMPORTANT: This is a simple example that returns a fixed value, but in practice the closure may include code that retrieves the current rate from a server, which takes time and consumes resources. If we only need instances to store the price it doesn't make sense to load the rate from the web every time an instance is created, and therefore we should declare the property as **lazy**. This keyword is not only useful when we load resources from the Internet but also when the property values are obtained from files or databases.

The other important keyword for properties is **private**. This keyword limits the access to the property to the file where the structure was defined. As we will see later, an application is composed of multiple files, each with its own code. If we want to make sure that a property is only modified by the code of its structure, we can define that structure in a separate file and declare the property as **private**. For instance, we can set the **rate** property of our **Price** structure to be **private** so it can only be accessed by the methods in the structure.

Listing 3-37: *Declaring private properties*

```

struct Price {
    var USD: Double
    private var rate = 1.09
    init(money: Double) {
        USD = money
    }
    func getCAD() -> Double {
        let total = USD * rate
        return total
    }
}

```

```
}  
var purchase = Price(money: 3.5)
```

The code of Listing 3-37 defines the `rate` property as private and adds a method called `getCAD()` to use the property's value. Because the property is private, codes defined outside the file where the structure's definition is contained can't read or modify its value, but they can still calculate the total of Canadian dollars by calling the `getCAD()` method.

IMPORTANT: The `private` keyword limits access to a property from one file to another. We will learn how to work with multiple files in Chapter 5, how to create Swift files in Chapter 13 (see Listing 13-22), and how to implement the `private` keyword in practical applications in Chapter 21 (see Listing 21-22).

Computed Properties

The properties we have declared up to this point are called *Stored Properties*. Their function is to store a value in memory. But there is another type of property called *Computed Properties*. These properties don't store a value of their own, instead they have access to the rest of the properties of the structure and can perform operations to set and retrieve their values.

Two special methods were included for computed properties to be able to set and retrieve a value: `get` and `set`. These methods are also called *getter* and *setter*, and are declared between braces after the property's name. Although both methods are useful, only the `get` method is mandatory. Here is a simple example.

Listing 3-38: Declaring computed properties

```
struct Price {  
    var USD: Double  
    var ratetoCAD: Double  
    var canadians: Double {  
        get {  
            let price = USD * ratetoCAD  
            return price  
        }  
    }  
}  
  
var purchase = Price(USD: 11, ratetoCAD: 1.09)
```

```
print("Price in CAD: \$(purchase.canadians)") // "Price in CAD: 11.99"
```

The structure defined in Listing 3-38 contains a stored property called **USD** to store the price in American dollars, a stored property called **ratetoCAD** to store the exchange rate for Canadian dollars, and a computed property called **canadians** that converts the US dollars into Canadian dollars and returns the result. The advantage of computed properties over methods is that the value is calculated every time the property is read. No matter if a process changes the value of the **ratetoCAD** property, the **canadians** property will always return the right price in Canadian dollars.

Including the **set** method for the **canadians** property we can, for example, set a new price using the same currency.

Listing 3-39: Adding the `set` method to set a new value

```
struct Price {  
  var USD: Double  
  var ratetoCAD: Double  
  var ratetoUSD: Double  
  var Canadians: Double {  
    get {  
      return USD * ratetoCAD  
    }  
    set {  
      USD = newValue * ratetoUSD  
    }  
  }  
}  
  
var purchase = Price(USD: 11, ratetoCAD: 1.09, ratetoUSD: 0.91744)  
  
purchase.canadians = 500  
  
print("Price: \$(purchase.USD)") // "Price: 458.72"  
print("Price in CAD: \$(purchase.canadians)") // "Price in CAD: 500.0048"
```

The new structure defined in Listing 3-39 can now retrieve and also set a price in Canadian dollars. When we set a new value for the **canadians** property, the value is stored in a constant called **newValue** (the constant is created automatically for us). Using this constant we can process the value received and perform the operations we need. In this example, the value of **newValue** is multiplied by the exchange rate to turn the Canadian dollars price into American dollars. The price is always stored in American dollars, but using the **canadians** property we can set it and retrieve it in Canadian dollars.

If we want to use a different name for the new value, we can set the parameter's name

between parentheses. In the following example the parameter was called **CAD** and used instead of **newValue** to calculate the value for the **USD** property.

Listing 3-40: Using a different name for the set method's parameter

```
struct Price {
    var USD: Double
    var ratetoCAD: Double
    var ratetoUSD: Double
    var canadians: Double {
        get {
            return USD * ratetoCAD
        }
        set(CAD) {
            USD = CAD * ratetoUSD
        }
    }
}

var purchase = Price(USD: 11, ratetoCAD: 1.09, ratetoUSD: 0.91744)

purchase.canadians = 500
print("Price: \(purchase.USD)") // "Price: 458.72"
print("Price in CAD: \(purchase.canadians)") // "Price in CAD: 500.0048"
```

Type Properties and Methods

The properties and methods declared above are accessible on the instances created from the definition of the structure. The access and execution of its properties and methods are done through the instances created from the definition, not from the definition itself. But there are times when being able to execute properties and methods from the definition make sense. We could need, for example, to get some information that affects all the instances, set some values to create instances with a specific configuration, or call methods to create instances with standard values. In Swift, this is possible by declaring type properties and methods (properties and methods accessible from the type, not the instance).

Type properties and methods for structures are declared adding the **static** keyword to their definition. Once a property or method is declared with this keyword, they are only accessible from the definition itself. In the following example we include the **currencies** property to inform how many currencies the structure can handle.

Listing 3-41: Defining type properties

```
struct Price {  
    var USD: Double  
    var CAD: Double  
    static var currencies = 2  
}  
  
print(Price.currencies) // 2
```

As illustrated by the code of Listing 3-41, there is no need to create an instance to access a type property or method. After the definition, the `currencies` property is read using the name of the structure and dot notation (`Price.currencies`). If we create an instance from this definition, the only properties accessible from the instance will be `USD` and `CAD`. The `currencies` property is a type property, only accessible from the type itself. The same happens with methods.

Listing 3-42: Defining type methods

```
struct Price {  
    var USD: Double  
    var CAD: Double  
    static func reserve() -> Price {  
        let instance = Price(USD: 10.0, CAD: 11.0)  
        return instance  
    }  
}  
  
var reserveprice = Price.reserve()  
  
print("Price in USD: \(reserveprice.USD) CAD: \(reserveprice.CAD)")
```

The example of Listing 3-42 adds a type method called `reserve()` to the structure's definition. The method creates and returns an instance of the structure it belongs to with standard values. This is a common procedure and another way to create our own initializer. If we use the initializer by default, the values have to be provided every time the instance is created, but with a type method the only thing we have to do is call the method on the type to get in return an instance already configured with specific values. In our example, the values correspond to a reserved price. We call the `reserve()` method on the `Price` type, the method creates the instance with the values 10.0 and 11.0, and then the instance returned is assigned to the `reserveprice` variable. At the end, the values of both properties are printed on the screen to confirm they have the reserve prices defined by the `reserve()` method (again, the method is not accessible by the instance, only from the type).

Primitive Type Structures and Casting

Including several properties and methods inside a structure and then assigning an instance of that structure to a variable is a simple way to wrap data and functionality together in a single portable unit of code. Structures are usually used this way, as practical wrappers of code, and Swift takes advantage of this feature extensively. In fact, all the primitive data types defined in Swift are structures. The syntax `variable: Int = value`, for example, is a shortcut provided by Swift for the initializer `variable = Int(value)`. Every time we assign a new value to a primitive variable we are assigning a structure that contains that value (all the conversions are done for us in the background). The following are the initializers for some of the primitive types studied in Chapter 2.

Int(value)—This is the initializer for the `Int` data type. The **value** attribute is the value we want to assign to the instance. If no value is provided, the value 0 is assigned by default. Initializers for similar types are also available (`Int8()`, `Int16()`, `Int32()`, and `Int64()`).

UInt(value)—This is the initializer for the `UInt` data type. The **value** attribute is the value we want to assign to the instance. If no value is provided, the value 0 is assigned by default. Initializers for similar types are also available (`UInt8()`, `UInt16()`, `UInt32()`, and `UInt64()`).

Float(value)—This is the initializer for the `Float` data type. The **value** attribute is the value we want to assign to the instance. If no value is provided, the value 0.0 is assigned by default.

Double(value)—This is the initializer for the `Double` data type. The **value** attribute is the value we want to assign to the instance. If no value is provided, the value 0.0 is assigned by default.

The structures for these types were already defined for us in the Swift's standard library. All we have to do to get an instance is to call its initializer with the value we want to store.

Listing 3-43: Initializing variables with initializers

```
var mynumber = Int(25)
var myprice = Double(4.99)
```

The variables `mynumber` and `myprice`, declared in Listing 3-43, are inferred to be of type `Int` and `Double`. The initializers create a structure for each value and assign the instances to the variables. This is the same as assigning the values directly to the variable (e.g., `var`

`myprice = 4.99`). There is no advantage in the use of initializers for primitive types except when the value provided is of a different data type. The definitions of these structures include several initializers that convert the value to the corresponding type. This is usually called *casting*, and we can use it to turn a variable of one type into another when required. For example, when we divide numbers, the system converts those numbers to the right type and performs the operation, but variables are already of a specific type so they have to be explicitly converted before the operation is performed or we get an error. The process doesn't really convert the variable, it just creates a new value of the right type using an initializer.

Listing 3-44: Casting a variable

```
var number1 = 10
var number2 = 2.5
var total = Double(number1) / number2 // 4.0
```

The variables `number1` and `number2` of Listing 3-44 are of type `Int` and `Double`. To perform a division between these two variables we have to cast one of them to the type of the other, because these operations can't be performed on values of different type. Using the `Double()` initializer, we create a new value of type `Double` from the value of `number1` and perform the operation (the value 10.0 created by the initializer is divided by the value 2.5 of `number2` to get the result 4.0). The process is described as “casting the `number1` variable to a `Double`”.

These initializers are also useful when working with `String` values. Sometimes the characters of a string represent numbers that we need to process. The problem is that strings can't be processed as numbers. We can't include a string in any arithmetic operation without first converting the string into a value of a numeric data type. Fortunately, the initializers for numeric types such as `Int` and `Double` can convert a value of type `String` into a number. If the operation can't be performed, the constructor returns `nil`.

Listing 3-45: Processing numbers in strings

```
var units = “45”

if let number = Int(units) {
    let total = number + 15
    print(“The total is \$(total)”) // “The total is 60”
}
```

Just as any other structure, the structures defined for primitive types also have their own properties and methods. This includes type properties and methods. For example, the integer types include two type properties called `min` and `max` to return the minimum and

maximum values they can handle.

Listing 3-46: Checking the maximum possible value

```
var mynumber: Int8 = 120
let increment = 10

if (Int8.max - mynumber) >= increment { // (127 - 120) >= 10
    mynumber += increment
}

print(mynumber) // "120"
```

The code of Listing 3-46 takes advantage of the `max` property to make sure that incrementing the value of a variable won't produce integer overflow (the result won't be greater than the numbers the variable can handle). The code starts by defining a variable of type `Int8` to store the result of the operation and another to store the number we want to add. Then we calculate how far the current value of `mynumber` is from the maximum value admitted by an `Int8` type variable (`Int8.max - mynumber`), and compare this result with the value of `increment`. If the number of units we have left is greater or equal than the value of `increment` we know that the operation can be performed without going over the limit (in this example the operation is not performed because the addition of `120 + 10` produces a result greater than the limit of 127 admitted by the `Int8` data type).

Do It Yourself: Replace the code in your Playground's file with the code of Listing 3-46. Read the `min` and `max` properties of other integer types, such as `Int16`, `Int32`, and `Int64` to know their limitations.

String Structures

Not only are the primitive data types structures, but so are the rest of the data types defined in the Swift's standard library, including the `String` data type. As we have seen in Chapter 2, we can initialize a `String` structure by simply assigning a string (a text between quotes) to a constant or a variable. This is another shortcut. In the background, instances of the `String` structure are created from the initializer included in the structure's definition.

String(value)—This initializer creates a string from the value provided by the `value` attribute. The `String` structure defines multiple versions of this initializer to create strings from different types of values, including other strings, characters, and numbers.

Most of the time we will assign a string directly to a variable as we have done so far, but the `String()` initializer may be useful in some circumstances; for instance, when we need to convert a number into a string.

Listing 3-47: Converting a number into a string

```
var age = String(44)
var mytext = "My age is " + age // "My age is 44"
```

Once the string is created, we can manipulate it with the properties and methods provided by the `String` structure. The following are the most frequently used.

capitalizedString—This property returns the string with the words capitalized (e.g., “Hello World”).

lowercaseString—This property returns the string in lowercase letters (e.g., “hello world”).

uppercaseString—This property returns the string in uppercase letters (e.g., “HELLO WORLD”).

isEmpty—This property returns a Boolean value indicating whether the value is an empty string or not. This is the same as comparing the string against an empty string (`string == ""`).

hasPrefix(value)—This method returns a Boolean value indicating whether the string begins with the text specified by the `value` attribute or not.

hasSuffix(value)—This method returns a Boolean value indicating whether the string ends with the text specified by the `value` attribute or not.

We can also process individual characters or a range of characters with the following properties and methods.

characters—This property returns a collection of `Character` values that represent the string in the structure. It is a gateway to access the string’s characters, as we will see next.

startIndex—This property returns the index value of the first character of the string.

endIndex—This property returns the index value of one position after the last character of the string. It is useful to manipulate range of characters, as we will see later.

insert(character, atIndex: index)—This method inserts the character provided by the **character** attribute at the position determined by the **index** attribute.

removeAtIndex(index)—This method removes and returns the character at the position determined by the **index** attribute.

replaceRange(range, with: string)—This method replaces the characters in the position determined by the **range** attribute with the string provided by the **string** attribute.

removeRange(range)—This method removes the characters in the position determined by the **range** attribute.

An important characteristic of Swift’s strings is that they are composed of Unicode characters. The particularity of these types of characters is that they can occupy a different amount of space in memory, even when the characters are visually the same. Because of this, we can only interact with strings through properties and methods capable of managing this type of character. One of these properties is **characters**. This property returns the characters in the string as a collection of **Character** values that we can access as any other collection type. For instance, we can create a **for in** loop to process each character.

***Listing 3-48:** Iterating through the characters in a string*

```
var text = "Hello World"
var counter = 0

for letter in text.characters {
    if letter == "l" {
        counter++
    }
}

print("The text has \(counter) letters L") // "The text has 3 letters L"
```

The loop of Listing 3-48 reads each character in the **characters** property of the **text** variable and stores it in the **letter** constant. Inside the loop, the value of the **counter** variable is incremented every time the value of **letter** is equal to the letter L (lowercase). In this example we use the loop to count characters, but if we want to count all the characters in the string, there is a simpler way. The **characters** property contains a structure that offers access to the **count** property. This property returns the total amount of characters in the

string.

Listing 3-49: Counting the characters in the string

```
var text = "Hello World"
let total = text.characters.count // 11
print("Total characters in the string: \("\(total)")")
```

Because strings are collections of Unicode characters, it is not possible to establish the position of a character using integer values. The index of the first character could always be 0, but the index of the consecutive characters depends on the size of their predecessors. Swift solved this problem adding a new data type called **String.Index**. This is a structure used by the **String** structure to manage string indexes. The **String** structure includes all the properties and methods necessary to work with these indexes and access the characters. Two of those properties are **startIndex** and **endIndex**. From these properties we can get the **String.Index** values for the first and last characters, calculate specific positions and process the characters we want. Once we have the right index, we have to declare it between square brackets after the name of the string to get the corresponding character, as we do with arrays.

Listing 3-50: Processing the string's characters

```
var text = "Hello World"
if !text.isEmpty {
    var start = text.startIndex
    var firstCharacter = text[start]
    print("First character is \("\(firstCharacter)")") // "First character is H"
}
```

The first thing we do in the example of Listing 3-50 is to check the value of the **isEmpty** property to make sure the string is not empty and there are characters to read (notice the **!** character to invert the condition). Once we know we can proceed, we get the index of the string's first character from the **startIndex** property and read the character in that position using square brackets. If we want to access a character in a different position, we have to increment the value returned by **startIndex**. The trick is that, since **String.Index** values are not integers, we can't add a simple number to them. Instead, we have to use the methods provided by the **String.Index** structure.

advanceBy(offset)—This method increments or decrements the index value and returns a new index with the result. The **offset** attribute specifies the amount of units to increment or decrement.

distanceTo(value)—This method calculates and returns the distance between the index and the index value specified by the **value** attribute.

predecessor()—This method calculates and returns the index that precedes the current index.

successor()—This method calculates and returns the index that succeeds the current index.

The following example advances the initial index 6 positions to get a different character.

Listing 3-51: Calculating a specific index

```
var text = "Hello World"
if !text.isEmpty {
    var start = text.startIndex
    var newIndex = start.advancedBy(6)

    print("The character is \text[newIndex]") // "The character is W"
}
```

The **advanceBy()** method takes an integer to calculate the new index. The original index is increased the number of units indicated by the integer and the resulting **String.Index** value is returned. In Listing 3-51 we use this function to calculate the index value of the character ‘W’ (remember that indexes start counting from 0).

If we want to get the previous index we could specify a negative number of units for the **advanceBy()** function, but a better way to move forward and backward is to implement the **predecessor()** and **successor()** methods. The following example gets the next index after the initial index and prints the corresponding character on the screen.

Listing 3-52: Getting the next index

```
var text = "John"
let index = text.startIndex
var next = index.successor()

print("Second letter is \text[next]") // "Second letter is o"
```

To get a specific index we can use a method implemented by the structure in the **characters** property called **indexOf()**. The value returned by this method is an optional containing the **String.Index** value of the first character that matches its attribute or **nil** in case of failure. In the following example, we use this method to search for the W character and

return its index.

Listing 3-53: Finding the index of a character

```
var text = "Hello World"
var findIndex = text.characters.indexOf("W")

if let index = findIndex {
    print("\(text[index]) is at index \(index)") // "W is at index 6"
}
```

Once we have the right index, we can call some of the `String` methods to insert or remove characters. The `insert()` method, for instance, inserts a single character in the position indicated by its second attribute. In the following example we call it with the value of the `endIndex` property to add a character at the end of the string.

Listing 3-54: Inserting a character in a string

```
var text = "Hello World"
text.insert("!", atIndex: text.endIndex)

print("New string is \(text)") // "New string is Hello World!"
```

Erasing a character is also very simple to do. We just have to call the `removeAtIndex()` method with the index of the character we want to erase.

Listing 3-55: Removing a character

```
var text = "Hello World"
var findIndex = text.characters.indexOf(" ")

if let index = findIndex {
    text.removeAtIndex(index)
    print("New string is \(text)") // "New string is HelloWorld"
}
```

If we want to work with groups of characters, we have to use ranges of `String.Index` values. We studied ranges before in Chapter 2. They are made using the `...` and the `..<` operators. In the examples of that chapter we used them to create ranges of integers, but the operators can take any type of values we need, including `String.Index` values.

Listing 3-56: Getting a range of characters

```
var text = "Hello World"
var start = text.startIndex
var end = start.advancedBy(4)

let range = start...end
print("The word is \(text[range])") // "The word is Hello"
```

In the code of Listing 3-56 we use the ... operator because we want to include the last character in the result, but we can ignore that character if necessary using the ..< operator.

Listing 3-57: Half-open range

```
var text = "Hello World"
var start = text.startIndex
var findIndex = text.characters.indexOf(" ")

if let end = findIndex {
    print("The first word is \(text[start..
```

The `indexOf()` method in Listing 3-57 looks for a space character and returns its index. With this value we can now create a range from the first character to the first space character and get the string's first word. But we have to be careful because the `end` index is pointing to the space character, not to the last character of the word. If we create a range between these two values, the space character will be included at the beginning. To create a range that doesn't include this character, we have to use the `..<` operator. This operator generates a half-open range that runs from one value to another but doesn't include the value on the right.

The ... and `..<` operators are actually shortcuts for the creation of a structure called **Range**. The **Range** structure includes an initializer with the attributes `start` and `end` to create an instance of a specific range of values. For example, the initializer `Range(start: 2, end: 10)` creates a range of integers from 2 to 10. This is the same as declaring `2..<10` or `2...9`. In case we need to confirm the limits of the range, the **Range** structure also includes the properties `startIndex` and `endIndex` to return the range's initial and final values.

Now that we know how to implement ranges, we can take advantage of the `replaceRange()` and `removeRange()` methods to replace and remove multiple characters.

Listing 3-58: Working with ranges of characters

```
var text = "Hello World"
```

```

var start = text.startIndex
var findIndex = text.characters.indexOf(" ")

if let end = findIndex {
    text.replaceRange(start..

```

The application of these methods is very similar to the rest of the methods studied before. In our example, the `replaceRange()` method replaces the characters from the beginning of the string up to the character before the space character (the word “Hello”) with the string “Goodbye”, getting the sentence “Goodbye World”, and the `removeRange()` method takes this sentence and removes the characters from the space character to the end of the string (“ World”), getting the final string “Goodbye”. Notice that after applying the methods over the same string, the indexes are lost and we have to calculate them again. That is why before calling the `removeRange()` method we look at the position of the space character once more and update the `findIndex` variable.

The `String` structure also includes properties and methods to work and modify the characters in the string. In the following example we use some of these properties and methods to check if a string contains the word “World” at the end and convert the lowercase letters into uppercase letters.

Listing 3-59: Implementing String methods

```

let text = "Hello World"

if text.hasSuffix("World") {
    print(text.uppercaseString) // "HELLO WORLD"
}

```

Array Structures

Arrays are also implemented as structures, and they include all the necessary properties and methods to add, insert, or delete elements. Here is a list of the most frequently used.

count—This property returns the total number of elements in the array.

isEmpty—This property returns a Boolean value to indicate whether the array is empty or not.

first—This property returns the first element of the array or **nil** if the array is empty.

last—This property returns the last element of the array or **nil** if the array is empty.

append(value)—This method adds a new element at the end of the array. The **value** attribute is the value we want to assign to the new element.

insert(value, atIndex: index)—This method adds a new element in a specific position of the array. The **value** attribute is the value we want to assign to the new element, and **index** represents the position of the array in which we want to insert the element.

insertContentsOf(array, at: index)—This method adds the array provided by the **array** attribute to the original array. The **index** attribute is an integer value that determines the position where we want to add the array.

reverse()—This method returns a new array with the elements of the array in reverse order.

minElement()—This method compares the elements in the array and returns the smallest.

maxElement()—This method compares the elements in the array and returns the largest.

indexOf(element)—This method searches for the element specified by the **element** attribute and returns its index.

enumerate()—This method is used to iterate through the elements of the array. It returns a tuple containing the index and the value of the current element.

sortInPlace(block)—This method modifies the order of the elements in the array. The **block** attribute is a block of code (closure or function) that determines the order.

sort(block)—This method returns a new array with the elements of the original array in the order determined by the **block** attribute. The **block** attribute is a block of code (closure or function) that determines the order.

dropFirst()—This method returns a new array containing all the elements of the array except the first one.

dropLast()—This method returns a new array containing all the elements of the array except the last one.

joinWithSeparator(separator)—This method returns a string that includes all the elements of the array joined by the string specified by the **separator** attribute.

filter(block)—This method filters an array and returns another array with the rest of the values. The **block** attribute is a block of code (closure or function) that filters the elements.

removeAtIndex(index)—This method removes an element from the array in the index specified by its attribute. The rest of the elements are moved and their indexes changed to occupy the vacant position.

removeLast()—This method removes the last element of the array. It returns the value of the element deleted.

removeAll(keepCapacity: value)—This method removes all the elements in the array. The **value** attribute is a Boolean value to indicate whether the space in memory occupied by the array is going to be reserved for future elements or not.

removeRange(range)—This method removes a range of elements from the array. The **range** attribute is a range of integers corresponding to the indexes of the elements we want to remove.

replaceRange(range, with: array)—This method replaces a range of elements with the elements of the array provided by the **array** attribute. The **range** attribute is a range of integers corresponding to the indexes of the elements we want to replace.

The **count** property is probably the one we are going to use the most. It returns the number of elements in the array, so it is perfect to create very useful customized loops. In some of the examples of Chapter 2 we used the **for in** loop to iterate through an array, but these types of loops work with an automatic counter. They use a pointer to point to the current element and then advance to the next in the following cycle. Instead, we can read the **count** property and use its value to limit our own counter.

Listing 3-60: Using the `count` property to iterate through an array

```
var fruits = ["Banana", "Orange", "Orange", "Apple"]

var total = 0

for var mycounter = 0; mycounter < fruits.count; mycounter++ {
    if fruits[mycounter] == "Orange" {
        total++
    }
}

print("We have \ (total) oranges") // "We have 2 oranges"
```

In the **for** loop of Listing 3-60 the value of the **mycounter** variable goes from 0 to 3 (the number returned by the **count** property of the **fruit** array), but the statements in the loop are only executed when the value is equal to 0, 1, and 2 (when it is equal to 3 the loop is over). Using this value as the index for the array we can read each element. The **if** statement inside the loop checks every value and increments the **total** variable when it is equal to “Orange”.

A practical alternative to iterate through an array is provided by the **enumerate()** method. This method returns a tuple containing the value of the element and its index, one element per cycle.

Listing 3-61: Reading indexes and values

```
let fruits = ["Banana", "Orange", "Apple"]
var message = "My fruits:"
for (myindex, myfruit) in fruits.enumerate() {
    message += " \(" + myindex + 1 + "\)-" + myfruit + " "
}
print(message) // "My fruits: 1-Banana 2-Orange 3-Apple"
```

The **enumerate()** method was designed to work with the **for in** instruction. Each cycle returns a tuple with the index and the value of the current element. The example of Listing 3-61 uses the variables **myindex** and **myfruit** to capture these values and generate the string for **message** (since the array’s indexes start from 0, we added 1 to **myindex** to get the string “My fruits: 1-Banana 2-Orange 3-Apple”).

The methods to add and remove elements from an array are straightforward. The following example illustrates how to implement them.

Listing 3-62: Adding and removing elements

```
var fruits = ["Banana", "Orange", "Apple"]
if !fruits.isEmpty {
    fruits.removeLast() // ["Banana", "Orange"]
    fruits.insert("Pear", atIndex: 1) // ["Banana", "Pear", "Orange"]
    fruits.removeAll(keepCapacity: false) // []
    fruits.append("Peach") // ["Peach"]
    fruits.insertContentsOf(["Cherry"], at: 1) // ["Peach", "Cherry"]
}
```

The comments in Listing 3-62 show how the **fruits** array is affected every time. The only method that demands some explanation is **removeAll()**. This method deletes all the elements

in the array, but it takes an argument to indicate whether the piece of memory currently occupied by the array should still be reserved for future values or not. The only purpose of this is to help the system manage the memory. If we think the new elements we are going to add to the array are going to require the same amount of memory, we can declare this argument as **true**, but if we are not going to store the same amount of values than before, then we can set it to **false** and free some memory for the system to use.

The Basics: Every time an array is affected, its indexes are reassigned. For instance, if you remove the first element of an array of three, the index 0 is reassigned to the second element and the index 1 to the third element. The system makes sure that the indexes of every array are always consecutive and start from 0. For this reason, it is not recommended to apply these methods from inside a loop.

Besides working with individual elements, we can do it with a range of elements. Ranges work with arrays the same way they work with strings. They offer an easy way to create new arrays with some of the elements of another.

Listing 3-63: Reading a range of elements

```
var fruits = ["Banana", "Orange", "Apple", "Cherry"]  
var someFruits = fruits[0..2] // ["Banana", "Orange"]  
  
print("The new selection has \\(someFruits.count) fruits")
```

The code of Listing 3-63 gets the elements at the indexes 0 and 1 from the **fruits** array and assigns them to the new **someFruits** array. Now we have two arrays: **fruits** with 4 elements and **someFruits** with 2.

Arrays created from a range of indexes are of type **ArraySlice**. This is a special type provided by Swift to store temporary arrays that are composed by elements taken from other arrays. We can iterate through these types of arrays with a loop or read its elements as we do with normal arrays, but if we want to assign them to other array variables or use them for persistent storage, we have to cast them as **Array** types using the **Array()** initializer. The initializer takes the values of the **ArraySlice** variable and returns a normal array. The following example uses the initializer to cast the **someFruits** array created before.

Listing 3-64: Casting arrays of type ArraySlice

```
var fruits = ["Banana", "Orange", "Apple", "Cherry"]  
var someFruits = fruits[0..2] // ["Banana", "Orange"]  
var newArray = Array(someFruits)
```


The `Array` structure also offers the `removeRange()` and `replaceRange()` methods to remove and replace a range of elements.

Listing 3-65: Removing and replacing elements

```
var fruits = ["Banana", "Orange", "Apple", "Banana", "Banana"]
fruits.removeRange(1...2) // ["Banana", "Banana", "Banana"]
fruits.replaceRange(0..<3, with: ["Cherry", "Cherry", "Cherry"])

print("The new array has \$(fruits.count) fruits")
```

In Listing 3-65 we call the `removeRange()` method to remove the range of elements from index 1 to index 2, getting an array filled with the value “Banana”, and then we call the `replaceRange()` method to replace the elements from index 0 to index 2 (all the elements in the array) with another array filled with Cherries. This is just a simple example that helps us understand how the methods work, but it shows a recurrent situation in app development where sometimes we need to fill a collection with elements of the same value. When working with arrays, this is simple to achieve. The `Array` structure includes a practical initializer that takes an integer called `count` and a value called `repeatedValue` and generates an array filled with the amount of elements indicated by `count` and with the value indicated by `repeatedValue`.

Listing 3-66: Initializing an array with elements of the same value

```
var fruits = ["Banana", "Orange", "Apple", "Banana", "Banana"]

let total = fruits.count
let newArray = Array(count: total, repeatedValue: "Cherry")
fruits.replaceRange(0..<total, with: newArray)

print("The new array has \$(fruits.count) fruits")
```

The advantage of this initializer is that we don’t need to know how many elements we need during development. We can calculate the number during execution and create an array with the elements we need in every situation. In the example of Listing 3-66, we create an array with the same amount of elements as the `fruits` array and then use the `replaceRange()` method to replace every single element with a new one.

The methods to remove and replace elements of an array are not selective enough; they affect the elements in a specific index or a range of indexes without considering their values. If we want to perform a more specific job, we have to use the `filter()` method. This method takes a function or a closure as an attribute. What it does is iterate

through the array and call the function or closure with the value of each element. If the function or closure returns **true**, the element is included in the new array, or is ignored otherwise. Therefore, the function or closure has to be able to receive a value of the same data type as the array and return a Boolean value to indicate whether or not the value should be part of the new array. For the following example we decided to call the **filter()** method with a function called **myfilter()**. What the function does is to return **true** if the value received is different than the string “Grape”. As a result, the new array returned by **filter()** will contain only the elements “Apple” and “Banana”.

Listing 3-67: Filtering array elements

```
var fruits = ["Apple", "Grape", "Banana", "Grape"]

func myfilter(fruit: String) -> Bool {
    if fruit != "Grape" {
        return true
    } else {
        return false
    }
}

var filteredArray = fruits.filter(myfilter) // ["Apple", "Banana"]
```

If we prefer, instead of a function we can provide a closure as the attribute and include the whole code in just one single instruction.

Listing 3-68: Sending a closure to filter the array

```
var fruits = ["Apple", "Grape", "Banana", "Grape"]

var filteredArray = fruits.filter({ (fruit: String) -> Bool in
    if fruit != "Grape" {
        return true
    } else {
        return false
    }
})

filteredArray // ["Apple", "Banana"]
```

Since the filter was made by just one instruction that already returns **true** or **false**, we can simplify the code returning the instruction itself.

Listing 3-69: Optimizing the closure

```
var fruits = ["Apple", "Grape", "Banana", "Grape"]
```

```
var filteredArray = fruits.filter({ (fruit: String) -> Bool in  
    return fruit != "Grape"  
})
```

This makes the code concise and easy to read, but the advantage of using closures is that we can simplify the instruction even more. First, there is type inference. The closure is provided as the argument of a method that belongs to a generic structure (a structure that can handle different data types), therefore Swift can infer the data type of its parameter. But that is not all. From the **return** instruction, Swift can also infer the data type of the value to be returned by the closure. Because of this, the only thing left is the name we want to use for the parameter.

Listing 3-70: Inferring the data types

```
var fruits = ["Apple", "Grape", "Banana", "Grape"]  
var filteredArray = fruits.filter({ (fruit) in return fruit != "Grape" })
```

Since we only have one statement to generate the value returned, the closure can be simplified even more by ignoring the **return** instruction.

Listing 3-71: Simplifying closures

```
var fruits = ["Apple", "Grape", "Banana", "Grape"]  
var filteredArray = fruits.filter({ (fruit) in fruit != "Grape" })
```

IMPORTANT: Closure's syntax may be simplified to the point of providing only the statement that processes the values received. It is not the purpose of this book to explore all the possible syntaxes and tools provided by the Swift language. For more information, visit our website and follow the links for this chapter.

When it comes to sorting the elements of an array, there are three options available: **reverse()**, **sort()** and **sortInPlace()**. The **reverse()** method takes the elements of an array and returns a new array with the same elements in reversed order. The value returned by the method is stored in a structure of type **RandomAccessReverseView**. As we did before with **ArraySlice** types, we can cast these values as **Array** structures with the **Array()** initializer.

Listing 3-72: Reversing the elements of an array

```
var fruits = ["Apple", "Blueberry", "Banana"]  
var array = Array(fruits.reverse()) // ["Banana", "Blueberry", "Apple"]
```

The `sort()` method, on the other hand, is a more complex option that allows us to customize how the order of the elements is determined. The method works in a similar way as the `filter()` method studied before. It takes a function or a closure that receives the value of two elements and returns `true` if the first element should appear before the second element, or `false` otherwise. According to this value, the method establishes the order of the elements and returns a new array with the elements in that order.

Listing 3-73: Sorting the elements of an array

```
var fruits = ["Apple", "Raspberry", "Banana", "Grape"]  
var newArray = fruits.sort({ (value1, value2) in value1 > value2 })  
print(newArray[0]) // "Raspberry"
```

The closure provided to the `sort()` method in Listing 3-73 is very simple. It receives two values we called `value1` and `value2` and returns a Boolean value. If `value1` is greater than `value2`, the closure returns `true`, otherwise it returns `false`. When the `sort()` method is executed, its statements perform a loop. On each cycle they read the `fruits` array and send the value of two elements to the closure. The closure compares these two values and returns `true` or `false` accordingly. This indicates to the `sort()` method which value should appear before the other in the new array, effectively sorting the elements.

Unlike the example we programmed for the `filter()` method, this one doesn't compare the attribute against a specific value. This allows us to order arrays of any data type. For example, we can use the same closure to sort an array of integers.

Listing 3-74: Sorting an array of integers

```
var numbers = [55, 12, 32, 5, 9]  
var newArray = numbers.sort({ (value1, value2) in value1 < value2 })  
print(newArray[0]) // 5
```

If we decide to work with specific data types, we can be less generic. For example, we can process `String` values to count the characters and order the strings according to their length.

Listing 3-75: Sorting strings according to the number of characters

```
var fruits = ["Apple", "Blueberry", "Banana", "Grape"]  
var newArray = fruits.sort({ (value1, value2) in value1.characters.count < value2.characters.count })  
print(newArray) // ["Apple", "Grape", "Banana", "Blueberry"]
```

The `Array` structure also includes the `sortInPlace()` method to perform the sorting in the

same array (the elements are sorted and assigned back to the original array). The following example uses the method to order the `fruits` array in reverse alphabetical order.

Listing 3-76: Sorting an array and storing the values in the same array

```
var fruits = ["Apple", "Blueberry", "Banana", "Grape"]
fruits.sortInPlace({ (value1, value2) in value1 > value2 })
print(fruits[0]) // "Grape"
```

Collections like arrays also include two powerful methods to compare elements: `minElement()` and `maxElement()`. These methods compare the values and return the smallest or largest, respectively.

Listing 3-77: Getting the largest element

```
let ages = [32, 54, 12, 27]

if ages.count > 0 {
    if let older = ages.maxElement() {
        let digits = String(older)
        print("The maximum age is \(digits.characters.count) digits long")
    }
}
```

The code of Listing 3-77 takes the largest value from an array of integers and counts the number of digits in the value returned. Because the `maxElement()` method returns an error if the collection is empty, we have to check the size of the collection first with the `count` property. The rest of the code unwraps the value returned by the `maxElement()` method, turns it into a string, and counts its characters to print the number of digits on the screen.

Besides selecting the largest or smallest value with the `maxElement()` and `minElement()` methods, we can also fetch values from the array using the `first` and `last` properties.

Listing 3-78: Getting the first value

```
let ages = [32, 54, 12, 27]

if let firstAge = ages.first {
    print("The first person is \(firstAge) years old")
}
```

The value returned by the `first` property is an optional. That is why we use optional

binding to get the value and store it in the `firstAge` constant. The `first` and `last` functions only get the first and last value, respectively. To search for any value in the array, we have to use the `indexOf()` method. The following example looks for the number 54 in the `ages` array and prints the value of its index.

Listing 3-79: *Getting the index of a specific value*

```
let ages = [32, 54, 12, 27, 54]
```

```
if let index = ages.indexOf(54) {  
    print("The value is at the position \ \(index)") // 1  
}
```

IMPORTANT: The `indexOf()` method returns the index of the first coincidence. For this reason, this method is only useful to confirm whether or not a value is in the array, but not to find multiple coincidences.

Set Structures

Sets are also implemented as structures and, because their elements are not associated with an index, the properties and methods they offer are the only way we have to process their values. The following are the most frequently used:

count—This property returns the total number of elements in the set.

isEmpty—This property returns a Boolean value to indicate whether the set is empty or not.

contains(value)—This method returns a Boolean value to indicate whether or not there is an element in the set with the value provided by the **value** attribute.

minElement()—This method compares the elements in the set and returns the smallest.

maxElement()—This method compares the elements in the set and returns the largest.

sort(block)—This method returns a new array with the elements of the set in the order determined by the **block** attribute. The **block** attribute is a block of code (closure or function).

insert(value)—This method inserts a new element in the set with the value provided by the **value** attribute.

union(collection)—This method returns a new set created with the values of the original set plus the values provided by the **collection** attribute. The **collection** attribute may be an array or another set.

subtract(collection)—This method returns a new set created by subtracting the elements provided by the **collection** attribute to the original set. The **collection** attribute may be an array or another set.

intersect(collection)—This method returns a new set created with the values of the original set that coincides with the values provided by the **collection** attribute. The **collection** attribute may be an array or another set.

remove(value)—This method removes from the set the element with the value provided by the **value** attribute.

removeAll(keepCapacity: value)—This method removes all the elements in the set. The **value** attribute is a Boolean value to indicate whether the space in memory occupied by the set is going to be reserved for future elements or not.

Because elements in a set are not associated with an index, if we want to know whether an element exists or not, we have to search for its value. Set structures provide the **contains()** function for this purpose.

Listing 3-80: Searching for an element

```
var fruits: Set = ["Apple", "Orange", "Banana"]

if fruits.contains("Apple") {
    print("Apple exists!")
}
```

To insert a new element we just have to execute the **insert()** method.

Listing 3-81: Inserting a new element

```
var fruits: Set = ["Apple", "Orange", "Banana"]

if !fruits.contains("Grape") {
    fruits.insert("Grape")
}

print("The set has \$(fruits.count) elements") // 4
```

In listing 3-81 we use the `contains()` method again to check if an element with the “Grape” value already exists, and if not we insert the element with the `insert()` method. Checking the value is not necessary; if the value is already part of the set, the `insert()` method doesn’t do anything.

To remove an element we have to call the `remove()` method.

Listing 3-82: Removing an element

```
var fruits: Set = ["Apple", "Orange", "Banana"]

if fruits.remove("Banana") != nil {
    print("The element was removed")
}
```

The `remove()` method removes the element which value matches the value of its attribute or returns `nil` in case of failure. In the code of Listing 3-82 we compare the value returned by the method to `nil` and print a message if the element was successfully removed.

Sets are collections without order. Every time we read a set, the order in which its values are returned is not guaranteed, but we can use the `sort()` method to create a new array with the values of the set in order. The following example orders the elements of the `fruits` set in reverse alphabetical order, creating a new array we called `orderFruits`.

Listing 3-83: Sorting the elements of a set

```
var fruits: Set = ["Apple", "Orange", "Banana"]
var orderFruits = fruits.sort({ (value1, value2) in value1 > value2 })
print(orderFruits[0]) // "Orange"
```

The application of the rest of the methods available to manipulate the set is straightforward. The `union()` method returns a new set with the values of the set plus the new values, the `subtract()` method returns a new set with the values of the set less the new values, and the `intersect()` method returns a new set with the elements of the set and the new elements that share the same value.

Listing 3-84: Combining sets

```
var fruits: Set = ["Apple", "Orange", "Banana"]

var set1 = fruits.union(["Grapes"]) // "Apple", "Grapes", "Orange", "Banana"
var set2 = fruits.subtract(["Apple", "Banana"]) // "Orange"
```



```
var set3 = fruits.intersect(["Apple", "Cherry"]) // "Apple"
```

The Basics: Use a set instead of an array every time you don't need an ordered list of elements and you are worried about performance. Sets are faster than arrays and dictionaries, and that is usually the only advantage they have to offer.

Dictionary Structures

Dictionaries are implemented as structures as well. Here is a list of their most useful properties and methods.

count—This property returns the total number of elements in the dictionary.

isEmpty—This property returns a Boolean value to indicate whether the dictionary is empty or not.

keys—This property returns a collection with the keys of the elements in the dictionary. It is of type **LazyMapCollection**, a collection structure. The structure provides the **array** property to get an array from the values.

values—This property returns a collection with the values of the elements in the dictionary. It is of type **LazyMapCollection**, a collection structure. The structure provides the **array** property to get an array from the values.

enumerate()—This method is used to iterate through the elements of the dictionary. It returns a tuple containing the index and the value of the current element.

sort(block)—This method returns a new array with the elements of the dictionary in the order determined by the **block** attribute. The **block** attribute is a block of code (closure or function).

updateValue(value, forKey: key)—This method updates the value of an element with the value and key specified by its attributes. If the key doesn't exist, the method creates a new element. It returns the previous value when the key already exists or **nil** otherwise.

removeValueForKey(key)—This method removes the element with the key equal to the value of the **key** attribute. It returns an optional containing the value of the deleted element or **nil** if no element with the specified key was found.

removeAll(keepCapacity: value)—This method removes all the elements in the dictionary. The **value** attribute is a Boolean value to indicate whether the space in

memory occupied by the dictionary is going to be reserved for future elements or not.

Listing 3-85: Adding elements to a dictionary

```
var fruits = ["one": "Banana", "two": "Apple", "three": "Pear"]

fruits.updateValue("Banana", forKey: "three")
fruits.removeValueForKey("one") // ["three": "Banana", "two": "Apple"]
```

The `updateValue()` method updates the value of an element when there is already an element with that key or creates a new one if the key doesn't exist. This is the same as assigning a value directly to an element (Chapter 2, Listing 2-54 and 2-55), but the method returns the previous value, which may be useful sometimes.

As sets, dictionaries are an unordered collection of values, but we can create an array with their elements in a specific order using the `sort()` method. The method processes the values as tuples, with the element's key as the first value of the tuple and the element's value as the second value of the tuple.

Listing 3-86: Sorting the values of a dictionary

```
var fruits = ["one": "Banana", "two": "Apple", "three": "Pear"]
var list = fruits.sort({ (value1, value2) in value1.0 < value2.0 })

print("The key of the second element is \(list[2].0)") // "two"
print("The value of the second element is \(list[2].1)") // "Apple"
```

To select the order inside the block, we have to read the values of the tuple using their indexes (see Listing 2-36). The code of Listing 3-86 compares the values at index 0 to sort the `fruits` dictionary by its keys in alphabetical order (`value1.0 < value2.0`). The array returned is a collection of tuples, with every element containing the keys and values of the original dictionary (`[("one", "Banana"), ("three", "Pear"), ("two", "Apple")]`).

We have already seen how to iterate through the elements of a dictionary with a `for in` loop in Chapter 2 (see Listing 2-78). The loop gets each element and generates a tuple with the key and value. There are times when we only need the element's key or the element's value. To help us with this, the dictionary structure provides two properties, `keys` and `values`. The properties return a collection containing only the keys or the values of the elements, respectively.

Listing 3-87: Iterating through the dictionary's keys

```
var fruits = ["one": "Banana", "two": "Apple", "three": "Pear"]
```

```
for key in fruits.keys {  
    if key == "two" {  
        print("We have an element with the key 'two'")  
    }  
}
```

The collections returned by the `keys` and `values` properties are structures of a basic collection type called `LazyMapCollection`. As we did before with other basic types, to work with the values of a `LazyMapCollection` collection we have to turn it into an array.

Listing 3-88: Reading a dictionary key

```
var fruits = ["one": "Banana", "two": "Apple", "three": "Pear"]
```

```
let keys = Array(fruits.keys)  
print("The first key in the array is \"(keys[0])\"") // "one"
```

IMPORTANT: The order in which the elements of a dictionary are stored and returned is not guaranteed. The `for in` loop and the `keys` and `values` properties may return the values in a different order than the one they were declared. If you need an ordered list, use an array.

3.4 Enumerations

Enumerations behave pretty much like structures. We can define our own properties and methods inside an enumeration and they also include initializers, properties and methods by default. The most useful property is called `rawValue`, which lets us read the values associated with each `case`. As we learned in Chapter 2, the values of an enumeration may have associated values. These values are called *raw values* and are accessible from this property. Raw values are usually used as a description, as in the following example.

Listing 3-89: Reading raw values

```
enum Number: String {
    case One = "Number One"
    case Two = "Number Two"
    case Three = "Number Three"
}
var mynumber = Number.One
print("The value is \"(mynumber.rawValue)\"") // "The value is Number One"
```

Enumeration types include a member initializer to create an instance from a raw value. Instead of declaring the variable using the name of the member (`One`), as we did in Listing 3-89, we can use the initializer and the raw value. The initializer includes the `rawValue` attribute to specify the value used to create the instance.

Listing 3-90: Creating an enumeration from its initializer

```
enum Number: String {
    case One = "Number One"
    case Two = "Number Two"
    case Three = "Number Three"
}
var mynumber = Number(rawValue: "Number Two")

if mynumber == .Two {
    print("Correct Value")
}
```

In Listing 3-90 we create an instance of `Number` with the value "Number Two" and then check that the variable contains the proper member value with an `if` statement. We can compare the member value or the raw value to identify an instance of an enumeration type. It doesn't really matter what value we use as long as we know what we are looking

for.

What makes enumerations part of the programming paradigm proposed by Swift is not their capacity to store different types of values but the possibility to include custom methods and also computed properties. The following example adds a method to our **Number** structure that prints a message depending on the current value.

Listing 3-91: Adding methods to an enumeration

```
enum Number: Int {
    case One = 1
    case Two = 2
    case Three = 3
    func printMessage() -> String {
        switch self {
        case .One:
            return "We are the best"
        case .Two:
            return "We have to study more"
        case .Three:
            return "This is just the beginning"
        }
    }
}

var mynumber = Number(rawValue: 2)

if mynumber != nil {
    print(mynumber!.printMessage()) // "We have to study more"
}
```

The syntax for the methods is the same as we used before for structures. When we need to check the current value of the instance from inside the method, we have to use the **self** keyword. This keyword refers to the instance where the method is being executed (in our case, **mynumber**), and this is how we can check for the instance's current value and return the right message (we will learn more about the **self** keyword next).

IMPORTANT: Enumeration types can be turned into complex processing units, combining multiple methods, properties, and data types. It is not the purpose of this book to show you every possible alternative. For more information, visit our website and follow the links for this chapter.

3.5 Objects

Objects are data types that include data and functionality in the form of properties and methods, but unlike structures and enumerations they are stored by reference, which means that more than one constant or variable can refer to the same object. This makes objects suitable for situations where the same piece of data has to be shared by different parts of the code.

IMPORTANT: Although Swift introduces its own paradigm based on structures and protocols, most of the frameworks provided by Apple are still programmed in Objective-C with classes (the blueprints for objects) and therefore we have to work with objects and implement the Object Oriented Programming paradigm to create our apps. At this moment, Apple’s advice is to work with objects when necessary and apply structures and protocols in our own code every time we feel appropriate. We will study protocols and the Protocol Oriented Programming paradigm at the end of this chapter.

Definition of Objects

Like structures and enumerations, objects are defined first and then instances are created from their definition. The definitions of objects are called *classes*, and what are actually called objects are the instances created from those classes. An object is nothing more than an instance of a class. Classes are declared the same way as the definitions of structures or enumerations, but instead of the `struct` or `enum` keywords we have to use the `class` keyword.

Listing 3-92: Defining a class

```
class Employee {  
    var name = "Undefined"  
    var age = 0  
}
```

The code of Listing 3-92 defines a simple class called `Employee` with two properties: `name` and `age`. As always, this doesn’t create anything, it is just defining a new custom data type. To store data in memory in this format we have to declare a constant or a variable of this type and assign to it an instance of the class created by an initializer.

Listing 3-93: Creating an object from a class

```
class Employee {  
    var name = "Undefined"  
    var age = 0  
}  
  
let employee1: Employee = Employee()
```

In Listing 3-93, the `Employee()` initializer creates a new instance of the class `Employee`. Since the words instance and object are synonyms, we can say that we have created a new object called `employee1` containing two properties: `name` and `age`. To access the properties of the `employee1` object we can use dot notation.

Listing 3-94: Assigning new values to the object's properties

```
class Employee {  
    var name = "Undefined"  
    var age = 0  
}  
  
let employee1 = Employee()  
  
employee1.name = "John"  
employee1.age = 32
```

Do It Yourself: Replace the code in your Playground's file with the code of Listing 3-94. The values of the properties are shown on the Results Side Bar. Repeat the same process for the rest of the examples in this chapter.

Another way to modify the values of the properties of an object is from its methods. In contrast to structures, methods that belong to objects can modify the properties of their own object without adding anything to the definition (they don't need to be declared as `mutating`).

Listing 3-95: Modifying properties from the object's methods

```
class Employee {  
    var name = "Undefined"  
    var age = 0  
  
    func changename(newname: String, newage: Int) {  
        name = newname  
        age = newage  
    }  
}
```

```
}  
let employee1 = Employee()  
employee1.changename("Martin", newage: 32)
```

In Listing 3-95, the `changename()` method is added to the `Employee` class to modify the values of its properties. The last statement of this example calls this method in the `employee1` object to assign the values “Martin” and 32 to the `name` and `age` properties, respectively.

IMPORTANT: As well as structures, we can create all the objects we need from the same definition (class). Each object will have its own properties, values, and methods.

Property Observers

Properties are accessible from every part of the code that has access to the structures or objects they belong to. Therefore, they may be modified at any moment by different processes, such as in response to user interaction or events produced by the system. Computed properties execute a setter when a new value is assigned to them, so we have control over that process, but stored properties store the value right away. To inform the instance of a structure or a class that one of its stored properties was modified, Swift introduces Property Observers.

Property Observers are special methods (like the `init()` method studied before) that can be added to a property to execute code before and after a new value is assigned to it. The methods are called `willSet()` and `didSet()`, and are declared between braces after the properties declaration.

Listing 3-96: Adding observers to a property

```
class Employee {  
    var name = "Undefined"  
    var state = "Active"  
    var difference = 0  
    var age: Int = 0 {  
        willSet {  
            if newValue > 65 {  
                state = "Retired"  
            }  
        }  
    }  
}
```



```

    didSet {
        difference = age - oldValue
    }
}

let employee1 = Employee()
employee1.age = 66

```

The **Employee** class of Listing 3-96 contains a total of four properties (**name**, **state**, **difference**, and **age**). Property Observers were declared for the **age** property. Every time a new value is assigned to this property, the **willSet()** and **didSet()** methods are executed. The **willSet()** method is called before the new value is assigned to the property and the **didSet()** method is called after the value was assigned. Swift automatically creates a parameter called **newValue** for the **willSet()** method to provide access to the value that is going to be assigned to the property. In our example, when this method is executed, the value of **newValue** is compared to the number 65 and, if the condition is true, the value of the **state** property is changed to “Retired”. Something similar happens with the **didSet()** method. Swift automatically creates a parameter called **oldValue** to provide access to the property’s old value after the new value was assigned. We subtract this value to the current value to get the difference between the old and the new age and then assign the result to the **difference** property.

We can change the names of the methods’ parameters by declaring our own names between parentheses. In the following example, the names provided by Swift are replaced by **newage** and **previousage**, respectively.

Listing 3-97: Providing our own names for the parameters of the observers

```

class Employee {
    var name = “Undefined”
    var state = “Active”
    var difference = 0
    var age: Int = 0 {
        willSet(newage) {
            if newage > 65 {
                state = “Retired”
            }
        }
        didSet(previousage) {
            difference = age - previousage
        }
    }
}

let employee1 = Employee()

```

employee1.age = 66

IMPORTANT: Property Observers are particularly useful with objects. As we will see next, there are different programming patterns that allow us to connect objects with other objects and delegate work. Property Observers are the best way to inform an object that its properties were modified, so it can update the rest of its data and adapt to the new situation when necessary (we will see a practical example in Chapter 14, Listing 14-14).

Type Properties and Methods

We have studied type properties and methods before for structures. These are properties and methods that are accessible from the type (the class) and not from the instances created from the type. They work in classes the same way as in structures, but instead of the `static` keyword we have to use the `class` keyword.

Listing 3-98: Declaring a type method for a class

```
class Employee {  
    var name = "Undefined"  
    var age = 0  
  
    class func description() {  
        print("This class stores the name and age of an employee")  
    }  
}  
  
Employee.description()
```

The code of Listing 3-98 defines a simple `Employee` class with two properties, `name` and `age`. The type method declared next is just describing the purpose of the class. Every time the `description()` method is executed, the description of the class is printed on the screen. Again, we don't have to create an instance of the class because the method is executed on the class itself.

Optional Chaining

As well as structures, objects can contain other objects, and their values are accessible

through dot notation. Listing 3-99, next, shows an example with two classes: **Salary** and **Employee**.

Listing 3-99: Creating objects inside objects

```
class Salary {  
    var money = 0.0  
}  
  
class Employee {  
    var name = "Undefined"  
    var age = 0  
    var salary = Salary()  
}  
  
let employee1 = Employee()  
  
employee1.name = "George"  
employee1.age = 34  
employee1.salary.money = 55000
```

This is a similar example to the one programmed before to connect structures (Listing 3-28). The **Employee** class includes a property called **salary** that contains an object of the **Salary** class. When a new instance of the **Employee** class is created, the **Salary()** initializer creates an instance of the **Salary** class and assigns the object to **salary**. Dot notation is used later to concatenate the values and access the properties of each object.

In this example, the **salary** object is created during instantiation, but this is not usually the case. Sometimes the values of the properties containing objects are defined after the instance is created and therefore those properties have to be declared as optionals. The problem with optionals is that we always have to check whether the variable or property has a value before we use it. One way to do it is to compare the optional against **nil** or use Optional Binding, studied in Chapter 2, but when we have one object inside another in a long chain, and more than one component of that chain is an optional, the code gets messy. To simplify this task, Swift introduces Optional Chaining.

Optional Chaining is a simple tool to access objects, properties and methods in a hierarchical chain that contains optional components. As always, the access to these components is done through dot notation, but a question mark is added to the names of the variables and properties that have optional values. When the system finds an optional, it checks whether or not it contains a value and continues only in cases of success. Here is the same previous example, but with the **salary** property turned into an optional.

Listing 3-100: Accessing optional properties

```
class Salary {
```

```

    var money = 0.0
}
class Employee {
    var name = "Undefined"
    var age = 0
    var salary: Salary?
}
let employee1 = Employee()

employee1.name = "George"
employee1.age = 34
employee1.salary?.money = 55000 // returns nil

```

The `salary` property in the code of Listing 3-100 is an optional (its initial value is not defined). Every time we need to read this property we have to unwrap its value. Using Optional Chaining we just have to concatenate the values with dot notation and add a question mark after the name of the optional. The system reads every component in the instruction and checks their values. If any of the optionals have no value, it returns `nil`, but when all the optionals have values, the instruction performs the task (in this case, assigning the number 55000 to `money`).

Reference Types

The data types we have studied before, including structures and enumerations, are value types. This means that every time we assign a variable of any of these types to another variable, the value is copied. For example, if we create an instance of a structure and then assign this instance to another variable, the instance is actually copied and we end up with two instances of the same structure (two copies in memory). Here is an example.

Listing 3-101: Creating a copy of an instance

```

struct Employee {
    var name = "Undefined"
}
var employee1 = Employee()

var employee2 = employee1
employee2.name = "George"

print("Employee1 Name: \(employee1.name)") // "Undefined"

```

The code of Listing 3-101 defines a simple structure called `Employee` with just one property called `name`. After the definition, a new instance is created with the `Employee()` initializer and assigned to the `employee1` variable. Next, the `employee2` variable is declared and the `employee1` variable is assigned to it. At this moment the system makes a copy of the structure contained by the `employee1` variable and assigns that copy to `employee2`. Both variables, `employee1` and `employee2`, have a copy of the same structure. Therefore, if we assign the value “George” to the property of the instance in the `employee2` variable, the value of the property in `employee1` will still be “Undefined”. The following figure illustrates this process.

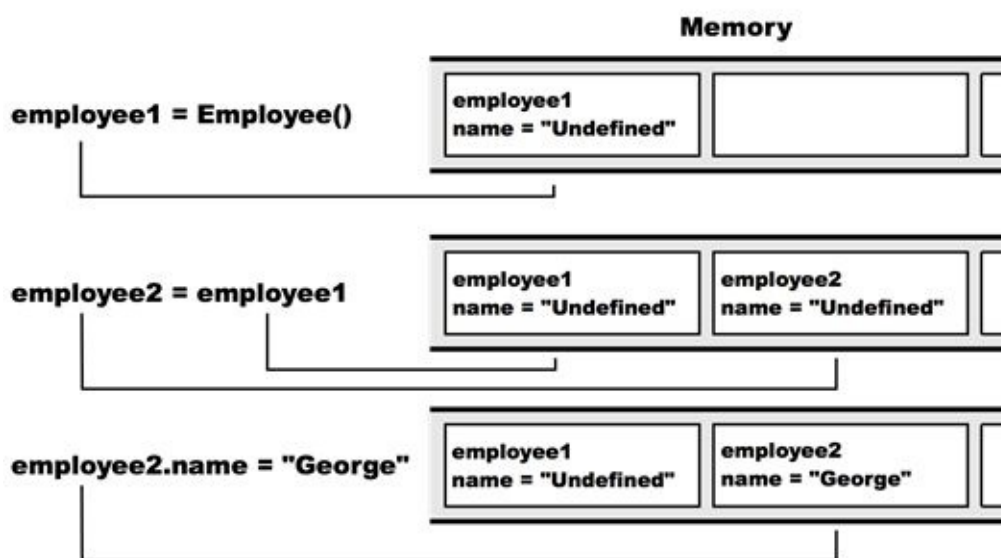


Figure 3-1: Value types stored in memory

Figure 3-1 shows how two different copies of the `Employee` structure, one referenced by the `employee1` variable and the other referenced by the `employee2` variable, are stored in memory. Any modification to the values of one of the instances won’t affect the other, because they are copies that occupy a different space in memory.

Objects, on the other hand, are passed by reference. This means when we assign an existing object to a constant or a variable, a reference to the object is assigned to it, not a copy of the object. Here is the same previous example, this time applied to objects (notice that we use the `class` keyword to define objects instead of structures).

Listing 3-102: *Creating a new reference to an object*

```
class Employee {
  var name = "Undefined"
}

let employee1 = Employee()
```

```
let employee2 = employee1
employee2.name = "George"
print("Employee1 Name: \(employee1.name)") // "George"
```

The object in `employee2` is exactly the same as the object in `employee1`. Any change in the `name` property through one of the constants is reflected in the other, because both point to the same object in memory (they refer to the same instance). Figure 3-2 next illustrates this process.

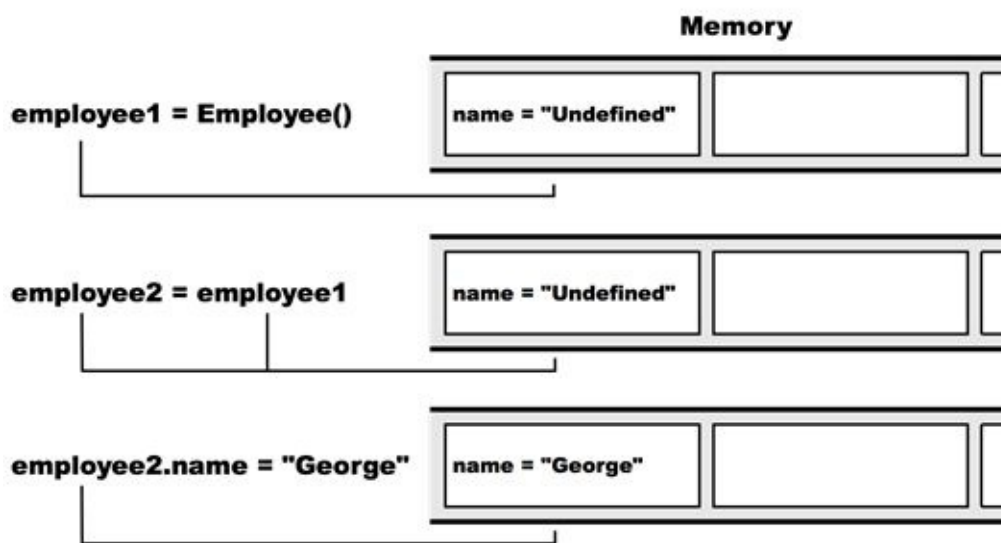


Figure 3-2: Objects stored in memory

Constants or variables that were assigned an object don't store the object; they actually store the value of the memory address where the object is located. When a constant or variable containing this address is assigned to another constant or variable, only the address is copied, and this is why the object is not duplicated. This is the most important characteristic of objects, and what makes them suitable for situations in which data in memory has to be accessed and shared by different parts of the code. A common example is when we need to define a data type to control the device's screen. The screen is only one, so we shouldn't generate multiple copies of the data that controls it. Instead, we create an object and pass the reference of that object to the rest of the code. Each part of the code accesses exactly the same object and works with the same properties and methods. Every change on its properties is always reflected on the screen.

IMPORTANT: Because constants and variables store a reference to an object (a memory address), when you want to compare the objects they reference you can't use the `==` operator. Instead, you have to use the operators `===` (identical to) and `!==` (not identical to) provided by Swift (otherwise you would be comparing references, not

objects).

The Basics: When you modify the properties of an object, its properties are modified but not its reference. This is why objects are usually stored in constants and only their properties are defined as variables. Unless we plan to replace one object by another later, it is always recommended to store objects in constants.

Using references is one of the characteristics that make objects unique and useful, but objects are always passed by reference, even when they are sent to a function or method. Sometimes this is not ideal. While developing our application we will find there are times when several copies of the same object are required to produce what we want (e.g., to create different buttons on the screen). In cases like this, we can create new instances from the same class assigning the initializer to multiple constants or variables.

Listing 3-103: Creating several objects from the same class

```
class Employee {  
    var name = "Undefined"  
}  
  
let employee1 = Employee()  
let employee2 = Employee()  
  
employee2.name = "George"  
print("Employee1 Name: \(employee1.name)") // "Undefined"
```

Two different objects are created by the code of Listing 3-103. Both objects, **employee1** and **employee2**, are independent instances and occupy a different space in memory. Assigning a new value to the **name** property in the **employee2** object doesn't affect the value of the same property in the **employee1** object (the **name** property on this object still has the value "Undefined").

Because the same object may be referenced by multiple constants or variables, every language that works with objects offers a way for the object to reference itself. In Swift this is done automatically, but there are situations in which this reference has to be declared explicitly. For this purpose, Swift defines a special keyword called **self**. The **self** keyword is just saying "this object" and it is declared before the name of the property or the method we want to access.

The most common situations in which the use of this keyword is required is when we need to declare the names of a method's parameters equal to the names of the object's properties. If the names are the same, the system doesn't know whether we are referring to the property or the parameter. The **self** keyword clarifies the situation.

Listing 3-104: Referring to the object with `self`

```
class Employee {  
    var name = "Undefined"  
  
    func changename(name: String) {  
        self.name = name  
    }  
}  
  
let employee1 = Employee()  
employee1.changename("Martin")  
  
print("Name: \(employee1.name)") // "Name: Martin"
```

The `self` keyword in the `changename()` method of Listing 3-104 represents the object created from the `Employee` class and helps the system understand what we are referring to when we use the word `name`. When we call the `changename()` method in the `employee1` object, the parameter receives the string "Martin" and the statement inside the method assigns the value of this parameter to the `name` property (`self.name`).

Memory Management

Because objects are stored by reference, they can be referenced by several variables or properties at the same time. If a variable or property is erased, the object it references can't be erased from memory because another variable or property could still be using it. This creates a situation where we can end up with objects that are not required anymore, filling up the device's memory. One approach to solve this problem was to provide the tools for programmers to erase the objects themselves. Having the proper tools was one step forward, but most programmers struggled to find the right place in the code to erase the object, figure out when the object was no longer being referenced by any variable or property, or remember to do it, delivering apps that after extended use would fill the memory and crash. The final solution was to create an automatic system that counts the number of variables or properties referencing an object and only removes the object from memory when all the references were erased (all the variables and properties were erased, set to `nil`, or they were assigned a reference to another object). The system was called ARC (Automatic Reference Counting) and it was immediately integrated into Objective-C and now Swift. Thanks to ARC, we don't have to worry about the memory anymore; the objects are automatically erased when there is no longer a variable or property containing a reference to that space in memory.

In an ideal scenario, this system works like magic, counting how many references we

create to the same object and erasing that object when none of those references exist anymore. But there are situations in which we can create something called a *Strong Reference Cycle*. This happens when two objects have a property that references the other object.

Listing 3-105: *Creating objects with strong references*

```
class Employee {  
    var name: String?  
    var location: Department?  
}  
  
class Department {  
    var area: String?  
    var person: Employee?  
}  
  
var employee = Employee()  
var department = Department()
```

The code of Listing 3-105 defines two classes: **Employee** and **Department**. Both classes contain a property that references an object of the other class. After the definition, objects of each class are created and stored in the **employee** and **department** variables. In this example we didn't assign any value to their properties, therefore the only references we have created are the references stored in the variables, as illustrated in the following figure.

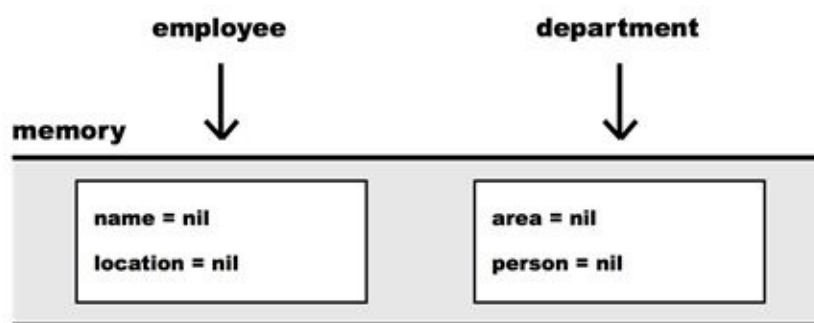


Figure 3-3: *Objects referenced only by variables*

In this case, if the variables are erased or their values are replaced, the corresponding objects are erased as well because they are only referenced by the **employee** and **department** variables. But everything changes when we reference the objects from their own properties.

Listing 3-106: *Referencing one object from another*

```

class Employee {
    var name: String?
    var location: Department?
}

class Department {
    var area: String?
    var person: Employee?
}

var employee = Employee()
var department = Department()

employee.name = "John"
employee.location = department

department.area = "Mail"
department.person = employee

```

In the code of Listing 3-106, new values are assigned to the objects' properties. The reference in the **department** variable is assigned to the **location** property of the **employee** object, and the reference in the **employee** variable is assigned to the **person** property of the **department** object. Now each object contains a reference to the other. The result is illustrated in Figure 3-4.

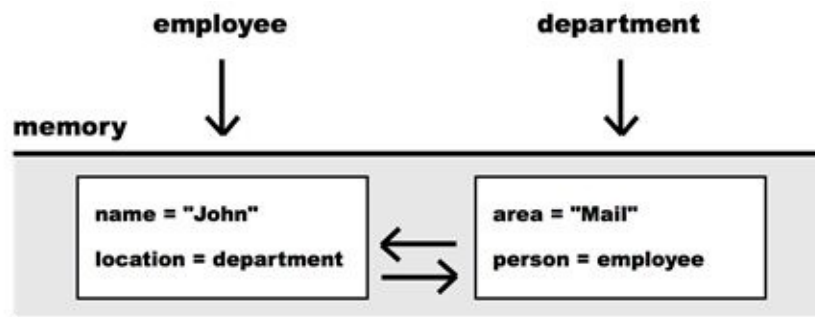


Figure 3-4: Objects referencing each other

At this point, each object is referenced by a variable and also a property. The object of the **Employee** class is referenced by the **employee** variable and the **person** property, and the object of the **Department** class is referenced by the **department** variable and the **location** property. If for some reason we don't need to access the objects from our code anymore and erase or replace the values of the **employee** and **department** variables, ARC won't erase the objects from memory because their properties still have a reference that keeps them alive. The object of the **Employee** class is not erased because it is referenced by the **person** property of the object created from the **Department** class, and this object is not erased because of the reference in the **location** property of the first object. This is illustrated in

Figure 3-5.

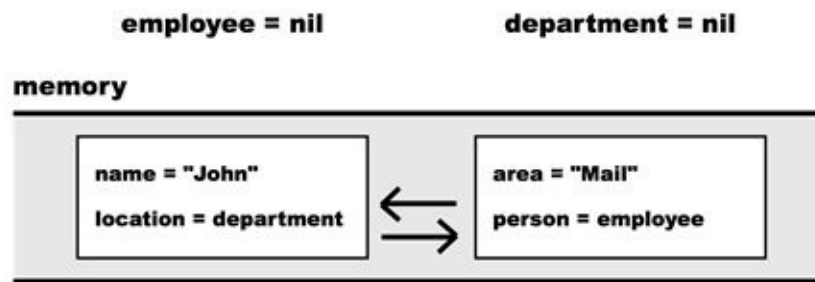


Figure 3-5: Objects preserved in memory

This example assigns `nil` to the `employee` and `department` variables, and in consequence the old objects are not accessible from our code anymore but they are preserved in memory because ARC has no way to know that they have become useless to our program.

Swift has solved this problem classifying the references into three categories: strong, weak, and unowned. Normal references are considered to be strong; they are always valid and the objects they point to are preserved in memory as long as they exist. These are the kind of references we have been using so far, and that is why the cycle created by our example is called Strong Reference Cycle. The solution to break this cycle is to define one of the references as `weak` or `unowned`. When ARC encounters one of these types of references to be the last reference to an object, the object is erased from memory as if the reference had never existed.

Listing 3-107: Assigning weak references

```
class Employee {
    var name: String?
    var location: Department?
}

class Department {
    var area: String?
    weak var person: Employee?
}

var employee = Employee()
var department = Department()

employee.name = "John"
employee.location = department

department.area = "Mail"
department.person = employee
```

In the code of Listing 3-107, the `person` property was declared as `weak`. Now, when the references from the variables are erased, the object created from the `Employee` class is erased from memory because the only reference it has left is the weak reference from the `person` property. After this object disappears, the object created from the `Department` class doesn't have any other strong reference either, so it is also erased from memory.

What the weak reference does is to reference an object without affecting ARC's counting. Weak references are not considered by ARC to determine whether an object should be erased or not. The unowned reference works the same way, but it differs with the weak reference on the type of values it applies to. Weak references apply to variables and properties with optional values (they can be empty at some point) and unowned references apply to non-optional values (they always have a value).

IMPORTANT: The use of weak and unowned references is not frequent, except for the construction of the user's interface, as we will see in Chapter 5. There are more complex situations in which these types of references become useful, but the topic is beyond the scope of this book. For more information visit our website and follow the links for this chapter.

Inheritance

One of the main purposes of structures and objects is to define pieces of code that can be copied and passed around. The code is defined once and then instances (copies) of that code are created every time they are required. This programming pattern works well when we define our own code, but presents some limitations when working with code programmed by other developers and shared through libraries and frameworks. The programmers creating the code for us can't anticipate how we are going to use it and all the possible variations required for every specific app. To provide a solution to this problem, classes incorporate inheritance. A class can inherit properties and methods from another class and then incorporate improvements adding some properties and methods of its own. This way, programmers can share their classes and developers can adapt them to their own needs, turning classes, and therefore objects, into the most widely adopted programming paradigm.

To illustrate how inheritance works, the following examples present a situation in which a class has to be expanded to contain additional information that was not initially contemplated.

Listing 3-108: Defining a basic class

```
class Employee {  
    var name = "Undefined"
```

```

var age = 0
func createbadge() -> String {
    return "Employee \ (name) \ (age)"
}
}

```

The **Employee** class declared in Listing 3-108 is a normal class, similar to those we have defined before. It has two properties and a method called **createbadge()** that returns a string containing the values of those properties. This class would be enough to create objects that generate the string of text necessary to print a badge for every employee showing its name and age. But for the sake of argument, let's say that some of the employees require a badge that also displays the department they work in. One of the options is to define another class with the same properties and methods and add what we need, but this produces redundant code and it is not possible when the class was taken from a library (we usually don't have access to the definition of the classes provided by other developers or they are too complex to modify or duplicate). The solution is to create a new class that inherits the characteristics of the basic class and adds its own properties and methods to satisfy the new requirements. To indicate that a class inherits from another class, we have to write the name of the basic class after the name of the new class separated by a colon.

Listing 3-109: Inheriting properties and methods from another class

```

class Employee {
    var name = "Undefined"
    var age = 0
    func createbadge() -> String {
        return "Employee \ (name) \ (age)"
    }
}

class OfficeEmployee: Employee {
    var department = "Undefined"
}

```

The **OfficeEmployee** class added to our code in Listing 3-109 only has one property called **department**, but inherits the **name** and **age** properties, and also the **createbadge()** method from the **Employee** class. All these properties and methods are available in any of the objects created from the **OfficeEmployee** class.

Listing 3-110: Creating objects from a subclass

```

class Employee {
    var name = "Undefined"
}

```

```

var age = 0
func createbadge() -> String {
    return "Employee \$(name) \$(age)"
}
}
class OfficeEmployee: Employee {
    var department = "Undefined"
}
let employee = OfficeEmployee()
employee.name = "George"
employee.age = 25
employee.department = "Mail"

var badge = employee.createbadge()
print("Badge: \$(badge)") // "Badge: Employee George 25"

```

A class like **Employee** is called *superclass*, and a class that inherits from another class is called *subclass*. In Listing 3-110 we define the **OfficeEmployee** subclass that inherits the properties and methods of its superclass **Employee**. The statements after the definitions create a new object called **employee** from the **OfficeEmployee** subclass, assign new values to its properties (**name**, **age**, and **department**), and call the **createbadge()** method to generate the badge.

Objects created from a class have access to the properties and methods of their class and all the classes that the class inherits from (a class can inherit from another class that already inherited from another class in an infinite chain). When a property is accessed or a method is called, the system looks for it on the object's class and, if it is not there, it keeps looking in the superclasses up the hierarchical chain until it finds it.

IMPORTANT: Inheritance doesn't work the other way around. For example, considering the code of Listing 3-110, objects created from the class **OfficeEmployee** have access to the **department** property of this class and also the properties and methods of the **Employee** class, but objects created from the **Employee** class don't have access to the **department** property.

Because of this hierarchical chain, sometimes a method doesn't have access to all the properties available for the object. For example, the **createbadge()** method called on the **employee** object created in Listing 3-110 have access to the properties declared on the **Employee** class but not to those declared in the **OfficeEmployee** class. If we want the method to also print the value of the **department** property, we have to declare it again in the **OfficeEmployee** class with the appropriate modifications. To overwrite a method of a superclass, we have to prefix it with the **override** keyword.

Listing 3-111: Overwriting an inherited method

```
class Employee {
  var name = "Undefined"
  var age = 0
  func createbadge() -> String {
    return "Employee \ (name) \ (age)"
  }
}

class OfficeEmployee: Employee {
  var department = "Undefined"
  override func createbadge() -> String {
    return "Employee \ (department) \ (name) \ (age)"
  }
}

let employee = OfficeEmployee()
employee.name = "George"
employee.age = 25
employee.department = "Mail"

var badge = employee.createbadge()
print("Badge: \ (badge)") // "Badge: Employee Mail George 25"
```

The new **OfficeEmployee** subclass of Listing 3-111 overwrites the **createbadge()** method of its superclass to generate a string that includes the value of the **department** property. Now, when the method is executed from an object of this class, the method called is the one declared in **OfficeEmployee** (the old method from the superclass is ignored), and the badge generated includes the values of the three properties.

Thanks to inheritance we have created two classes without modifying previous classes or duplicating any code. The **Employee** class can create objects to store the name and age of an employee and generate a badge with this information, and the **OfficeEmployee** class can create objects to store the name, age, and also the department of the employee and generate a more complete badge with the values of all these properties.

As we explained before, when we call the **createbadge()** method on the **employee** object created from the **OfficeEmployee** class in Listing 3-111, the method executed is the one defined in the **OfficeEmployee** class. If we want to execute the method on the superclass instead, we have to use a special keyword called **super**.

The **super** keyword is similar to the **self** keyword, but instead of referring to the object, **super** refers to the superclass. It is often used when we have overwritten a method but we still need to run the method of the superclass. Here is an example.

Listing 3-112: Calling a method from the superclass

```
class Employee {
    var name = "Undefined"
    var age = 0
    func createbadge() -> String {
        return "Employee \ (name) \ (age)"
    }
}

class OfficeEmployee: Employee {
    var department = "Undefined"
    override func createbadge() -> String {
        let oldbadge = super.createbadge()
        return "\ (oldbadge) \ (department)"
    }
}

let employee = OfficeEmployee()
employee.name = "George"
employee.age = 25
employee.department = "Mail"

var badge = employee.createbadge()
print("Badge: \ (badge)") // "Badge: Employee George 25 Mail"
```

This is the same as the previous example, but now, when the `createbadge()` method of an object created from the `OfficeEmployee` class is called, the method calls the `createbadge()` method of the superclass first and assigns the result to the `oldbadge` constant. The value of this constant is later added to the value of the `department` property to generate the final string.

Type Casting

Inheritance not only transfers functionality from one class to another but also connects the classes together. The superclasses and their subclasses are linked together in a hierarchical chain. Because of this, whenever we declare a variable of the type of the superclass, objects of the subclasses can be assigned to that variable too. This is a very important feature that allows us to do things like creating arrays of objects that are of different classes but belong to the same class hierarchy.

Listing 3-113: Creating an array of objects from different subclasses

```
class Employee {  
    var name = "Undefined"  
    var age = 0  
}  
  
class OfficeEmployee: Employee {  
    var deskNumber = 0  
}  
  
class WarehouseEmployee: Employee {  
    var area = "Undefined"  
}  
  
var list: [Employee] = [OfficeEmployee(), WarehouseEmployee()]
```

The code of Listing 3-113 defines a basic class called **Employee** and then two subclasses that inherit from **Employee** called **OfficeEmployee** and **WarehouseEmployee**. The purpose is to have the common information for every employee in one class and then have classes for specific types of employee. This allows us to create objects that only carry the information they need. For instance, an employee working at the warehouse don't need the **deskNumber** property, because he doesn't work on a desk, and an employee working at the office doesn't need the **area** property because he doesn't work on a specific area, he works on his desk. Following this organization, we can create objects that only contain the **name**, **age**, and **deskNumber** properties to represent employees working at the office and objects that only contain the **name**, **age**, and **area** properties to represent employees working at the warehouse.

No matter the differences between one object and another, they all represent employees of the same company, so sooner or later we will have to include them in the same list. The class hierarchy allows us to do that. We can declare a collection, such as an array, of the data type of the superclass and then store objects of the subclasses in it. In Listing 3-113 we did exactly that with the **list** array. The array contains only two elements but there are no limitations on the amount of objects we can store as long as the objects belong to the same class hierarchy of the array's data type.

This is all good until we try to read the array. The elements of the array are all considered to be of type **Employee**, so we can't access all of their properties, only the ones defined in the **Employee** class. Also, there is no way to know what type of object each element is. We could have an **OfficeEmployee** object at index 0 and later replace it by a **WarehouseEmployee** object. The indexes and the array itself don't provide any clue to identify the objects. Swift solves both problems with the **is** and **as** operators.

is—This operator returns a Boolean value indicating whether or not the value is of a certain type.

as—This operator converts a value of one class to another class when possible.

Identifying an object is easy with the `is` operator. This operator returns a Boolean value that we can use in an `if` statement to check the object's class.

Listing 3-114: Identifying the object's data type

```
class Employee {
    var name = "Undefined"
    var age = 0
}
class OfficeEmployee: Employee {
    var deskNumber = 0
}
class WarehouseEmployee: Employee {
    var area = "Undefined"
}
var list = [OfficeEmployee(), WarehouseEmployee(), OfficeEmployee()]

var countOffice = 0
var countWarehouse = 0

for obj in list {
    if obj is OfficeEmployee {
        countOffice++
    } else if obj is WarehouseEmployee {
        countWarehouse++
    }
}

print("We have \$(countOffice) employees working at the office") // 2
print("We have \$(countWarehouse) employees working at the warehouse") // 1
```

In Listing 3-114, we create the `list` array again with objects from the same classes defined in the previous example, but this time we add a `for in` loop to iterate through the array and count how many objects of each class we found. The `if` statement inside the loop uses the `is` operator to check if the current object stored in the `obj` constant is of type `OfficeEmployee` or `WarehouseEmployee` and increments the counter, respectively (`countOffice` or `countWarehouse`). At the end, the values of the counters are printed on the screen to show how many objects of each class are in the array.

Counting objects is not really what these operators are all about. The idea is to figure out the type with the `is` operator and then convert the object with the `as` operator to be able to access the right properties and methods. The `as` operator converts a value of one type to another. The conversions are not always guaranteed, and that is why this operator comes

in two forms: **as!** and **as?**. These versions of the **as** operator work in similar way as the optionals. The **as!** operator forces the conversion and returns an error if the conversion is not possible, and the **as?** operator tries to convert the object and returns an optional with the new object or **nil** in case of failure.

Listing 3-115: Casting an object

```
class Employee {
  var name = "Undefined"
  var age = 0
}
class OfficeEmployee: Employee {
  var deskNumber = 0
}
class WarehouseEmployee: Employee {
  var area = "Undefined"
}
var list = [OfficeEmployee(), WarehouseEmployee(), OfficeEmployee()]

for obj in list {
  if obj is OfficeEmployee {
    let temp = obj as! OfficeEmployee
    temp.deskNumber = 100
  } else if obj is WarehouseEmployee {
    let temp = obj as! WarehouseEmployee
    temp.area = "New Area"
  }
}
```

When we use the **as!** operator we are forcing the conversion, so we have to make sure that the conversion is possible or otherwise the app will crash (this is the same that happens when we unwrap optionals with the exclamation mark). In the code of Listing 3-115, we only use this operator after we have already checked with the **is** operator that the object is of the right class. Once the object is casted (converted) into its real type, we can access its properties and methods. In this example, the objects returned by the **as!** operator are stored in the **temp** constant and then new values are assigned to the **deskNumber** and **area** properties, respectively.

Checking for the type before converting is a little bit redundant. To simplify this code we can use the **as?** operator. Instead of forcing the conversion and crashing our app in case of failure, this particular version of the **as** operator tries to perform the conversion and returns an optional reflecting the result of the operation.

Listing 3-116: Casting an object with the `as?` operator

```
class Employee {
    var name = "Undefined"
    var age = 0
}

class OfficeEmployee: Employee {
    var deskNumber = 0
}

class WarehouseEmployee: Employee {
    var area = "Undefined"
}

var list = [OfficeEmployee(), WarehouseEmployee(), OfficeEmployee()]

for obj in list {
    if let temp = obj as? OfficeEmployee {
        temp.deskNumber = 100
    } else if let temp = obj as? WarehouseEmployee {
        temp.area = "New Area"
    }
}
```

In this new example, we use optional binding to cast the object and assign the result to the **temp** constant. First we try to cast **obj** as an **OfficeEmployee** object. If we are successful, we assign the value 100 to the **deskNumber** property, but if the value returned is **nil**, then we try to cast the object to the **WarehouseEmployee** class to modify its **area** property.

Casting can also be performed on the fly if we are sure that the conversion is possible. The statement to cast the object is the same, but we have to declare it between parentheses.

Listing 3-117: Casting an object on the fly

```
class Employee {
    var name = "Undefined"
    var age = 0
}

class OfficeEmployee: Employee {
    var deskNumber = 0
}

class WarehouseEmployee: Employee {
    var area = "Undefined"
}
```

```
}  
var list = [OfficeEmployee(), WarehouseEmployee(), OfficeEmployee()]  
  
let area = (list[1] as! WarehouseEmployee).area  
print("The area of employee 1 is \ (area)") // "Undefined"
```

Here we don't assign the object to any variable; we just cast the element of the `list` array at index 1 as a `WarehouseEmployee` object inside the parentheses and then access its `area` property. It is the value of this property that is stored in the `area` constant and then printed on the screen. Remember that conversions performed with the `as!` operator are only possible when we are sure it is going to be successful (in this case we have to know for sure that the element of the array at index 1 is a `WarehouseEmployee` object).

IMPORTANT: The `as!` operator is applied when the conversion is guaranteed to be successful, and the `as?` operator is used when we are not sure about the result. But it is also possible to use the basic `as` operator when the Swift compiler can verify that the conversion will be successful. There are two circumstances when this is possible: when we are converting an object of a subclass into its superclass and when we are casting some primitive data types. We will see an example in Chapter 4, when we will study how to convert `String` values into `NSString` objects.

AnyObject

The `as` operator works on objects that belong to the same class hierarchy. Because sometimes the objects that require casting are not related to each other, Swift defines a special class called `AnyObject` that can represent objects of any type. The objects of this type are related to others, as if they would belong to the same class hierarchy. Thanks to this generic type, we can now create collections with objects of classes that are not associated with each other.

Listing 3-118: Working with objects of `AnyObject` type

```
class Employee {  
    var name = "Undefined"  
}  
  
class Department {  
    var area = "Undefined"  
}  
  
var list: [AnyObject] = [Employee(), Department(), Department()]
```

```

for obj in list {
    if let temp = obj as? Employee {
        temp.name = ""
    } else if let temp = obj as? Department {
        temp.area = ""
    }
}

```

The `list` array declared in Listing 3-118 is of type `AnyObject` and therefore it can contain objects of any type (any class). To populate the array, we created two simple and independent classes: `Employee` and `Department`. A few objects are created from these classes and included into the array. The objects are later casted by the `as?` operator inside a `for in` loop and their corresponding properties are modified following the same procedure used in previous examples.

IMPORTANT: Swift also offers a generic type called `Any`. While the `AnyObject` type was designed to represent only classes, the `Any` type has the purpose to represent instances of any type.

Initialization

We have been initializing the properties during definition in every class declared so far. This is because classes don't provide member initializers as structures do. The properties of a class have to be initialized explicitly in the definition or during instantiation by the `init()` method. This is the same method previously introduced for structures and we can use it in our classes to create our own initializers.

Listing 3-119: Declaring a Designated_INITIALIZER

```

class Employee {
    var name: String
    var age: Int
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }
}

let employee1 = Employee(name: "George", age: 28)

```

The `init()` method declared for the `Employee` class in Listing 3-119 initializes every property of the class with the values specified in the `Employee()` initializer. This type of initializer is called *Designated Initializer* and its purpose is to prepare the properties for the object that has been created. When we declare a Designated Initializer we have to make sure all the properties are initialized.

If we know that in some circumstances our code won't be able to provide all the values during initialization, we can also declare a Convenience Initializer. A Convenience Initializer is an initializer that offers a convenient way to initialize an object with default values for some or all of its properties. It is declared as an `init()` method but preceded by the `convenience` keyword. The Convenience Initializer always calls the Designated initializer of the same class with the corresponding values.

Listing 3-120: Declaring a Convenience Initializer

```
class Employee {  
    var name: String  
    var age: Int  
    init(name: String, age: Int) {  
        self.name = name  
        self.age = age  
    }  
    convenience init() {  
        self.init(name: "Undefined", age: 0)  
    }  
}  
  
let employee1 = Employee()
```

When we create an instance of `Employee`, the system detects the number and type of attributes provided and executes the corresponding initializer. For example, if we provide the values for the `name` and the `age` parameters in the code of Listing 3-120, the system executes the Designated Initializer, because this is the initializer that contains the necessary parameters to receive those values. But if the call doesn't include any attribute, as in this example, the Convenience Initializer is executed and then the Designated Initializer is called with values by default ("Undefined" for the `name` parameter and 0 for the `age` parameter).

This process is similar to what we have done with structures. The difference is inheritance. Classes can inherit properties and methods from other classes, and this includes the `init()` method. When a subclass doesn't provide its own Designated Initializer, the initializer of its superclass is executed.

Listing 3-121: Inheriting the Designated Initializer

```

class Employee {
  var name: String
  var age: Int
  init(name: String, age: Int) {
    self.name = name
    self.age = age
  }
}

class OfficeEmployee: Employee {
  var department: String = "Undefined"
}

let employee1 = OfficeEmployee(name: "George", age: 29)

```

The code of Listing 3-121 introduces the subclass **OfficeEmployee** that inherits from the **Employee** class. The **OfficeEmployee** class doesn't provide any initializer, so the only initializer available is the one provided by its superclass. This initializer only initializes the properties **name** and **age** of the superclass. The **department** property of **OfficeEmployee** is explicitly initialized with the value "Undefined" (every object created from this class has this value by default). To provide an initializer that also includes this property, we have to declare a new Designated Initializer in the **OfficeEmployee** class.

Listing 3-122: Declaring a Designated Initializer for the subclass

```

class Employee {
  var name: String
  var age: Int
  init(name: String, age: Int) {
    self.name = name
    self.age = age
  }
}

class OfficeEmployee: Employee {
  var department: String
  init(name: String, age: Int, department: String) {
    self.department = department
    super.init(name: name, age: age)
  }
}

let employee1 = OfficeEmployee(name: "John", age: 24, department: "Mail")

```

The Designated Initializer of a subclass has to initialize the properties of its own class

first and then call the initializer of its superclass. This is done by calling the `init()` method on `super`. The `super` keyword refers to the superclass, so when the system executes the `super.init()` statement in the code of Listing 3-122, the `init()` method of the superclass is executed and the `name` and `age` properties of this class are initialized.

IMPORTANT: If you need to declare the same initializer of the superclass in the subclass (e.g., `init()`) you have to overwrite the initializer in the subclass with the `override` keyword, as in `override init()`, and call the initializer of the superclass with `super` (e.g., `super.init()`).

The Basics: There are different ways to combine Designated and Convenience initializers. The possibility of classes to inherit from other classes in an unlimited chain can turn initialization into a very complex process. This book doesn't explore all the possibilities provided by Swift in this respect. For more information, visit our website and follow the links for this chapter.

Deinitialization

There is a counterpart of the initialization process called *Deinitialization*. Despite its name, this process is not directly related to the initialization process but rather with the ARC system. ARC, as we studied previously in this chapter, is an automatic system adopted by Swift to manage memory. Letting the system manage the memory and take care of removing the objects our program doesn't need anymore presents a huge advantage, but it also means that we don't always know when an object is going to be removed. There are times when an object is using resources that have to be closed, information that has to be stored, or we just have to report to other objects that the object is not available anymore. Whatever the task, Swift offers the `deinit` method to execute any last-minute instructions we need before the object is erased from memory.

Listing 3-123: Declaring a deinitializer

```
class Item {  
    var quantity = 0.0  
    var name = "Not defined"  
    var price = 0.0  
    deinit {  
        print("this instance was erased")  
    }  
}
```

```
var purchase: Item? = Item()
```

```
purchase = nil
```

The code of Listing 3-123 defines a simple class with a deinitializer. The object is created and assigned to an optional variable. Right after that the **nil** value is assigned to the same variable to erase the reference and test the **deinit** method. In this example we just print a message on the console. Deinitialization is usually required for complex objects in charge of managing files or other resources, but the declaration of the **deinit** method always remains the same.

3.6 Protocols

The two main characteristics of classes and objects are the capacity to encapsulate data and functionality and the possibility to share and improve code through inheritance. This introduced a huge advantage over previous paradigms and turned Object Oriented Programming into the industry standard. But that changed with the introduction of structures in Swift. Swift's structures not only can encapsulate data and functionality, just like objects, but thanks to the adoption of protocols they can also inherit code. Figure 3-6 illustrates the differences between these two paradigms.

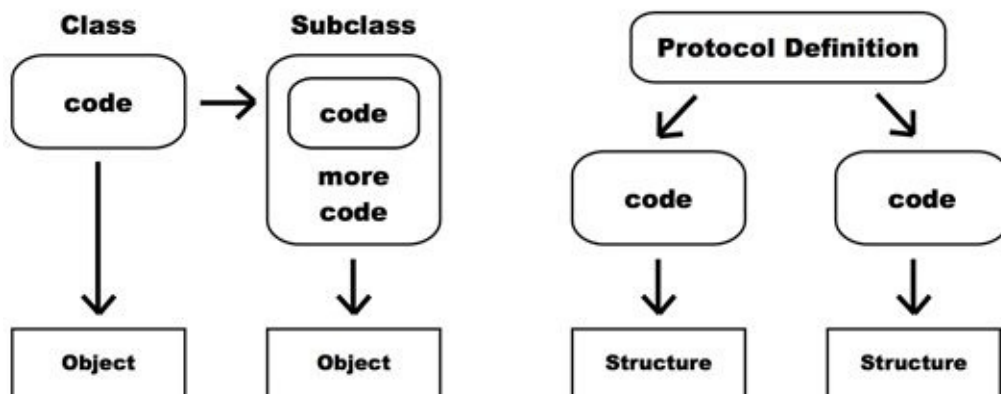


Figure 3-6: Object Oriented Programming and Protocol Oriented Programming

In OOP, the code is implemented inside a class and then objects are created from that class. If we need to create objects with additional functionality, we define a subclass that inherits the code from the class and adds some of its own. Protocols offer a slightly different approach. The properties and methods we want the structures to have in common are defined in the protocol and then implemented by the structures' definitions. This lets us associate different structures together through a common pattern. The code implemented by each structure is unique, but they follow a blueprint set by the protocol. If we know that a structure adopts a protocol, we can always be sure that besides its own definitions, it will also include the properties and methods defined by the protocol. In addition, protocols can be extended to provide their own implementations of the properties and methods we want the structures to have in common, allowing the paradigm to completely replace classes and objects and offer a more intuitive way to create our programs.

The Basics: Swift paradigm is built from the combination of structures and protocols, but protocols may also be adopted by enumerations and classes. Objective-C and the frameworks programmed in this language use protocols to offer a programming pattern called *delegation*. We will study how to adopt protocols from classes and how to implement the delegation pattern later in this chapter.

Definition of Protocols

Protocols are defined with the **protocol** keyword followed by the name and the list of properties and methods between braces. No values or statements are assigned or declared inside a protocol, only the names and the corresponding types; the structure that adopts the protocol is in charge of the implementation. Because of this, methods are defined as always, but omit the braces and the statements, and properties have to include the **get** and **set** keywords between braces to indicate whether they are read-only properties or we can read and also assign values to them (see Listing 3-39 for an example of getters and setters). The adoption of the protocol by the structure is declared including the protocol's name after the structure's name separated by a colon, as shown in the following example.

Listing 3-124: Defining protocols

```
protocol Printer {
    var name: String { get set }
    func printdescription()
}

struct Employees: Printer {
    var name: String
    var age: Int
    func printdescription() {
        print("Description: \(name) \(age)")
    }
}

let employee1 = Employees(name: "John", age: 32)
employee1.printdescription() // "Description: John 32"
```

The protocol is only a blueprint; it tells the structure what properties and methods are required, but the structure has to provide its own implementations. In the example of Listing 3-124 we define a protocol called **Printer** that includes the **name** property and the **printdescription()** method. The **Employees** structure defined next adopts this protocol, and along with the protocol's property and method it also implements its own property called **age**. Although this property was not defined in the protocol, we can read it inside the **printdescription()** method and print its value. The advantage of this practice is evident when structures of different types adopt the same protocol, as shown in the following example.

Listing 3-125: Defining multiple structures that adopt the same protocol

```

protocol Printer {
    var name: String { get set }
    func printdescription()
}

struct Employees: Printer {
    var name: String
    var age: Int
    func printdescription() {
        print("Description: \(name) \(age)")
    }
}

struct Offices: Printer {
    var name: String
    var employees: Int
    func printdescription() {
        print("Description: \(name) \(employees)")
    }
}

let employee1 = Employees(name: "John", age: 32)
let office1 = Offices(name: "Mail", employees: 2)
office1.printdescription() // "Description: Mail 2"

```

Although the structures created in Listing 3-125 from the `Employees` and `Offices` definitions are different (they have different properties), they both adopt the `Printer` protocol and provide their own implementation of the `printdescription()` method. The common functionality defined by the protocol assures us that no matter what type of structure we are working with, it will always have an implementation of the `printdescription()` method.

IMPORTANT: When a structure adopts a protocol, it has to implement its properties and methods. If you forget to implement any property or method defined in the protocol, Xcode shows you an error.

Protocols not only act as blueprints, they are also considered to be data types. And this is the real advantage of using protocols. We can treat a structure as if it is of the data type of the protocol it adopted. This lets us associate structures by their common functionality.

Listing 3-126: Using protocols as data types

```

let employee1 = Employees(name: "John", age: 32)

```

```
let office1 = Offices(name: "Mail", employees: 2)
```

```
var list: [Printer] = [employee1, office1]
```

```
for element in list {
```

```
    element.printdescription()
```

```
}
```

Listing 3-126 uses the same protocol and structures defined in the previous example, but this time it stores the instances in an array. The type of the array was defined as **Printer**, which means the array may contain structures of any type as long as they adopt the **Printer** protocol. Because of this, no matter the element's data type (**Employees** or **Offices**) we know that they always have an implementation of the **name** property and the **printdescription()** method that we can read or execute.

Because protocols are data types, we can use them as any other type, define variables, send and receive them from functions, etc. The following example declares a function that returns a value of type **Printer**.

Listing 3-127: Returning values of a protocol data type

```
func getFile(type: Int) -> Printer {
```

```
    var data: Printer!
```

```
    switch type {
```

```
    case 1:
```

```
        data = Employees(name: "John", age: 32)
```

```
    case 2:
```

```
        data = Offices(name: "Mail", employees: 2)
```

```
    default:
```

```
        break
```

```
    }
```

```
    return data
```

```
}
```

```
let file = getFile(1)
```

```
file.printdescription() // "Description: John 32"
```

The **getFile()** function of Listing 3-127 creates an instance of a structure depending on the value received. If the **type** parameter is equal to 1, it returns an instance of **Employees**, but if the value is equal to 2, it returns an instance of **Offices**. But because the value returned by the function is of **Printer** type we know it will always include the **printdescription()** method.

When we process a structure as a protocol type we can only access the properties and methods defined by the protocol. If we need to access the properties and methods defined by the structure, we have to cast it using the **is** and **as** operators studied before (see Listing

3-115).

Listing 3-128: *Accessing the properties defined in the `Employees` structure*

```
let employee1 = Employees(name: "John", age: 32)
```

```
let office1 = Offices(name: "Mail", employees: 2)
```

```
var list: [Printer] = [employee1, office1]
```

```
for element in list {
```

```
    if element is Employees {
```

```
        let employee = element as! Employees
```

```
        print(employee.age) // "32"
```

```
    }
```

```
    element.printdescription()
```

```
}
```

Once we cast the element to the original data type, we can access its own specific properties and methods. In Listing 3-128, we check the type of the element with the `is` operator inside the loop, and if the element is of type `Employees` we cast it with the `as!` operator and read its `age` property.

Do It Yourself: The last three examples use the protocol and structures defined in Listing 3-125. Copy the definitions of Listing 3-125 in your Playground’s file and add the code of the example you want to try.

IMPORTANT: Properties and methods may be declared as optional with the `optional` keyword. Members declared as optional don’t have to be implemented by the class that adopts the protocol (they can be implemented only when we need them). At this moment, protocols in Swift have to be converted to Objective-C protocols prefixing the definition with the `@objc` attribute to be able to support optionals.

Extensions

Protocols only define the properties and methods that structures are going to have in common, but they don’t include any implementation. This is good when we have unique implementations for every type of structure, but force us to repeat code when the implementations are the same. To avoid duplication we can take advantage of a Swift’s feature called *extensions*. Extensions are special declarations that add functionality to an

existent data type. We can use them with structures, enumerations, and classes, but they are particularly useful with protocols because this is the way protocols can provide their own implementations of properties and methods. The following example adds an extension to the **Printer** protocol that implements the **printdescription()** method.

Listing 3-129: Extending the protocol

```
protocol Printer {
    var name: String { get set }
}

extension Printer {
    func printdescription() {
        print("The name is \(name)")
    }
}

struct Employees: Printer {
    var name: String
    var age: Int
}

struct Offices: Printer {
    var name: String
    var employees: Int
}

let employee1 = Employees(name: "John", age: 45)
let office1 = Offices(name: "Mail", employees: 2)

office1.printdescription() // "The name is Mail"
```

The syntax includes the **extension** keyword followed by the name of the data type we want to extend. In the example of Listing 3-129, we define a **Printer** protocol with just the **name** property and then extend it to include a common implementation of the **printdescription()** method. Now the **Employees** and **Offices** structures share the same implementation and produce the same result when their **printdescription()** methods are executed.

As we mentioned, extensions are not only available for protocols but also for any other data type. We can use them to extend structures, enumerations, or classes. This is particularly useful when we don't have access to the definitions of the data types and we need to add some functionality (like when they are part of a library or framework). In the following example we extend the **Int** structure to provide a method that prints a description of its value.

Listing 3-130: Extending other data types

```
extension Int {  
    func printdescription() {  
        print("The number is \(self)")  
    }  
}  
  
let number = 25  
number.printdescription() // "The number is 25"
```

The `Int` type is a structure defined in the Swift standard library. We can't modify its definition but we can extend it to add more functionality. In this last example, we add a method called `printdescription()` to print a message with the current value (notice the use of the `self` keyword to refer to the value of the instance). This method is not included in the original definition but it is now available for our code.

Delegates

As objects, structures can also be assigned to another structure's property. For example, we could have an instance of a structure called `Employees` with a property that contains an instance of a structure called `Offices` to store information about the office where the employee works. This opens the door to new programming patterns where structures adopt different roles. The most useful pattern is called *delegation*. A structure delegates responsibility for the execution of certain tasks to another structure.

Listing 3-131: Delegating tasks

```
struct Salary {  
    func showMoney(name: String, money: Double) {  
        print("The salary of \(name) is \(money)")  
    }  
}  
  
struct Employees {  
    var name: String  
    var money: Double  
    var delegate: Salary  
    func generatereport() {  
        delegate.showMoney(name, money: money)  
    }  
}
```

```

let salary = Salary()
var employee1 = Employees(name: "John", money: 45000, delegate: salary)

employee1.generatereport() // "The salary of John is 45000.0"

```

The **Employees** structure of Listing 3-131 contains three properties. The properties **name** and **money** are in charge of the employee's data, while the **delegate** property stores the instance of the **Salary** structure in charge of printing that data (the instance created from **Employees** stores the data and delegates the task of printing that data to the instance created from **Salary**). The code creates the **Salary** instance first and then uses it to create the **Employees** instance. When we call the **generatereport()** method on the **employee1** structure at the end, the method calls the **showmoney()** method on **delegate**, effectively delegating the task of printing the data to this object.

This pattern presents two problems. First, the structure that is delegating has to know the data type of the structure that is going to become the delegate (in our example, the **delegate** property had to be declared of type **Salary**). Following this approach, not every structure can be a delegate, only the ones specified in the definition (only structures of type **Salary**). The second problem is related to how we know which are the properties and methods the delegate has to implement. If the structure is too complex or is taken from a library, we could forget to implement some methods or properties and get an error when the structure tries to access them. Both problems are solved by protocols. Instead of declaring a specific structure as the delegate, we define a protocol and declare the **delegate** property to be of that type, as shown in the following example.

Listing 3-132: Following protocols

```

protocol SalaryProtocol {
    func showMoney(name: String, money: Double)
}

struct Salary: SalaryProtocol {
    func showMoney(name: String, money: Double) {
        print("The salary of \$(name) is \$(money)")
    }
}

struct Employees {
    var name: String
    var money: Double

    var delegate: SalaryProtocol

    func generatereport() {
        delegate.showMoney(name, money: money)
    }

```

```

}
let salary = Salary()
let employee1 = Employees(name: "John", money: 45000, delegate: salary)

employee1.generatereport() // "The salary of John is 45000.0"

```

The advantage of protocols is that we can use structures of different types to perform the task. It doesn't matter what type they are as long as they adopt the delegate's protocol and implement its properties and methods. For example, we could create two different structures to print the data of our last example and assign to the delegate one instance or another depending on what we want to achieve at that particular time.

Listing 3-133: Using different delegates

```

protocol SalaryProtocol {
    func showMoney(name: String, money: Double)
}

struct Salary: SalaryProtocol {
    func showMoney(name: String, money: Double) {
        print("The salary of \(name) is \(money)")
    }
}

struct BasicSalary: SalaryProtocol {
    func showMoney(name: String, money: Double) {
        if money > 40000 {
            print("Salary is over the minimum")
        else {
            print("The salary of \(name) is \(money)")
        }
    }
}

struct Employees {
    var name: String
    var money: Double
    var delegate: SalaryProtocol
    func generatereport() {
        delegate.showMoney(name, money: money)
    }
}

let salary = Salary()
var employee1 = Employees(name: "John", money: 45000, delegate: salary)

```

```
employee1.delegate = BasicSalary()
```

```
employee1.generatereport() // "Salary is over the minimum"
```

The **BasicSalary** structure added in Listing 3-133 adopts the **SalaryProtocol** and implements its **showMoney()** method, but instead of printing a single message it produces two different results depending on the current employee's salary. The output produced by the execution of the **generatereport()** method now depends on the type of structure we previously assigned to the **delegate** property.

The Basics: Swift proposes its new paradigm that integrates structures with protocols, but most of the frameworks provided by Apple are still programmed in Objective-C using classes. The delegate pattern is extensively used in these frameworks to communicate objects and make them work together. Later, we will see more examples of their application in practical situations.

Chapter 4

Frameworks

4.1 iOS SDK

The programming tools studied in previous chapters are not enough to build a useful application. Creating an app involves accessing intricate features and technologies and also performing complex tasks that usually require hundreds or sometimes thousands of lines of code. Because the tasks are similar in every application, developers usually end up programming the same code over and over again. Considering this situation, systems always provide pre-programmed code that perform common tasks and allow developers to focus on more meaningful goals. These pieces of code (classes, structures, functions, etc.) are organized according to their purpose in groups called *libraries*, and when these libraries share a common goal they are grouped together in what are called *frameworks*.

Apple has been building libraries and frameworks to help developers create professional applications since the production of the first personal computer. These frameworks were developed in different languages and through a long period of time, resulting in a very diverse set of tools, with old and new frameworks that sometimes overlap and even replace each other. Some of them conform what is now known as the iOS SDK (Software Development Kit) that is included in the package we get along with Xcode.

The iOS SDK includes frameworks of different sizes and characteristics, including some for general use and others with very specific functionality. Some cover basic aspects of mobile app development such as processing data, showing graphics on the screen or managing user interaction, while others provide access to sophisticated hardware included in Apple's devices. Basically, what the frameworks do is to give us definitions of functions, structures, and classes that we later instantiate to get the blocks of code our application needs to work. To learn how to program applications, we have to learn what these frameworks provide and how to implement them in our code.

IMPORTANT: In this chapter we will study a few aspects of some of the frameworks that you will use extensively in your applications. We only explain the features that are considered essential for app development and are required in the rest of the examples of this book. For a complete list of functions, classes, properties, and methods, read the official references published by Apple at **developer.apple.com** or visit our website and follow the links for this chapter.

Importing Frameworks

The standard libraries included with the Swift language are automatically loaded for us, but when our code requires the use of external frameworks we have to indicate it to the

system. This is done by adding the **import** instruction at the beginning of each of the files, followed by the name of the framework we want to include (e.g., **import Foundation**). Once the framework is imported, it is automatically included in our file, giving us access to all the structures, classes, and any of the values defined in its code.

4.2 Foundation

Foundation is probably one of the first frameworks provided by Apple. It was written in Objective-C and developed by Steve Jobs's second company NeXT (that is why the names of its classes are prefixed with the NS letters, which stands for NeXTstep, the operative system developed by NeXT before being acquired by Apple). The framework was created to manage basic tasks but also to store simple data. Data storage is actually one of its most important functions. Foundation provides its own data types (structures and classes) to store numbers and strings of characters, to create arrays and dictionaries, and even a primary class called `NSObject` with basic behaviour that every other class inherits from. Most of these definitions are now obsolete, replaced by Swift's data types, and others automatically bridge to Swift, so we don't need to know them, but we can still use them to access some functionalities that remain essential for app development.

IMPORTANT: We will study parts of the Foundation framework in this chapter and explore the rest later in more practical situations. For a complete list of the classes and tools included in this framework, visit our website and follow the links for this chapter.

More Standard Functions

No matter the language, there are simple operations that are always required by any program. The codes to perform tasks such as calculate the power of a number or generate a random value were always provided through elemental libraries. Objective-C took advantage of these libraries and Swift is no exception, but the language makes them available through diverse paths. Some functions are provided by the Swift's standard library while others are loaded from old libraries by frameworks like Foundation. The libraries automatically loaded by the Foundation framework include generic functions that we can use to perform basic operations. The following are the most frequently used.

floor(value)—This function returns the largest integer number not greater than **value** (it rounds down the number). The **value** attribute is a number of type **Double**.

ceil(value)—This function returns the smallest integer number greater or equal than **value** (it rounds up the number). The **value** attribute is a number of type **Double**.

round(value)—This function rounds up or down **value** depending on the approximation to the nearest integer (e.g., **round(3.6)** returns 4.0, while **round(3.4)** returns 3.0). The **value** attribute is a number of type **Double**.

arc4random()—This function returns a random number. It has to be combined with the remainder operator to get a number from a predefined range (e.g., **arc4random()** % 10 returns a number between 0 and 9). There is a version of this function called **arc4random_uniform(value)** that doesn't require the use of the remainder. This additional function returns a random number between 0 and **value**.

pow(value, exponent)—This function returns the result of raising **value** to the power of **exponent**. The attributes are numbers of type **Double**.

sqrt(value)—This function returns the square root of **value**. The **value** attribute is a number of type **Double**.

log(value)—This function returns the natural logarithm of **value**. Other related functions are **log2()**, **log10()**, **log1p()**, and **logb()**. The **value** attribute is a number of type **Double**.

sin(value)—This function returns the sine of **value**. Other related functions are **asin()**, **sinh()**, and **asinh()**. The **value** attribute is a number of type **Double**.

cos(value)—This function returns the cosine of **value**. Other related functions are **acos()**, **cosh()**, and **acosh()**. The **value** attribute is a number of type **Double**.

tan(value)—This function returns the tangent of **value**. Other related functions are **atan()**, **atan2()**, **tanh()**, and **atanh()**. The **value** attribute is a number of type **Double**.

Most libraries also offer predefined values. These values work like constant values; they are represented by a name and can't be changed. When the compiler finds them, it replaces their names with the corresponding value before compiling the program. The following is a list with the most frequently used.

M_PI—This name represents the value of PI. Other related names are **M_PI_2** (PI divided by 2), **M_PI_4** (PI divided by 4), **M_1_PI** (1 divided by PI), and **M_2_PI** (two times 1 divided by PI).

M_E—This name represents the value of the base of the natural logarithm. Other related names are **M_LOG2E** (the logarithm base 2 of **M_E**) and **M_LOG10E** (the logarithm base 10 of **M_E**).

The application of these functions and values is straightforward, as shown in the following example.

Listing 4-1: Applying math functions

```
import Foundation
```

```
let square = sqrt(4.0)
let power = pow(2.0, 2.0)
let maximum = max(square, power)

print("The maximum value is \(maximum)") // "The maximum value is 4.0"
```

The first thing we do in the code of Listing 4-1 is to import Foundation. After this, we can implement any of the tools defined inside the framework, including the generic functions introduced before. This example gets the square root of 4.0, calculates 2.0 to the power of 2.0, and then compares the results using the `max()` function from the Swift's standard library. The following example demonstrates how to calculate a random number.

Listing 4-2: Calculating random numbers

```
import Foundation

var randomnumber = arc4random_uniform(10) + 1
print("The number is: \(randomnumber)")
```

The code of Listing 4-2 uses the `arc4random_uniform()` function to get a random number between 1 and 10. As we explained before, this function doesn't require the use of the remainder operator, but it calculates the number starting from 0. The values produced by `arc4random_uniform(10)` will be from 0 to 9. To get a random number from 1 to 10 (the maximum value we really want), we have to add 1 to the result.

Strings

Swift was developed to work seamlessly with the existent frameworks, and can automatically convert some of its data types into old Objective-C data types and vice versa. One of these data types is `String`. There is a class in the Foundation framework called `NSString` that was used to store and manage strings in codes programmed in Objective-C. Swift converts `String` values into `NSString` objects and the other way around every time possible. This turns the `NSString` class obsolete in Swift, because all of its properties and methods are available from `String` values. But in some circumstances we may need to create an `NSString` object from scratch, especially when we are working with old frameworks and classes that only take these types of values. The `NSString` class includes several Initializers. The one usually implemented in Swift to create an `NSString` object takes an attribute called `string` with the string of characters we want to assign to the object.

Listing 4-3: Creating an NSString object

```
import Foundation  
  
var text: NSString = NSString(string: "Hello")
```

The Basics: The `NSString` name is just the name of the class, like the custom `Employee` class created in Chapter 3. The difference between these classes and our own is that we don't have to define them because all their properties and methods were already defined for us inside the framework.

If we already have a `String` value in our code, we can cast it into an `NSString` object with the `as` operator.

Listing 4-4: Converting a String value into an NSString object

```
import Foundation  
  
var text = "Hello World"  
  
var newText = text as NSString
```

As we mentioned, most of the properties and methods defined in the `NSString` class are available from `string` values. This also includes some of its initializers. For instance, the following is a very useful initializer we can use to format strings.

`String(format: string, arguments: arguments)`—This initializer creates a string from the string provided by the **`string`** attribute and the array of values provided by the **`arguments`** attribute. The **`string`** attribute is a template used to create the string, and the **`arguments`** attribute is an array with the values we want to include in the string.

Although we can incorporate values into strings with string interpolation (see Chapter 2, Listing 2-25), sometimes the code may get complicated and difficult to maintain. The **`String(format: string, arguments: arguments)`** initializer is used to create more complex formats. The initializer takes a string and replaces the placeholders inside the string with values from an array. The placeholders are constructed with the `%` symbol followed by a character that represents the type of value we want to include. For example, if we want the placeholder to be replaced by an integer, we have to use the characters `%d`.

Listing 4-5: Creating a formatted string

```
import Foundation
```

```
var age = 44
var mytext = String(format: "My age is %d", arguments: [age])
```

The code of Listing 4-5 replaces the %d characters in the string with the value of the `age` variable and stores the resulting string in the `mytext` variable (“My age is 44”).

There are different placeholders available. The most frequently used are %d for integers, %f for floating-point numbers, and %@ for objects (including strings and numbers that Swift can convert to objects). We can use any of these characters and as many as we need. All we have to remember is to select the characters according to the type of value we want to insert and respect the order.

Listing 4-6: Formatting a string with multiple values

```
import Foundation

var age = 44
var mytext = String(format: "My %@ is %d", arguments: ["age", age])
```

The code of Listing 4-6 replaces the placeholders in the string with two values: a string and an integer. The string stored in the `mytext` variable will be “My age is 44”. This is straightforward and very similar to what we would obtain with string interpolation, but there are things we can’t do with other methods. For example, there is a syntax for the %f placeholder that may be useful when working with floating-point numbers. This placeholder can include the amount of decimal digits we want to consider. For example, the characters %.2f are replaced by a `Float` or a `Double` value with only two digits after the decimal point.

Listing 4-7: Formatting a string with floating-point values

```
import Foundation

var length = 12.3472
var mytext = String(format: "%.2f meters", arguments: [length]) // 12.35
```

As we have seen in Chapter 3, `String` structure can handle strings very efficiently, but there are still a few `NSString` methods that can simplify our work. For example, the following methods provide better options to compare strings.

compare(string)—This method compares the original string with the string provided by the `string` attribute and returns an enumeration of type `NSComparisonResult` with a value

corresponding to the lexical order of the strings. The **OrderedSame** value is returned when the strings are equal, the **OrderedAscending** value is returned when the original string precedes the value of the **string** attribute, and the **OrderedDescending** value is returned when the original string follows the value of the **string** attribute.

compare(string, options: options)—This method compares the original string with the string provided by the **string** attribute. It works exactly like the **compare()** method but adds the possibility to define searching options. The **options** attribute is a property of the **NSStringCompareOptions** structure. The properties available are **CaseInsensitiveSearch** (it considers lowercase and uppercase letters to be the same), **LiteralSearch** (performs a byte-to-byte comparison), **DiacriticInsensitiveSearch** (ignores diacritic marks such as the visual stress on vowels), **WidthInsensitiveSearch** (ignores the width difference in characters that occurs in some languages), and **ForcedOrderingSearch** (the comparison is forced to return **OrderingAscending** or **OrderingDescending** when the strings are equivalent but not strictly equal).

compare(string, options: options, range: range)—This method compares the original string with the string provided by the **string** attribute. It works exactly like the **compare()** method but adds the possibility to define a range that determines the portion of the original string we want to compare.

caseInsensitiveCompare(string)—This method compares the original string with the string provided by the **string** attribute. It works exactly like the **compare()** method but with the option **CaseInsensitiveSearch** set by default.

These methods may be used to search for a value inside a string, but the **NSString** class offers more specific methods to do it.

rangeOfString(string)—This method searches for the string specified by the **string** attribute and returns a range to indicate where the string was found or **nil** in case of failure.

rangeOfString(string, options: options)—This method searches for the string specified by the **string** attribute. It works exactly like the **rangeOfString()** method but adds the possibility to define searching options. The **options** attribute is a property of the **NSStringCompareOptions** structure. The properties available for this method are the same we have for the **compare()** method, with the difference that we can specify three more: **BackwardsSearch** (searches from the end of the string), **AnchoredSearch** (matches characters only at the beginning or the end, not in the middle), and **RegularExpressionSearch** (searches with a regular expression).

rangeOfString(string, options: options, range: range)—This method searches for the string specified by the **string** attribute. It works exactly like the **rangeOfString()** method but adds the possibility to define a range that determines the portion of the

original string where we want to search.

With the `compare()` method we can compare two strings the same way we do with the `==` operator, but this time we get a more comprehensive result. Notice that, although these methods were defined in the `NSString` class we can call them from a `String` value.

Listing 4-8: Comparing String values

```
import Foundation

var string = "Orange"
var search = "Apple"

var result = string.compare(search)
switch result {
case .OrderedSame:
    print("String and Search are equal")
case .OrderedDescending:
    print("String follows Search")
case .OrderedAscending:
    print("String precedes Search")
}
```

The `compare()` method takes a string, compares it to the original string, and returns a value of the `NSComparisonResult` enumeration type to indicate the order. The `NSComparisonResult` enumeration type contains three values: `OrderedSame`, `OrderedDescending`, and `OrderedAscending`. After comparing the values of the `string` and `search` variables in Listing 4-8, the `result` variable contains one of these values according to the lexical order of the strings. In our example, the value "Orange" assigned to `string` is bigger (follows) the value "Apple" assigned to `search`, so the value returned is `OrderedDescending` (the order is descending from `string` to `search`). We check this value with a `switch` statement and print a message according to the result.

The `compare()` method implemented in Listing 4-8 and the `==` operator studied in Chapter 2 consider a lowercase string different than an uppercase string. Implementing other versions of the `compare()` method we can compare two strings without considering lowercase or uppercase letters.

Listing 4-9: Comparing String values with options

```
import Foundation

var string = "Orange"
```

```
var search = "ORANGE"
```

```
var result = string.compare(search, options: .CaseInsensitiveSearch)
```

```
switch result {
```

```
    case .OrderedSame:
```

```
        print("String and Search are equal")
```

```
    case .OrderedDescending:
```

```
        print("String follows Search")
```

```
    case .OrderedAscending:
```

```
        print("String precedes Search")
```

```
}
```

The strings stored in the `string` and `search` variables in Listing 4-9 are different, but because of the `CaseInsensitiveSearch` option, they are considered equal. This type of comparison is very common, which is why the class includes the `caseInsensitiveCompare()` method that all it does is call the `compare()` method with the `CaseInsensitiveSearch` option already set.

Despite this being the most common scenario, there are situations in which we need to be more precise. And that is what the third version of the `compare()` method is for. Adding the **range** attribute we can specify the range of characters in the original string we want to compare.

Listing 4-10: Comparing only a range of characters

```
import Foundation
```

```
var string = "905-525-6666"
```

```
var search = "905"
```

```
var start = string.startIndex
```

```
var end = string.characters.indexOf("-",)
```

```
if let endIndex = end {
```

```
    var result = string.compare(search, options: .CaseInsensitiveSearch, range: start..
```

```
    switch result {
```

```
        case .OrderedSame:
```

```
            print("String and Search are equal")
```

```
        case .OrderedDescending:
```

```
            print("String follows Search")
```

```
        case .OrderedAscending:
```

```
            print("String precedes Search")
```

```
    }
```

```
}
```


The example of Listing 4-10 compares only the initial characters of a string to check the area code of a phone number. The code defines a range that goes from the first character of the `string` variable to the position before the `-` character. This range is provided to the `compare()` method and in consequence the value of the `search` variable is compared only against the first three characters of the `string` variable.

Besides comparing strings we can also search for strings using the `rangeOfString()` method. This method searches for a string inside another string and returns a range that determines where the string was found.

Listing 4-11: Searching and replacing characters in a string

```
import Foundation
```

```
var string = "The Suitcase is Black"
```

```
var search = "black"
```

```
var range = string.rangeOfString(search, options: .CaseInsensitiveSearch)
```

```
if let rangeToReplace = range {
```

```
    string.replaceRange(rangeToReplace, with: "Red") // "The Suitcase is Red"
```

```
} else {
```

```
    print("Not Found")
```

```
}
```

The `rangeOfString()` method returns an optional value that contains the range where the string was found or `nil` in case of failure. In Listing 4-11 we search for the value of the `search` variable inside the `string` variable and check the optional value returned. When we have a range to work with (which means that the value was found) we use it to call the `replaceRange()` method (introduced in Chapter 3). This method replaces the characters inside the range by the string `"Red"`. Notice that, as we did before with the `compare()` method, we use the `CaseInsensitiveSearch` option to ignore the difference between lowercase and uppercase letters.

Ranges

Although Swift includes the `Range` structure to store ranges of values, some old libraries programmed in Objective-C still implement an old Foundation class called `NSRange`. The `NSRange` class is slightly different than `Range`. Instead of storing the initial and final values of the range, the `NSRange` objects store the initial value and the length of the string of characters. The `NSRange` initializer requires these two values to create an object.

NSRange(location: **start, length: **length**)**—This initializer creates an **NSRange** object to represent a range that goes from the value provided by the **start** attribute and to a distance provided by the **length** attribute. The attributes are values of type **Int**.

Just as the **Range** structure has the **startIndex** and **endIndex** properties, the **NSRange** class also includes two properties to retrieve its values: **location** and **length**.

Listing 4-12: Creating and reading an NSRange value

```
import Foundation

let range = NSRange(location: 5, length: 2)
print("From \(range.location) to \(range.location + range.length)")
```

The **String** values are easily converted to **NSString** objects with the **as** operator, as we have seen in this chapter, but conversions between **NSRange** objects and **Range** structures require a little bit of work. Since there is no direct conversion available at this moment, their values have to be processed one by one. For example, to convert an **NSRange** object to a **Range** structure we have to calculate the initial and final values.

Listing 4-13: Converting an NSRange object into a Range structure

```
import Foundation

let range = NSRange(location: 3, length: 5)

let start = range.location
let end = range.location + range.length
let swiftRange = Range(start: start, end: end) // 3..<8
```

The reverse conversion requires the use of the **distanceTo()** method (see Chapter 3). This method compares two indexes and returns an integer representing the difference between them. Since **NSRange** objects work with integers, we can use it to convert the initial and final values of a **Range** structure to the location and length values of an **NSRange** object.

Listing 4-14: Converting Range values into NSRange objects

```
import Foundation

let text = "Hello World"
```

```

let range = text.rangeOfString("World") // 6..<11

if let tempRange = range {
    let location = text.startIndex.distanceTo(tempRange.startIndex)
    let length = tempRange.startIndex.distanceTo(tempRange.endIndex)
    let newRange = NSRange(location: location, length: length) // (6, 5)
}

```

In Listing 4-14, we search for the string “World” inside the value of the `text` constant and get a `Range` value in return. From this range we calculate the location of the initial character and the length of the string to create a new `NSRange` object. The location is obtained calculating the distance between the position of the string’s first character and the range’s first value. For the length we just need to get the distance between the initial and final values of the original range. The `NSRange` object is created next using these results.

IMPORTANT: It’s not safe to convert `Range` structures into `NSRange` objects (or the other way around) when working with strings. In these situations it is better to convert the string to the right type first and then work with the corresponding range type. For example, if you have a `String` value and need an `NSRange` object to work with some of its characters, don’t try to get a `Range` structure and then convert it into an `NSRange` object; convert the string first to an `NSString` object and then get the `NSRange` object from this value (see Chapter 7, Listing 7-12).

Numbers

Foundation offers a class called `NSNumber` to represent and store different types of numbers. This was, along with the primitive types offered by the C language, the default tool to store and work with numbers before Swift. With the inclusion of the Swift’s primitive types, the use of this class is no longer necessary, but there are a few old frameworks programmed in Objective-C that still require these types of values. To create an `NSNumber` object from a Swift primitive type, we have to use the following initializers.

`NSNumber(bool: value)`—This initializer creates an `NSNumber` with the value specified by the `value` attribute. The attribute is of type `Bool`.

`NSNumber(integer: value)`—This initializer creates an `NSNumber` with the value specified by the `value` attribute. The attribute is of type `Int`.

`NSNumber(float: value)`—This initializer creates an `NSNumber` with the value specified by the `value` attribute. The attribute is of type `Float`.

NSNumber(double: value)—This initializer creates an **NSNumber** with the value specified by the **value** attribute. The attribute is of type **Double**.

The class also provides properties to perform the opposite operation, getting Swift primitive types from **NSNumber** objects.

boolValue—This property returns a **Bool** value with the object's value.

integerValue—This property returns an **Int** value with the object's number.

floatValue—This property returns a **Float** value with the object's number.

doubleValue—This property returns a **Double** value with the object's number.

The following example shows how to create **NSNumber** objects and how to get them back as Swift primitive types. Once the values are converted, we can perform arithmetic operations on them as we do with any primitive type. In further chapters we will implement them in more practical situations.

Listing 4-15: Working with NSNumber objects

```
import Foundation
```

```
var number = NSNumber(double: 35)
var duplicate = number.doubleValue * 2 // 70
```

Foundation also includes two classes related to **NSNumber** that are very useful when working with decimal numbers or we need to print a value on the screen.

NSNumberFormatter—This class provides properties and methods to create a textual representation of a number.

NSDecimalNumber—This class provides properties and methods to work with decimal numbers in a specific format.

The **Float** and **Double** types in Swift always use the maximum amount of digits allowed for their type to store a value. Every time we read a variable of these types, we get the whole number, including all the numbers after the decimal point. This is desirable when performing calculations, but not appropriate when we want to print those results on the screen. Most of the time, the values that users need to see have a limited number of digits

or demand a specific representation. The `NSNumberFormatter` class was defined to create a string that contains a number in a specific format. The class offers properties and methods to work with standard formats or create our own.

numberStyle—This property sets or returns a value that determines the style used to format the number. It is an enumeration of type `NSNumberFormatterStyle` with the values `NoStyle` (default), `DecimalStyle`, `CurrencyStyle`, `PercentStyle`, `ScientificStyle`, and `SpellOutStyle`.

roundingMode—This property sets or returns the rounding mode. It is an enumeration of type `NSNumberFormatterRoundingMode` with the values `RoundCeiling`, `RoundFloor`, `RoundDown`, `RoundUp`, `RoundHalfEven` (default), `RoundHalfDown`, and `RoundHalfUp`.

minimumFractionDigits—This property sets or returns an integer value that determines the minimum number of digits allowed after the decimal point.

maximumFractionDigits—This property sets or returns an integer value that determines the maximum number of digits allowed after the decimal point.

stringFromNumber(number)—This method returns a string that represents the number provided by the **number** attribute in the format specified by the properties of the object.

The default configuration of an `NSNumberFormatter` object sets the style as `NoStyle` and the rounding mode as `RoundHalfEven`, which means the string produced by the `stringFromNumber()` method will be composed of a number with the decimal part rounded to the nearest integer, but we can change this behaviour assigning new values to its properties.

***Listing 4-16:** Formatting a number to represent money*

```
import Foundation

var money = 5.6897

let format = NSNumberFormatter()
format.numberStyle = NSNumberFormatterStyle.CurrencyStyle
format.roundingMode = NSNumberFormatterRoundingMode.RoundFloor

var price = format.stringFromNumber(money) // $ 5.68
```

The process involves the creation of the `NSNumberFormatter` object, the configuration of its properties, and the execution of its `stringFromNumber()` method to get the string we need. In the example of Listing 4-16, we create an `NSNumberFormatter` object and then set its style

as `CurrencyStyle` and its rounding mode as `RoundFloor`. The number stored in the `money` variable is sent to the `stringFromNumber()` method of this object to get the string in the specified format. The final result is a string with the currency symbol and the amount of decimal digits corresponding to the device's configuration. For example, if the device was set in the United States, the string produce by this example will be “\$5.68”.

If we want to format a number with a specific amount of digits, we can take advantage of the `minimumFractionDigits` and `maximumFractionDigits` properties.

Listing 4-17: Specifying the number of digits

```
import Foundation

var money = 5.6897

let format = NSNumberFormatter()
format.numberStyle = NSNumberFormatterStyle.DecimalStyle
format.minimumFractionDigits = 3
format.maximumFractionDigits = 3

var price = format.stringFromNumber(money) // $ 5.690
```

Once the `NSNumberFormatter` object is created, it can be used to format all the numbers we want. In the following example we create two strings to represent the price and the total price of a fictitious transaction.

Listing 4-18: Formatting a number to represent money

```
import Foundation

var money = 5.6897
var total = money * 4

let format = NSNumberFormatter()
format.numberStyle = .CurrencyStyle
format.roundingMode = .RoundCeiling

var price = format.stringFromNumber(money) // $5.69
var result = format.stringFromNumber(total) // $22.76
```

We can change the style and the rounding mode to get the format we need. The most common styles are `CurrencyStyle` to represent money, `PercentStyle` to represent percentages, and `SpellOutStyle` to represent the number in words (“five point six eight nine seven”). The rounding modes used more often are `RoundCeiling`, `RoundFloor`, and `RoundHalfEven` to round the

last digit up, down, or to the closest integer, respectively.

IMPORTANT: Besides the standard configurations, the `NSNumberFormatter` class offers multiple properties to set any custom format necessary. For example, instead of relying on the system to provide the symbol for the currency, you can specify it explicitly by assigning a new value to the `currencySymbol` property. To see all the possibilities, visit our website and follow the links for this chapter.

With the `NSNumberFormatter` class we can round and format the numbers that are going to be shown on the screen, but sometimes the operations performed to get those numbers require a specific amount of decimal digits to produce the right result. This is especially true when working with numbers that represent money. The numbers have to be rounded all the way through, usually to only two decimals. The `NSDecimalNumber` class was designed to solve this problem.

The `NSDecimalNumber` class stores a number and provides the methods to perform operations following a specific behaviour set for the class. The class provides a simple initializer to create an `NSDecimalNumber` object from a `Double`.

Listing 4-19: Creating `NSDecimalNumber` objects

```
import Foundation
```

```
var money = NSDecimalNumber(double: 35.57)
var quantity = NSDecimalNumber(double: 3.20)
```

Before performing any operation on these values we have to configure the behaviour of the class. This is established by an object of the `NSDecimalNumberHandler` class. The following is the specification of its initializer.

`NSDecimalNumberHandler(roundingMode: mode, scale: value, raiseOnExactness: Bool, raiseOnOverflow: Bool, raiseOnUnderflow: Bool, raiseOnDivideByZero: Bool)`—This initializer creates an `NSDecimalNumberHandler` object with the configuration set by its attributes.

The `NSDecimalNumberHandler` object is provided to the class through a type method (a method that is executed on the class itself) called `setDefaultBehavior()`. The following example shows how to create the object and assign it to the class.

Listing 4-20: Defining the behaviour of the `NSDecimalNumber` objects

import Foundation

```
let roundup = NSDecimalNumberHandler(roundingMode: NSRoundingMode.RoundPlain, scale: 2, raiseOnExactness: false, raiseOnOverflow: false, raiseOnUnderflow: false, raiseOnDivideByZero: true)
```

```
NSDecimalNumber.setDefaultBehavior(roundup)
```

```
var money = NSDecimalNumber(double: 35.57)
```

```
var quantity = NSDecimalNumber(double: 3.20)
```

The first step is to create the `NSDecimalNumberHandler` object. The attributes provide the values for the properties of the object that establish how the operations over `NSDecimalNumber` objects will be processed. The `roundingMode` attribute is an enumeration of type `NSRoundingMode` with the values `RoundUp` (rounds the last digit up), `RoundDown` (rounds the last digit down), `RoundPlain` (rounds the last digit up or down to the closest integer), and `RoundBankers` (rounds the last digit up or down to the closest integer, contemplating when the distance is even). The `scale` attribute sets the number of digits after the decimal point that should be contemplated when performing calculations. And the rest of the attributes establish how the system should respond when an error occurs (e.g., a value is divided by zero). The values of these attributes are usually those presented in the example of Listing 4-20, except for the rounding mode and the scale that may vary according to the country.

The operations are not performed by the arithmetic operators used in Swift, but by the methods provided by the `NSDecimalNumber` class:

`decimalNumberByAdding(number)`—This method adds the number of the object to the number provided by the `number` attribute and returns a new `NSDecimalNumber` object containing the result.

`decimalNumberBySubtracting(number)`—This method subtracts the number provided by the `number` attribute to the number of the object and returns a new `NSDecimalNumber` object containing the result.

`decimalNumberByMultiplyingBy(number)`—This method multiplies the number of the object to the number provided by the `number` attribute and returns a new `NSDecimalNumber` object containing the result.

`decimalNumberByDividingBy(number)`—This method divides the number of the object by the number provided by the `number` attribute and returns a new `NSDecimalNumber` object containing the result.

`decimalNumberByRaisingToPower(number)`—This method raises the number of the object to the power provided by the `number` attribute and returns a new `NSDecimalNumber` object containing the result.

decimalNumberByMultiplyingByPowerOf10(number)—This method multiplies the number of the object to the number 10 raised by the number provided by the **number** attribute and returns a new **NSDecimalNumber** object containing the result.

The following example shows how to prepare the class and perform the operations on two **NSDecimalNumber** objects with a specific behaviour.

Listing 4-21: *Performing operations with NSDecimalNumber objects*

```
import Foundation

let roundup = NSDecimalNumberHandler(roundingMode: NSRoundingMode.RoundPlain, scale: 2, raiseOnExactness:
false, raiseOnOverflow: false, raiseOnUnderflow: false, raiseOnDivideByZero: true)
NSDecimalNumber.setDefaultBehavior(roundup)

var money = NSDecimalNumber(double: 35.57)
var quantity = NSDecimalNumber(double: 3.20)

var total = money.decimalNumberByMultiplyingBy(quantity) // 113.82
```

The values provided as attributes of the **NSDecimalNumber**'s methods may be another **NSDecimalNumber** object, as in the example of Listing 4-21, or a value of any Swift type. Swift takes care of converting its primitive types to the objects required by the methods.

Do It Yourself: Copy the code of Listing 4-21 in your Playground's file. Replace the **decimalNumberByMultiplyingBy()** method by any of the methods provided by the class to check how they work.

To access the value of an **NSDecimalNumber** object, the class includes the **doubleValue** property. The property returns a value of type **Double** that can then be assigned to a variable or be used as a normal Swift value.

Listing 4-22: *Converting an NSDecimalNumber object into a Swift Double type*

```
import Foundation

let roundup = NSDecimalNumberHandler(roundingMode: NSRoundingMode.RoundPlain, scale: 2, raiseOnExactness:
false, raiseOnOverflow: false, raiseOnUnderflow: false, raiseOnDivideByZero: true)
NSDecimalNumber.setDefaultBehavior(roundup)

var money = NSDecimalNumber(double: 35.57)
var quantity = NSDecimalNumber(double: 3.20)
```

```
var total = money.decimalNumberByMultiplyingBy(quantity) // 113.82  
var finalvalue = 10 * total.doubleValue // 1138.2
```

IMPORTANT: An `NSDecimalNumber` object doesn't prepare the number to be shown to the user. For example, the number 3.20 is shown as 3.2, the last digit is not shown, no matter how many digits you have set for the scale. To show all the digits on the screen you have to format the value of the `NSDecimalNumber` object with the `NSNumberFormatter` class.

Dates

Often, classes have to be implemented together to achieve a common purpose. We have seen this pattern in the classes studied above, and Foundation also uses it to work with dates and times. The framework defines four specific classes that work together to create, process, and present dates.

NSDate—This class provides properties and methods to represent a single point in time (including date and time) with a generic calendar, time zone, and localization values. It provides properties and methods to compare dates and calculate intervals between dates.

NSCalendar—This class encapsulates information about calendars, and provides properties and methods to process date components, calculate dates, and create date objects from independent components.

NSDateComponents—This class provides a simple way to access the components of a date. It can be used to read and set new values for dates and also create new date objects from independent components.

NSDateFormatter—This class provides properties and methods to create textual representations of dates and time.

NSLocale—This class provides properties and methods to manage information about a certain region, such as language, country, symbols, etc.

NSTimeZone—This class provides properties and methods to manage time zones.

The essential class, the one that creates the object to store the actual date, is **NSDate**. The following are some of the initializers included in this class.

NSDate()—This initializer creates an `NSDate` object with the system’s current date.

NSDate(timeIntervalSinceNow: *time*)—This initializer creates an `NSDate` object with a date calculated from the addition of the current date plus the time specified by the *time* attribute. The *time* attribute is a value of type `NSTimeInterval` (a typealias of `Double`) that indicates how many seconds the date is from the initial date.

NSDate(timeInterval: *time*, sinceDate: *date*)—This initializer creates an object with a date calculated from the addition of the date specified by the *date* attribute plus the time specified by the *time* attribute. The *time* attribute is a value of type `NSTimeInterval` (a typealias of `Double`) that indicates how many seconds the date is from the initial date.

The following example shows different ways to initialize a date.

Listing 4-23: *Storing dates with NSDate objects*

```
import Foundation

var currentdate = NSDate()
var nextday = NSDate(timeIntervalSinceNow: 24 * 60 * 60)
var tendays = NSDate(timeInterval: -10 * 24 * 3600, sinceDate: nextday)
```

The value of the interval, for the initializers that require one, is specified in seconds. An easy way to calculate the seconds is multiplying every component. For example, the date for the `nextday` object created in Listing 4-23 is calculated adding 1 day to the current date. The amount of seconds in 1 day are calculated multiplying the 24 hours of the day times the 60 minutes in an hour times the 60 seconds in a minute ($24 * 60 * 60$). There are 86400 seconds in a day, but it is much easier to get them multiplying the components. For the `tendays` object we apply the same technique. The initializer used to get this object adds the interval to a specific date (`nextday`). The seconds are calculated by multiplying the components, albeit this time it multiplies the previous result by -10 to get a date 10 days previous to `nextday`.

IMPORTANT: These methods require a value of type `Double` to declare the interval in seconds, but instead of `Double` the framework calls it `NSTimeInterval`. This is a type alias, an alternative name for an existing type. Once defined, aliases are used exactly like regular type names. In Swift, these alternative names are defined by the `typealias` instruction (e.g., `typealias myinteger = Int`).

Besides these initializers, the class also includes two types of methods to create special dates.

distantFuture()—This method returns an `NSDate` object with a value that represents a date in a distant future.

distantPast()—This method returns an `NSDate` object with a value that represents a date in a distant past.

These methods produce values that are useful to set limits and order lists, as we will see in further chapters.

Listing 4-24: Creating special dates

```
import Foundation

var future = NSDate.distantFuture()
var past = NSDate.distantPast()
```

The `NSDate` class also includes methods to calculate and compare dates. The following are the most frequently used.

isEqualToDate(date)—This method compares the date in the `NSDate` object with the date specified by the **date** attribute and returns a Boolean value to report if they are equal or not.

earlierDate(date)—This method compares the date in the `NSDate` object with the date specified by the **date** attribute and returns another `NSDate` object with the earlier date.

laterDate(date)—This method compares the date in the `NSDate` object with the date specified by the **date** attribute and returns another `NSDate` object with the later date.

compare(date)—This method compares the date in the `NSDate` object with the date specified by the **date** attribute and returns an enumeration of type `NSComparisonResult` with a value corresponding to the temporal order of the dates. The possible values are `OrderedSame` (the dates are equal), `OrderedAscending` (the date is earlier in time than **date**), and `OrderedDescending` (the date is later in time than **date**).

timeIntervalSinceDate(date)—This method compares the date in the `NSDate` object with the date specified by the **date** attribute and returns the interval between both dates in seconds. The value returned is of type `NSTimeInterval` (a typealias of `Double`).

dateByAddingTimeInterval(interval)—This method adds the seconds specified by the **interval** attribute to the date in the `NSDate` object and returns a new `NSDate` object with the result. The attribute is a value of type `NSTimeInterval` (a typealias of `Double`).

Comparing dates and calculating the intervals between dates is a constant requirement in app development. For instance, we can calculate the time remaining for an event. The following example compares the current date with a date calculated from the value of the `days` variable (a value that could be provided by the user or may be specific for each event). If the date of the event is later in time than the current date, the code prints a message on the console to show the time remaining.

Listing 4-25: Comparing two dates

```
import Foundation

var days = 7

let today = NSDate()
let event = NSDate(timeIntervalSinceNow: Double(days) * 24 * 3600)

if today.compare(event) == .OrderedAscending {
    let interval = event.timeIntervalSinceDate(today)
    print("We have to wait \(interval) seconds")
}
```

The dates in `NSDate` objects are generic and therefore they don't have an associated calendar. This means that to be able to interpret the components in a date (year, month, day, etc.) we have to decide first in the context of which calendar the date is going to be interpreted. The calendar for a date is defined by an object of the `NSCalendar` class. This class provides properties and methods to process a date according to a specific calendar (Gregorian, Buddhist, Chinese, etc.). To initialize an `NSCalendar` object we have the following initializer and type method.

`NSCalendar(calendarIdentifier: calendar)`—This initializer creates an object with the calendar specified by its attribute. The **`calendar`** attribute is a predefined constant. There is a constant for every calendar available, with the `NSCalendarIdentifierGregorian` as the most widely used (it creates a Gregorian or Western calendar). The value returned is an optional with the calendar required or `nil` if the calendar is not supported by the system.

`currentCalendar()`—This type method returns an object with the current calendar set in the system.

An `NSCalendar` object includes the following properties and methods to manage the calendar and also to get and set new dates.

calendarIdentifier—This property returns a string that identifies the calendar.

locale—This property sets or returns the `NSLocale` object used by the `NSCalendar` object to process dates. The value by default is the `NSLocale` object defined by the system.

timeZone—This property sets or returns the `NSTimeZone` object used by the `NSCalendar` object to process dates. The value by default is the `NSTimeZone` object defined by the system.

components(units, fromDate: date)—This method returns the components indicated by the **units** attribute from the date indicated by the **date** attribute. The **units** attribute is a set with constants of the `NSCalendarUnit` structure. The most frequently used values are `Year`, `Month`, `Day`, `Hour`, `Minute`, and `Second`. The value returned is an `NSDateComponents` object.

components(units, fromDate: date1, toDate: date2, options: options)—This method returns the components indicated by the **units** attribute which values represent the difference between the dates specified by the **date1** and **date2** attributes. The **units** attribute is a set with constants of the `NSCalendarUnit` structure. The most frequently used values are `Year`, `Month`, `Day`, `Hour`, `Minute`, and `Second`. The **options** attribute is a set of values from the `NSCalendarOptions` structure to indicate how the overflow of the addition of one component should be carried to the next one. It is usually defined by default with an empty set (`()`).

dateByAddingComponents(components, toDate: date, options: options)—This method returns the result of adding the components indicated by the **components** attribute (an `NSDateComponents` object) to the date indicated by the **date** attribute. The value returned is an `NSDate` object. The **options** attribute is a set of values from the `NSCalendarOptions` structure to indicate how the overflow of the addition of one component should be carried to the next one. It is usually defined by default with an empty set (`()`).

dateFromComponents(components)—This method returns a date created from the values provided by the **components** attribute (an `NSDateComponents` object). The value returned is an `NSDate` object.

The `NSCalendar` class works along with the `NSDateComponents` class to read and return components from a date. The objects created from the `NSDateComponents` class include the following properties to read and set the values of the components.

year—This property sets or returns an integer that represents the number of the year.

month—This property sets or returns an integer that represents the number of the month.

day—This property sets or returns an integer that represents the number of the day.

hour—This property sets or returns an integer that represents the number of hours.

minute—This property sets or returns an integer that represents the number of minutes.

second—This property sets or returns an integer that represents the number of seconds.

weekday—This property sets or returns an integer that represents the number of the day in the week. In the Gregorian calendar, the values are assigned from Sunday, starting by the number 1.

The example of Listing 4-26, next, combines all these elements together to get the year of the current date.

Listing 4-26: Extracting components from a date

```
import Foundation

var today = NSDate()
var calendar = NSCalendar.currentCalendar()
var components = calendar.components(NSCalendarUnit.Year, fromDate: today)

print("The year is \ \(components.year)")
```

Every time we want to extract a component from a date we have to call one of the `components()` methods of an `NSCalendar` object, so the calendar has to be created first. In Listing 4-26, the `NSCalendar` object is created by the `currentCalendar()` method (the calendar set in the system), and then the `components()` method is executed to get the year.

Several components may be fetched at the same time declaring the values as a set. The following example gets the year, month and day of the current date.

Listing 4-27: Extracting multiple components from a date

```
import Foundation

var today = NSDate()
var calendar = NSCalendar.currentCalendar()
var components = calendar.components([.Year, .Month, .Day], fromDate: today)
```

```
print("Today \ (components.day)-\ (components.month)-\ (components.year)")
```

NSDateComponents objects are used to retrieve the components of existing dates and also to set the values for new dates. In the following example, a new **NSDate** object is created from the values of an **NSDateComponents** object.

Listing 4-28: Creating a new date from components

```
import Foundation

var calendar = NSCalendar.currentCalendar()
var components = NSDateComponents()
components.year = 1970
components.month = 8
components.day = 13

var birthday = calendar.dateFromComponents(components)
```

The **dateFromComponents()** method of the **NSCalendar** object returns a new date with the values provided by the **NSDateComponents** object. The components which values are not explicitly defined take values by default.

Generating a new date demands the use of a specific calendar. For example, in the code of Listing 4-28, the values for the components are declared with the format established by the Gregorian calendar. This means that in cases like this we can't rely on the calendar returned by the system. To be sure that the **NSCalendar** object contains the correct calendar, we have to set it ourselves from the **NSCalendar** initializer.

Listing 4-29: Using a Gregorian calendar

```
import Foundation

var calendar = NSCalendar(calendarIdentifier: NSCalendarIdentifierGregorian)
var components = NSDateComponents()
components.year = 1970
components.month = 8
components.day = 13

var birthday = calendar?.dateFromComponents(components)
```

The object returned by the **NSCalendar** initializer is an optional. To access the values of the object we have to unwrap it first using an **if** statement or conditional chaining, as we do in the last statement of Listing 4-29. If for any reason the initializer can't return a valid

calendar, the `calendar` variable will be `nil`.

Declaring a specific calendar is not only recommended when creating new dates but also when calculating dates by adding components, as in the following example.

Listing 4-30: Adding components to a date

```
import Foundation

var calendar = NSCalendar(calendarIdentifier: NSCalendarIdentifierGregorian)
var components = NSDateComponents()
components.day = 120

var today = NSDate()
var appointment = calendar?.dateByAddingComponents(components, toDate: today, options: [])
```

The `dateByAddingComponents()` method adds components to a date and returns a new `NSDate` object with the result. In Listing 4-30, the component `day` was set to a value of 120. The `dateByAddingComponents()` method takes this value and adds it to the date of the `today` object (the current date). The result stored in the `appointment` variable will be the `NSDate` object with the new date or `nil`, depending on the success of the creation of the `NSCalendar` object.

A common task when working with multiple dates is getting the lapse of time between dates, such as the hours remaining for a process to be finished or the days remaining for an event. The `NSCalendar` class includes a version of the `components()` method that allows us to compare two dates and get the difference expressed in a specific component.

Listing 4-31: Comparing dates

```
import Foundation

var calendar = NSCalendar.currentCalendar()
var components = NSDateComponents()
components.year = 1970
components.month = 8
components.day = 13

var today = NSDate()
var birthday = calendar.dateFromComponents(components)

if var olddate = birthday {
    var components = calendar.components(.Day, fromDate: olddate, toDate: today, options: [])
    print("Days between dates: \(components.day)")
}
```

This example calculates the days between a birthday and the current date. The value returned by the `dateFromComponents()` method, used to generate the date for the birthday, returns an optional. If the method is not able to calculate a date from the provided components, the value returned will be `nil`. Because of this, before calculating the days between both dates, we have to unwrap this value. In Listing 4-31, the value of the `birthday` object is unwrapped and assigned to the `olddate` variable in case of success. This date is compared to the current date in the `today` object and the number of days between both dates is returned and printed on the screen.

Printing independent components is not a problem because they are just integer values, but a complete date requires a format that will vary depending on the requirements of the program. For example, sometimes the limited space available on the screen to print the date demands expressing it just with basic values, but in other circumstances the application may require a full description, including text to describe the names of the month and the day of the week. Foundation provides the `NSDateFormatter` class to produce a textual representation of dates and times. The class includes several properties and methods to configure the object and generate the string.

dateStyle—This property sets or returns the style of the date. It is an enumeration of type `NSDateFormatterStyle` with the values `NoStyle`, `ShortStyle`, `MediumStyle`, `LongStyle`, and `FullStyle`.

timeStyle—This property sets or returns the style of the time. It is an enumeration of type `NSDateFormatterStyle` with the values `NoStyle`, `ShortStyle`, `MediumStyle`, `LongStyle`, and `FullStyle`.

dateFormat—This property sets or returns a custom format for the date and time. Its value is a string with characters that specify the position of the components of the date.

monthSymbols—This property returns an array of strings with the names of the months in correlative order (index 0 returns “January”, index 1 “February”, and so on). Other related properties are `shortMonthSymbols` (returns an abbreviation of the name) and `veryShortMonthSymbols` (returns the initials).

locale—This property sets or returns the `NSLocale` object used by the `NSDateFormatter` object to process the date. The value by default is the `NSLocale` object defined by the system.

timeZone—This property sets or returns the `NSTimeZone` object used by the `NSDateFormatter` object to process the date. The value by default is the `NSTimeZone` object defined by the system.

stringFromDate(date)—This method returns a string with the date provided by the `date` attribute in the format specified by the properties of the object. The `date` attribute is an `NSDate` object.

dateFromString(string)—This method interprets the value of the **string** attribute as a date and returns an **NSDate** object containing the value or **nil** when no date is found.

The process to format a date is simple. The **NSDateFormatter** object is created first, then the **dateStyle**, **timeStyle**, or **dateFormat** properties are initialized with the values desired, and the **stringFromDate()** method is called to get the string.

Listing 4-32: Formatting dates

```
import Foundation

var today = NSDate()

let formatter = NSDateFormatter()
formatter.dateStyle = NSDateFormatterStyle.MediumStyle
formatter.timeStyle = NSDateFormatterStyle.MediumStyle

var mydate = formatter.stringFromDate(today)
```

In the code of Listing 4-32 the style is declared as **MediumStyle** for both date and time, but we can specify other values to produce a different result. If only the date or the time are required, we can declare the style **NoStyle** for the one we want to hide or leave the property with the value by default, as in the following example.

Listing 4-33: Formatting only the date

```
import Foundation

var today = NSDate()

let formatter = NSDateFormatter()
formatter.dateStyle = NSDateFormatterStyle.FullStyle

var mydate = formatter.stringFromDate(today)
```

The **stringFromDate()** method in Listing 4-33 returns a string containing the full date without the time (the style for the time was not set). The full date includes the names of the day and the month. If we want to design our own format, we can take advantage of the **dateFormat** property.

Listing 4-34: Defining our own format

```
import Foundation
```

```
var today = NSDate()

var formatter = NSDateFormatter()
formatter.dateFormat = “(yyyy) MM dd”

var mydate = formatter.stringFromDate(today)
```

The `dateFormat` property uses specific characters to represent date components. The `stringFromDate()` method replaces those characters by the corresponding values from the date and returns the new string. In the example of Listing 4-34, we used the characters for the year (yyyy) the month (MM), and the day (dd), but there are also characters available for the hour (HH), minutes (mm), seconds (ss), and more.

Do It Yourself: Replace the code in your Playground’s file with the code of Listings 4-32, 4-33, or 4-34. Try different values and combinations for the `dateStyle`, `timeStyle`, and `dateFormat` properties to see the styles available.

IMPORTANT: The list of characters available is extensive. There are characters to represent every component of the date, and also for different configurations. For a complete list, visit our website and follow the links for this chapter.

The `NSCalendar` and `NSDateFormatter` classes process dates according to local conventions, including the language, symbols, etc. This means that the components of a date are going to be interpreted according to the conventions currently defined on the device. For example, the same date will look like this “Friday, September 18, 2016” for a user in the United States and like this “2016年9月18日 星期五” for a user in China. How dates and other relevant information are going to be processed is determined by an object of the `NSLocale` class. Every device has an object assigned by default, and the `NSCalendar` and `NSDateFormatter` objects in our code will work with this default object, unless we decide otherwise. To get a reference to the current object or create a new one, the class includes the following initializer and type method.

NSLocale(localeIdentifier: *identifier*)—This initializer creates an `NSLocale` object configured for the region determined by the value of the `identifier` attribute. The attribute is a string that represents a language and a region (e.g., `en_US` for the United States, or `zh_CN` for China).

currentLocale()—This type method returns the `NSLocale` object assigned by default to the device or defined by the user in the Settings app.

The class also offers properties and methods to get information from an `NSLocale` object or the class. The following are the most frequently used.

objectForKey(key)—This method returns an object with the information related to the key specified by the **key** attribute. The attribute is a constant defined by Foundation. There is an extensive list of constants available, with the following as the most frequently used: `NSLocaleIdentifier`, `NSLocaleLanguageCode`, `NSLocaleCountryCode`, `NSLocaleCalendar`, `NSLocaleUsesMetricSystem`, `NSLocaleDecimalSeparator`, `NSLocaleCurrencySymbol` and `NSLocaleCurrencyCode`.

availableLocaleIdentifiers()—This type method returns an array of strings with the locale identifiers available in the system.

preferredLanguages()—This type method returns an array with the languages preferred by the user.

Although it is recommended to use the `NSLocale` object set by the system and keep the values by default, there are times when our application has to present the information with a specific configuration. For example, we may need to create an application that always shows dates formatted in Chinese, no matter where the user is located. We can do this by assigning a new `NSLocale` object to the `locale` property of the `NSDateFormatter` object used to format the date.

Listing 4-35: Defining a different `NSLocale` object

```
import Foundation

let today = NSDate()
let formatter = NSDateFormatter()
formatter.dateStyle = .FullStyle

let chinaLocale = NSLocale(localeIdentifier: "zh_CN")
formatter.locale = chinaLocale

let displayDate = formatter.stringFromDate(today)
```

The code of Listing 4-35 creates a new `NSLocale` object with the `zh_CN` identifier, which corresponds to China and the Chinese language, and assigns it to an `NSDateFormatter` object. Now every string generated by this object will contain a date formatted according to Chinese conventions.

IMPORTANT: The list of identifiers you can use to create an `NSLocale` object is extensive. You can call the type method `availableLocaleIdentifiers()` on the `NSLocale` class to get an array with the values available. For a more comprehensive list and to get all the keys available for the `objectForKey()` method, visit our website and follow the links for this chapter.

The Basics: The `NSLocale` class is not only used to define date formats but also multiple linguistic and cultural norms that apply to a particular region. For this reason, the class is used to prepare the app for international distribution; a process called Internationalization. We will study Internationalization in Chapter 31.

The date in an `NSDate` object is a generic value that is actually not a date but the number of seconds between the date represented by the object and an arbitrary date in the past (January 1st, 2001). To process these values and get the actual date, the `NSCalendar` and `NSDateFormatter` classes need to know the time zone the date belongs to. Foundation includes the `NSTimeZone` class to manage time zones. An object is assigned by default to the system containing the time zone where the device is located (this is why when we display a date it coincides with the date on our device), but we can create a different one as we did with `NSLocale` objects. To get a reference to the current object or create a new one, the `NSTimeZone` class includes the following initializer and type method.

`NSTimeZone(name: name)`—This initializer creates an `NSTimeZone` object configured for the time zone determined by the value of the **`name`** attribute. The attribute is a string that represents the name of the time zone (e.g., “Europe/Paris”, “Asia/Bangkok”).

`localTimeZone()`—This type method returns the `NSTimeZone` object assigned by default to the device or defined by the user in the Settings app.

The `NSTimeZone` class also offers properties and methods to get information about the time zone and the class. The following are the most frequently used.

`name`—This property returns the name of the time zone.

`daylightSavingTime`—This property returns a Boolean value that determines whether the `NSTimeZone` object is using daylight savings time or not.

`knownTimeZoneNames()`—This type method returns an array with the names of

time zones available.

Time zones are a very complex subject, but usually the applications are quite simple. For example, we can calculate the time in different parts of the world from the same date. The following example calculates what time it is now in two different cities.

Listing 4-36: Working with different time zones

```
import Foundation

let tokyoTimeZone = NSTimeZone(name: "Asia/Tokyo")
let madridTimeZone = NSTimeZone(name: "Europe/Madrid")

let today = NSDate()
let formatter = NSDateFormatter()
formatter.dateFormat = ".FullStyle"
formatter.timeStyle = ".FullStyle"

formatter.timeZone = tokyoTimeZone
let tokyoDate = formatter.stringFromDate(today)

formatter.timeZone = madridTimeZone
let madridDate = formatter.stringFromDate(today)
```

The code of Listing 4-36 creates two `NSTimeZone` objects, one for Tokyo's time zone and another for Madrid, and then assigns the objects to an `NSDateFormatter` object to generate the strings from the current date.

IMPORTANT: The list of names for the time zones is stored in a database. You can call the type method `knownTimeZoneNames()` on the `NSTimeZone` class to get an array with the values available.

4.3 Core Graphics

Core Graphics is an old framework programmed in the C language. It was developed to provide a platform-independent two-dimensional drawing engine for Apple systems. The framework is conformed by basic drawing tools, including its own specific data types, and also a complete low-level drawing API known as Quartz 2D. Because of its characteristics, instead of being replaced, the framework was integrated with other frameworks that came along, and now it is an important piece in the construction of graphic interfaces for mobile applications.

IMPORTANT: In this chapter we are going to explore the part of Core Graphics used by modern frameworks to build user interfaces, but the framework is extensive and includes tools for the creation of custom graphics that we will study in Chapter 24.

CGGeometry

The most widely adopted feature of Core Graphics is a series of data types and functions defined under the name of **CGGeometry** (CG stands for Core Graphics). The **CGGeometry** reference includes data types to store information for geometric primitives and functions to operate on these values. They are not only used for drawing but also to position graphic elements on the screen, such as buttons, labels, etc. The following are the most frequently used.

CGFloat—This is a typealias of the **Float** type. It is used to store **Float** type values for drawing purposes.

CGSize—This is a structure that contains two properties of type **CGFloat** called **width** and **height** to represent dimensions.

CGPoint—This is a structure that contains two properties of type **CGFloat** called **x** and **y** to represent a single point.

CGRect—This is a structure that contains two properties called **origin** and **size** to represent a rectangle. The **origin** property is a structure of type **CGPoint** and the **size** property is a structure of type **CGSize**.

CGVector—This is a structure that contains two properties of type **CGFloat** called **dx** and **dy** to represent a vector (the letter **d** stands for distance).

CGRectEqualToRect(rect1, rect2)—This function returns **true** if the rectangle indicated by the **rect1** attribute is equal to the rectangle indicated by the **rect2** attribute. Both attributes are **CGRect** values.

CGRectIntersectsRect(rect1, rect2)—This function returns **true** if the rectangles indicated by the **rect1** and **rect2** attributes intersect. Both attributes are **CGRect** values.

CGRectContainsPoint(rect, point)—This function returns **true** if the point indicated by the **point** attribute is inside the rectangle indicated by the **rect** attribute. The **rect** attribute is a **CGRect** value and the **point** attribute is a **CGPoint** value.

CGRectContainsRect(rect1, rect2)—This function returns **true** if the rectangle indicated by the **rect2** attribute is inside the rectangle indicated by the **rect1** attribute, otherwise it returns **false**. Both attributes are **CGRect** values.

Besides the **origin** and **size** properties, the **CGRect** structure also includes valuable properties to get the minimum, middle, and maximum values of the rectangle.

minX—This property returns the minimum value of the rectangle's **x** coordinate.

minY—This property returns the minimum value of the rectangle's **y** coordinate.

maxX—This property returns the maximum value of the rectangle's **x** coordinate.

maxY—This property returns the maximum value of the rectangle's **y** coordinate.

midX—This property returns the value of the rectangle's **x** coordinate located at the center of the rectangle.

midY—This property returns the value of the rectangle's **y** coordinate located at the center of the rectangle.

Variables of Core Graphics types are declared and initialized as any other variables, but we have to import the Core Graphics framework first for the types to be recognized by Xcode.

Listing 4-37: Declaring Core Graphics variables

```
import CoreGraphics
```

```
var myfloat: CGFloat = 35
```

```
var mysize: CGSize = CGSize(width: 250, height: 250)
```

```
var mypoint: CGPoint = CGPoint(x: 20, y: 50)
```

```
var myrect: CGRect = CGRect(origin: mypoint, size: mysize)
```

```
var myvector: CGVector = CGVector(dx: 30, dy: 30)
```

IMPORTANT: In some examples on the web you will find that functions are used to create these structures. Their names are defined adding the word **Make** to the type's name (e.g., **CGRectMake()**). This is no longer necessary in Swift. Since most of these data types are structures, we can use their initializers instead.

The Basics: The values in a coordinate system are identified with the letters **x** and **y**. Core Graphics follows this convention and assigns these names to the parameters of the structures that store coordinate values. We will study coordinates and coordinate systems later in this chapter.

The **CGSize**, **CGPoint**, and **CGVector** structures have to be initialized with their member initializers, but the **CGRect** structure provides an additional initializer to create the instance from the values of its internal structures.

Listing 4-38: *Using a CGRect initializer*

```
import CoreGraphics

var myrect = CGRect(x: 30, y: 20, width: 100, height: 200)

print("The origin is at \(myrect.origin.x) and \(myrect.origin.y)")
print("The size is \(myrect.size.width) by \(myrect.size.height)")
```

As shown in Listing 4-38, to initialize a **CGRect** structure we can take advantage of its initializer to define the **x**, **y**, **width**, and **height** properties independently, but to read them we have to remember to access the **origin** and **size** properties. The way we access these values is by concatenating the variables and properties with dot notation. For example, the instruction **myrect.origin.x** reads the value of the **x** property that is inside the **origin** property that is inside the **myrect** variable.

The **origin** and **size** properties are **CGPoint** and **CGSize** structures, respectively, so they can be copied into other variables or properties as normal values.

Listing 4-39: *Accessing the structures inside a CGRect structure*

```
import CoreGraphics

var myrect = CGRect(x: 30, y: 20, width: 100, height: 200)

var mypoint = myrect.origin
var mysize = myrect.size
```

```
print("The origin is at \mypoint.x) and \mypoint.y)")
print("The size is \mysize.width) by \mysize.height)")
```

The framework includes three predefined structures with all its properties initialized with the value 0. This is useful when we need to create a structure but we don't know its values yet (they could be provided later by the user, for example). The names of the structures are **CGPointZero** (**CGPoint** type), **CGRectZero** (**CGRect** type), and **CGSizeZero** (**CGSize** type). We just have to assign any of these structures to our variable and use it like a normal structure.

Listing 4-40: Assigning empty structures

```
import CoreGraphics

var myrect = CGRectZero

print("The origin is at \myrect.origin.x) and \myrect.origin.y)")
print("The size is \myrect.size.width) by \myrect.size.height)")
```

The **myrect** variable of Listing 4-40 is a **CGRect** structure with all its properties initialized with the value 0. Assigning the **CGRectZero** structure is the same as using the initializer **CGRect(x: 0, y: 0, width: 0, height: 0)**.

The **CGRect** structure also includes properties to calculate values from its coordinates and size. For example, the **midX** and **midY** properties return the coordinates at the center of the axis.

Listing 4-41: Calculating the coordinate at the center of the rectangle

```
import CoreGraphics

var rect = CGRect(x: 0, y: 0, width: 100, height: 100)

print("The horizontal center is \rect.midX)") // 50.0
```

The **CGGeometry** functions are easy to implement as well. They take **CGGeometry** values and return a Boolean value to indicate if the condition is true or not. The following example creates a rectangle and a point and then checks whether the point is inside the rectangle or not with the **CGRectContainsPoint()** function.

Listing 4-42: Detecting if a point is inside a rectangle

```
import CoreGraphics
```

```
var rect = CGRect(x: 0, y: 0, width: 100, height: 100)
```

```
var point = CGPoint(x: 10, y: 50)
```

```
if CGRectContainsPoint(rect, point) {
```

```
    print("The point is inside the rectangle")
```

```
}
```

4.4 UIKit

The UIKit (User Interface Kit) is the framework that defines the elements of the graphic interface for mobile applications. From strings of text to buttons and switches all the standard elements that users manipulate on the screen to interact with the application are defined by the classes in this framework. Its primary focus is the user's interface, but it also provides elemental classes the application needs to work and connect with the rest of the system.

The application's ecosystem is a very complex mesh of elements that communicate back and forward. To check and respond to messages coming from these systems, including the operative system, the application has to keep calling itself in an infinite loop and at the same time fulfill its own purposes. These complex tasks are simplified by dividing the ecosystem into multiple parts that take control of every piece of the puzzle. This scheme is depicted in Figure 4-1.

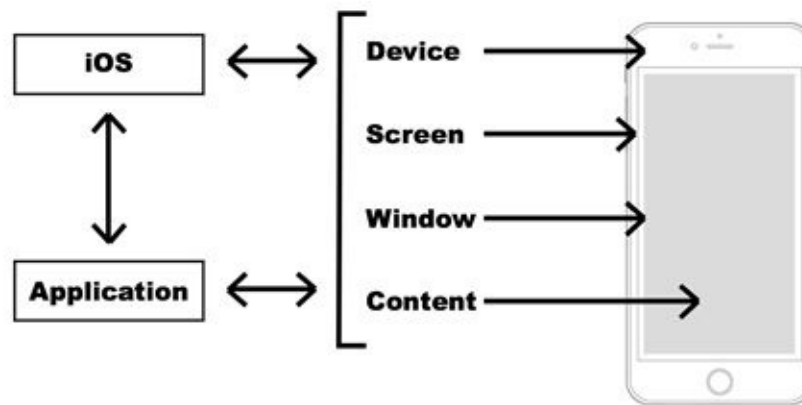


Figure 4-1: App's ecosystem

The app has to read information from the device and the screen, generate the graphics for the window and its content, and also keep the loop running to constantly check for events produced by any of these systems, such as the device informing that a rotation has taken place, a button on the screen reporting that has been tapped, or the system reporting that the app was suspended because of an incoming phone call, to name a few. This is an extremely complex process that demands a deep understanding of the technologies involved. To alleviate the work for developers and let them concentrate on what is really important for their apps, UIKit includes several classes that represent and take care of every part of this system.

UIApplication—This class is instantiated as soon as the app is executed. Its code initiates the cycle to check for events and controls the window and its content.

UIDevice—This class provides properties and methods to get information about the

device and the operative system.

UIScreen—This class provides properties and methods to get information about the main screen and the external screens connected to the device. An object is created for each screen to inform about their sizes, brightness, etc.

UIWindow—This class provides the properties and methods to manage the app's main window. An object is instantiated for each screen. The object controls the window and its content, and offers easy access to the initial view (screen).

UIApplication

When the app is built, the compiler includes basic code that is in charge of performing the first steps to initiate and transfer control of the device to our app. One of the tasks of this code is to create an object of the **UIApplication** class. This object generates a loop to process events, prepares the app for work, and also transfers the control to our own code when everything is ready. The **UIApplication** class includes a type method that we can call from our code to have access to this object.

sharedApplication()—This type method returns the instance of the **UIApplication** class that was created for our app.

Once we get a reference to the object, we can access its properties and methods. These are the most frequently used.

delegate—This property returns a reference to the object assigned as the delegate for the **UIApplication** object. As we will see later, a delegate object is assigned to the **UIApplication** object to communicate with our code. This property is often used to access data declared in the delegate from every part of our code.

windows—This property returns an array with references to the windows created for the app. If the device is not connected to an external screen, this array only contains one element referencing the window assigned to the device's main screen.

keyWindow—This property returns a reference to the active window. Its value is an optional of type **UIWindow** (it returns a **UIWindow** object or **nil** if no window was found).

applicationState—This property returns the current state of the app. It is an enumeration of type **UIApplicationState** with the values **Active** (the app is active), **Inactive** (the app is running but not active), and **Background** (the app is running in the background).

statusBarFrame—This property returns a **CGRect** structure with the values of the rectangle that defines the area occupied by the status bar.

The Basics: The status bar is the small bar displayed by mobile devices at the top of the screen. It contains basic information about the system, such as the battery level, the time, and the status of the connection. It is recommended to design the app considering the space occupied by this bar and keep it visible all the time, but we will see later that you have the option to remove it for the entire app or individual views (screens) when other information or graphics have to be shown on its place.

The **UIApplication** object is created for us in the background as soon as the application is executed, so we don't have access to this object in our code, but we can call the **sharedApplication()** method to get a reference to the current object created for our app.

Listing 4-43: *Accessing the UIApplication object*

```
import UIKit
```

```
let app = UIApplication.sharedApplication()
```

```
var state = app.applicationState
if state == UIApplicationState.Active {
    var rect = app.statusBarFrame
    print("Origin: \(rect.origin.x), \(rect.origin.y)")
    print("Size: \(rect.size.width) x \(rect.size.height)")
}
```

In Listing 4-43, a reference to the **UIApplication** object is stored in the **application** variable. From this variable we access some of its properties to check the app's state and print the size of the status bar.

One thing to notice is that we used a **CGRect** structure in this example, but there was no need to import the CoreGraphics framework. UIKit is the only basic framework we have to mention in our code; the rest, including Foundation and CoreGraphics, are imported automatically as soon as the UIKit framework is loaded.

Do It Yourself: The examples in this chapter were designed for testing on Playground. Playground was developed to learn, practice, and test code, but cannot simulate a system such as a mobile device. Therefore, some of the classes presented here will produce fake values or generate a simple graphic representation of the result. For example, the object from the **UIApplication** class is not created on

Playground and you will get empty values if you execute the code of Listing 4-43. This will only work on real applications. We will learn how to create these kinds of applications in Chapter 5.

UIDevice

The `UIDevice` class was defined to get information from the device where the app is running. The class includes a type method to get the object that represents the current device and properties to configure the object and return the data.

currentDevice()—This type method returns an instance of the `UIDevice` class that represents the device where the app is currently running.

name—This property returns a string with the name of the device. This is the name that is assigned to the device by the user.

model—This property returns a string with the model of the device (e.g., “iPod Touch”).

systemName—This property returns a string with the name of the operative system running in the device (e.g., “iPhone OS”).

systemVersion—This property returns a string with the version of the operative system running in the device (e.g., “9.0”).

userInterfaceIdiom—This property returns the type of device running the application. The value returned is an enumeration of type `UIUserInterfaceIdiom` with the values `Unspecified`, `Phone`, `Pad` and `TV`.

orientation—This property returns a value that determines the current orientation of the device. It is an enumeration of type `UIDeviceOrientation` with the values `Unknown`, `Portrait`, `PortraitUpsideDown`, `LandscapeLeft`, `LandscapeRight`, `FaceUp`, and `FaceDown`.

batteryMonitoringEnabled—This is a Boolean property that determines if the battery has to be monitored. The value `true` has to be assigned to this property if we want to check the battery level.

batteryLevel—This property returns a value between 0 and 1 to report the battery level. The `batteryMonitoringEnabled` property has to be `true` for this property to return a meaningful value.

proximityMonitoringEnabled—This is a Boolean property that determines if the proximity sensor has to be monitored. The value `true` activates the monitoring.

proximityState—This property returns a Boolean value to indicate the proximity of the user’s face to the phone. The value `true` means the sensor is close to the user. It was created to let apps know if the user is watching the screen or answering a phone call.

The values returned by some of these properties, such as the name of the device, don’t have a meaningful purpose, but some of them often help us make decisions in our code. For example, we are able to know in which position the user is holding the device from the value returned by the `orientation` property, or detect the operative system by reading the `systemVersion` property.

The implementation of this class is simple. We get an instance of the object that represents the current device first and then read its properties.

Listing 4-44: Getting information from the device

```
import UIKit

let device = UIDevice.currentDevice()

let orientation = device.orientation
switch orientation {
case .Portrait:
    print("Portrait Orientation")
case .LandscapeLeft, .LandscapeRight:
    print("Landscape Orientation")
default:
    print("Another")
}

print("Name: \(device.name) Model: \(device.model)")
print("System: \(device.systemName) \(device.systemVersion)")
```

The example of Listing 4-44 gets the information of the current device and prints it on the screen. One of the properties we read here is the `orientation` property (an enumeration type). The value of this property is checked with a `switch` statement and a message corresponding to the current value is printed along with the rest of the values.

Do It Yourself: Replace the code in your Playground’s file by the code of Listing 4-44. Again, in Playground this code produces values by default. The only way to get values that are significant to our application is to execute it on a real device. We will learn how to do it in Chapter 5.

UIScreen

The `UIScreen` object allows us to get information from the main screen and also external screens connected to the device. The class offers two type methods to get these objects and several properties to return information about the screen.

mainScreen()—This method returns a `UIScreen` object with properties and methods to manage the main screen.

screens()—This method returns an array with `UIScreen` objects that represent the main screen and any other external screen connected to the device.

bounds—This property returns a `CGRect` value with the dimensions of the screen expressed in points. The values vary according to the device's orientation.

nativeBounds—This property returns a `CGRect` value with the dimensions of the screen expressed in pixels. The values are always returned considering a portrait orientation.

scale—This property returns the scale of the screen. This is a value that translates between points and real pixels on the screen, as we will see next.

brightness—This property sets or returns the value of the brightness of the screen. It takes values from 0.0 to 1.0.

The screen of the device is conformed by a grid of hundreds of dots called *pixels*, ordered in rows and columns. The color of each pixel is established by three values contained within the pixel (four including transparency). The values represent a combination of the color red, green, and blue. By combining the pixels on the grid and the colors, the screen can display any type of graphics, including buttons, text, pictures, etc.

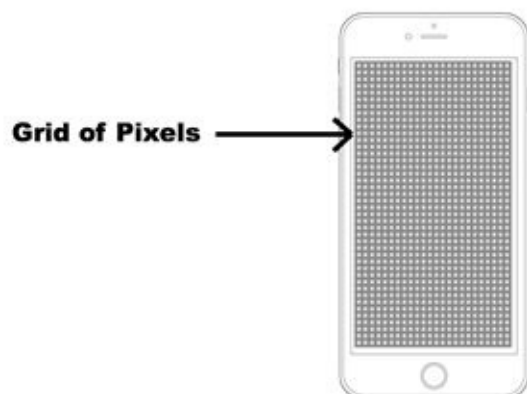


Figure 4-2: The screen is a grid of pixels

The amount of pixels varies from one device to another. Anticipating this scenario, Apple adopted a long time ago the concept of points (sometimes called *logical pixels*). The goal was to have a unit of measurement that is independent of the device and the density of the pixels on the screen. A point occupies a square of one or more pixels, but the developer doesn't need to know how many, it is all managed by the system. This way, we can program an app that shows graphics on the screen that are always the same size, independent of the amount of pixels required to draw every point.

At this moment, points in Apple mobile devices may represent a square of up to three pixels, depending on the device and the technology. Here is the list of the amount of points and the corresponding pixels for every device.

• iPhone 3 and older	320 x 480 points	320 x 480 pixels	1x scale
• iPhone 4	320 x 480 points	640 x 960 pixels	2x scale
• iPhone 5	320 x 568 points	640 x 1136 pixels	2x scale
• iPhone 6	375 x 667 points	750 x 1334 pixels	2x scale
• iPhone 6 Plus	414 x 736 points	1242 x 2208 pixels	3x scale
• iPad, iPad 2 and Mini	768 x 1024 points	768 x 1024 pixels	1x scale
• iPads with Retina	768 x 1024 points	1536 x 2048 pixels	2x scale
• iPad Pro	1366 x 1024 points	2732 x 2048 pixels	2x scale

The difference between points and pixels represents the scale. iPhones 3 and older have a scale of 1 (1 point represents 1 pixel), while iPhone 4, 5, and 6 have a scale of 2 (1 point represents a square of 2 pixels). Some iPhones and every iPad share a grid with the same amount of points, independently of the scale, but with the introduction of the iPhone 6 Plus and its larger screen, the scale was expanded to 3 (1 point represents 3 pixels). This allows developers to create one graphic interface that adapts automatically to every device, no matter how many pixels their screens have.

IMPORTANT: Larger screens produced an increment in the size of the grid of points for iPhones. The advantage of using points was diminished by the many configurations now available. This forced Apple to introduce the concept of *Adaptivity*. We will study Adaptivity in Chapter 6.

The properties `bound`, `nativeBounds`, and `scale` included in the `UIScreen` class return the values

corresponding to the screen of the device where the app is running. The `bound` and `nativeBounds` properties contain `CRect` values with their `size` and `origin` structures inside, while `scale` is just a `CGFloat` value.

Listing 4-45: Getting information from the screen

```
import UIKit
```

```
let screen = UIScreen.mainScreen()
```

```
let pointsWidth = screen.bounds.size.width
```

```
let pointsHeight = screen.bounds.size.height
```

```
print("Width: \(pointsWidth) x Height: \(pointsHeight)")
```

```
let pixelsWidth = screen.nativeBounds.size.width
```

```
let pixelsHeight = screen.nativeBounds.size.height
```

```
print("Width: \(pixelsWidth) x Height: \(pixelsHeight)")
```

```
print("Scale: \(screen.scale)")
```

The native resolution in pixels is always returned considering a portrait orientation, but the size in points is returned according to the current orientation. For example, the dimensions for the iPhone 6's screen are 375 points wide and 667 points tall in portrait, but 667 points wide and 375 points tall in landscape.

Do It Yourself: Replace the code in your Playground's file by the code in Listing 4-45. Playground can't simulate a mobile device but it shows some default values for every property.

UIWindow

Every app generates windows to visually communicate with their users. Desktop applications include several windows, but mobile devices, due to the limited size of their screens, only have one. This window is like a virtual space of the size of the screen, where the visual elements of the app are laid out. UIKit includes the `UIWindow` class to create the object that manages the window for our app. Usually, this object is automatically created as soon as the app is executed, but we can access it from our code reading the `windows` or `keywindow` properties of the `UIApplication` object.

Because of the especial characteristics of mobile devices, once the window is created it never changes. If the graphics or the interface has to be modified, the content of the window is the one replaced, but the window remains the same. For this reason, we barely

interact with this object. The following are the most frequently used properties.

screen—This property returns a `UIScreen` object that represents the device’s screen. This is the same object returned by the `mainScreen()` method of the `UIScreen` class.

frame—This property returns a `CGRect` value that determines the position and size of the rectangular area occupied by the window.

rootViewController—This property returns a reference to the object controlling the app’s initial view (screen).

Listing 4-46: Accessing the `UIWindow` object

```
import UIKit

let app = UIApplication.sharedApplication()

let window = app.keyWindow
if let windowFrame = window?.frame {
    print("Size: \(windowFrame.size.width) x \(windowFrame.size.height)")
}
```

This example gets a reference to the `UIWindow` object created for our app and prints the window’s size on the Results Side Bar. The `keyWindow` property returns an optional, so we use Optional Chaining to read the `frame` property of the `window` object and Optional Binding to assign the value returned by this property to the `windowFrame` constant (see Chapter 3 for more information on Optional Chaining and Optional Binding).

IMPORTANT: The `rootViewController` property stores a reference to an object that contains the initial view of the interface and is in charge of managing that view. These objects are called *View Controllers*. We will study views next and View Controllers in Chapter 5.

UIView

The window, as we just mentioned, is the space where the graphics are displayed, but it doesn’t generate any visible content by itself. The user’s interface is actually built inside the window from similar containers called *views*. These views are rectangular areas of custom size, designed to display graphics on the screen. Some views are used just as containers while others are improved to present graphic tools, such as buttons and

switches, and also graphic content such as images and text. The views are organized in a hierarchy, one inside another, with a view of the size of the window as the root (usually called *main view* or *container view*).

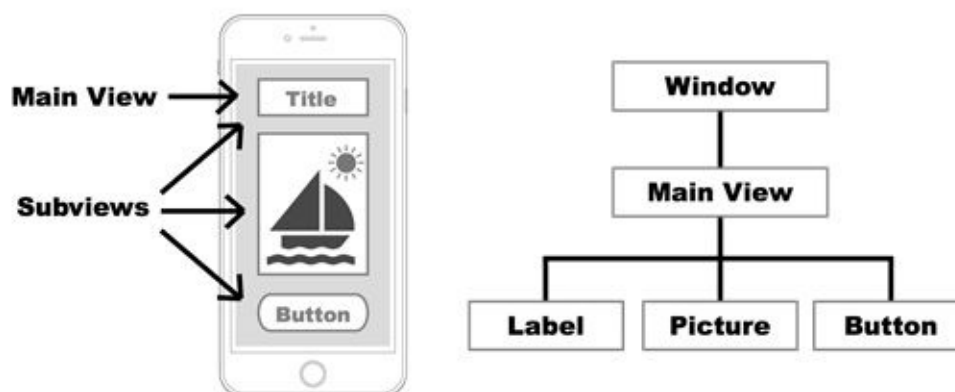


Figure 4-3: Views hierarchy

No matter the purpose of the view (container or content), they are all created from objects of the `UIView` class. This is a basic class that is only capable of creating and managing the rectangular area occupied by the view, but subclasses of this class may be defined to add functionality and present any type of content we want. Although the objects created directly from the `UIView` class are very limited, they have an extensive list of properties and methods to configure and draw their views. The following are the most useful initializer, properties, and methods of this class.

`UIView(frame: frame)`—This initializer creates a `UIView` object in the position and size determined by the **frame** attribute. The attribute is a value of type `CGRect`.

frame—This property sets or returns a `CGRect` value that determines the position and size of the rectangular area occupied by the view. The values for the position are determined from the coordinates of the view's container (the window or another view).

bounds—This property sets or returns a `CGRect` value that determines the position and size of the rectangular area occupied by the view inside its own frame. The values for the position are determined from the view's coordinates system.

center—This property sets or returns a `CGPoint` value that determines the coordinates of the center of the view's frame.

backgroundColor—This property sets or returns an object that determines the color of the view's background. The value is an object of the `UIColor` class (studied next).

alpha—This property sets or returns a `CGFloat` value that determines the view's alpha

level (the level of transparency). The property takes values from 0.0 (transparent) to 1.0 (opaque).

hidden—This property sets or returns a Boolean value that determines whether the view is visible (**false**) or hidden (**true**).

opaque—This property sets or returns a Boolean value that determines whether the view is opaque or not.

contentMode—This property sets or returns the mode used by the view to lay out its content when its size changes. It is an enumeration of type **UIViewContentMode** with the values **ScaleToFill**, **ScaleAspectFit**, **ScaleAspectFill**, **Redraw**, **Center**, **Top**, **Bottom**, **Left**, **Right**, **TopLeft**, **TopRight**, **BottomLeft**, and **BottomRight**. It is useful when the view contains an image, as we will see later.

clipsToBounds—This property sets or returns a Boolean value that determines whether or not the content of the view is confined to the view's bounds.

userInteractionEnabled—This property sets or returns a Boolean value that determines whether or not the view responds to the user's interaction (e.g., a tap of the finger).

multipleTouchEnabled—This property sets or returns a Boolean value that determines if the view can handle multiple touch events.

exclusiveTouch—This property sets or returns a Boolean value that determines whether or not the view delivers to other views the touch events that can't handle.

superview—This property returns a reference to the **UIView** object that is the container of the view. If the view is not inside another view, the value returned is **nil**.

subviews—This property returns an array containing references to the **UIView** objects that are inside of the view (subviews).

window—This property returns a reference to the **UIWindow** object that the view belongs to. This is the same object returned by the **windows** or **keywindow** properties of the **UIApplication** object.

tag—This property sets or returns an integer value that identifies the view.

viewWithTag(tag)—This method searches for a view in the **subviews** array with the tag specified by the **tag** attribute. The value returned is an optional containing a **UIView** object or **nil** in case of failure.

The window is positioned inside the screen, while the views are positioned inside the window and inside one another, generating a hierarchical structure. To establish the position of every element in its container, UIKit uses a coordinate system. The next figure

illustrates this system considering the screen of an iPhone 6 (375 points wide by 667 points tall).

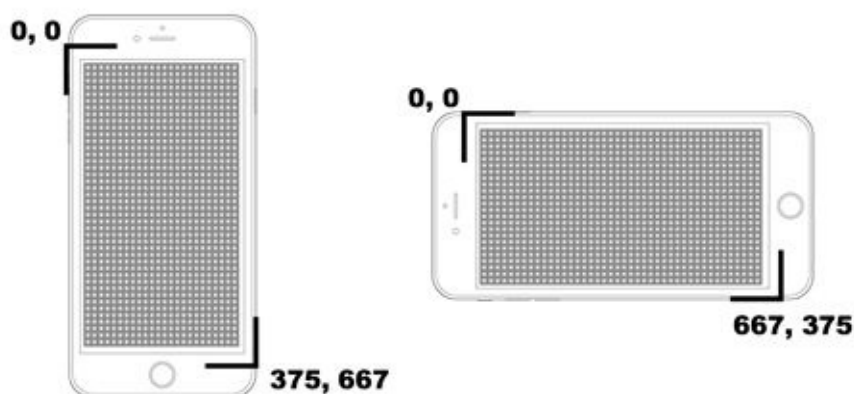


Figure 4-4: *Coordinate system*

The rows and columns of points are counted from the top-left corner to the bottom-right corner of the container (the values are incremented from left to right and top to bottom). For example, the first point at the top-left corner of the screen is at the position 0, 0 (column 0, row 0), while the point at the bottom-right corner is at the position 375, 667 (667, 375 in landscape orientation). Following standard conventions, the point at the position 0, 0 is called the *origin*, and the values for the positions are identified by the letters **x** and **y** (**x** for columns and **y** for rows).

In UIKit, this same coordinate system is applied to every element of the interface, including the window and the views. The window and the main view occupy the whole screen, and therefore they share its same origin and size. The next example creates a view of 375 points by 667 points to represent the main view that might be contained by a window of an iPhone 6.

Listing 4-47: *Creating the main view*

```
import UIKit

var mainframe = CGRect(x: 0, y: 0, width: 375, height: 667)
let container = UIView(frame: mainframe)
```

The initializer of the `UIView` class takes a parameter called `frame` of type `CGRect` to establish the position and size of the view. The view created by the code of Listing 4-47 will be positioned at the coordinates 0, 0 of the window and will have a size of 375 by 667 points, covering the whole screen of an iPhone 6.

Do It Yourself: Replace the code in your Playground's file by the code of Listing 4-

47. Click on the Show Result button corresponding to the line that creates the **container** object to see a representation of the view in between your code. At this moment you will only be able to see a white square. We will add some color next.

IMPORTANT: As we mentioned before, Playground offers the Show Result button on the Results Side Bar to get a graphic representation of the result produced by the code. This is only a reference of what the graphics will look like on a real screen, but it is handy for practice and testing. To add a Result window to your code, press the Show Result button (Figure 2-2) of the line of code you want to see. You can add to your code all the windows you need to represent the changes of the values over time.

The background color of the **UIView** object is set as transparent by default. If we want to change the color, we have to assign a new value to the **backgroundColor** property of the view. This value has to be provided as an object of the **UIColor** class. The class includes some initializers and type methods to create these objects.

UIColor(red: **color1, green: **color2**, blue: **color3**, alpha: **value**)**—This initializer creates a **UIColor** object with a color set by the values of its attributes. It takes **CGFloat** values from 0.0 to 1.0. The **color1** attribute defines the level of red, **color2** the level of green, **color3** the level of blue, and **value** defines the alpha level or transparency for the color.

UIColor(hue: **value1, saturation: **value2**, brightness: **value3**, alpha: **value4**)**—This initializer creates a **UIColor** object with a color set by the values of its attributes. It takes **CGFloat** values from 0.0 to 1.0. The **value1** attribute defines the hue component, **value2** the saturation, **value3** the brightness, and **value4** the alpha or transparency.

UIColor(patternImage: **image)**—This initializer creates a **UIColor** object with a color defined as the image provided by the **image** attribute. The image will be replicated as many times as necessary to cover the area where the color is assigned. The **image** attribute is a **UIImage** object (we will study **UIImage** in Chapter 8).

blackColor()—This type method returns a **UIColor** object with the color black.

grayColor()—This type method returns a **UIColor** object with the color gray.

darkGrayColor()—This type method returns a **UIColor** object with a dark gray color.

lightGrayColor()—This type method returns a **UIColor** object with a light gray color.

whiteColor()—This type method returns a **UIColor** object with the color white.

redColor()—This type method returns a **UIColor** object with the color red.

greenColor()—This type method returns a `UIColor` object with the color green.

blueColor()—This type method returns a `UIColor` object with the color blue.

cyanColor()—This type method returns a `UIColor` object with the color cyan.

yellowColor()—This type method returns a `UIColor` object with the color yellow.

magentaColor()—This type method returns a `UIColor` object with the color magenta.

orangeColor()—This type method returns a `UIColor` object with the color orange.

purpleColor()—This type method returns a `UIColor` object with the color purple.

brownColor()—This type method returns a `UIColor` object with the color brown.

The `UIColor` objects are assigned to the views and almost any other element of the user interface to define their colors. They are created from the initializers or type methods listed above and then assigned to the corresponding view's property.

Listing 4-48: Assigning a color to the main view

```
import UIKit

let container = UIView(frame: CGRect(x: 0, y: 0, width: 375, height: 667))

var color = UIColor(red: 1.0, green: 0.0, blue: 0.0, alpha: 1.0)
container.backgroundColor = color
```

The color for the `UIColor` object created in Listing 4-48 is defined by the attributes of the initializer. The system used in this case is called RGB. The color is constructed by adding the values of every component: Red, Green, and Blue. In this example, the `red` attribute was set to 1.0 while the rest of the colors were set to 0.0, getting a pure red. Exactly the same object is returned by the type method `redColor()`.

Listing 4-49: Assigning the color red to the main view

```
import UIKit

let container = UIView(frame: CGRect(x: 0, y: 0, width: 375, height: 667))

var color = UIColor.redColor()
container.backgroundColor = color
```

Do It Yourself: Replace the code in your Playground's file by the code of Listing 4-49. Click on the Show Result button corresponding to the line that assigns the new

color to the `backgroundColor` property of the view. You should see a rectangular red view in between your code.

The Basics: The `UIColor` initializer uses values from 0.0 to 1.0 to determine the levels of red, green, and blue that compose the color, but RGB colors are usually determined with integer values from 0 to 255. If you prefer to work with these values, you can convert them to the values required by the initializer by dividing the values by 255.0. For example, if the level of red is 190, you can get the value for the initializer with the formula $190.0 / 255.0$.

The user's interface is built by adding views to parent views that work as containers, creating the hierarchical structure we mentioned before. The container view is called the *superview* and the views inside are called *subviews*. When a view is added to a superview, the object is stored in the `subviews` property of the superview. This is a property containing an array of views (objects of the `UIView` class or any of its subclasses). To add and manage the views inside this array, the `UIView` class offers the following methods.

`addSubview(view)`—This method adds a view at the end of the `subviews` array.

`insertSubview(view, atIndex: index)`—This method inserts a view in the `subviews` array at the index indicated by the `index` attribute.

`insertSubview(view, aboveSubview: subview)`—This method inserts a view in the `subviews` array after the subview indicated by the `subview` attribute.

`insertSubview(view, belowSubview: subview)`—This method inserts a view in the `subviews` array before the subview indicated by the `subview` attribute.

`exchangeSubviewAtIndex(index1, withSubviewAtIndex: index2)`—This method exchanges the positions in the `subviews` array of the subviews at the indices indicated by the `index1` and `index2` attributes.

`bringSubviewToFront(subview)`—This method moves the subview indicated by the `subview` attribute to the end of the `subviews` array.

`sendSubviewToBack(subview)`—This method moves the subview indicated by the `subview` attribute to the beginning of the `subviews` array.

`removeFromSuperview()`—This method is called on a subview to remove it from its superview.

The subviews are shown on the screen according to their position on the `subviews` array. The subviews with a lower index are drawn first, so the views with a higher index are

shown at the top. This is why the method most frequently used to add a view to another view is `addSubview()`. This method adds the view to the end of the `subviews` array, effectively drawing the new view at the top.

Listing 4-50: Adding subviews

```
import UIKit

let container = UIView(frame: CGRect(x: 0, y: 0, width: 375, height: 667))
container.backgroundColor = UIColor.redColor()

let subview1 = UIView(frame: CGRect(x: 20, y: 20, width: 335, height: 300))
subview1.backgroundColor = UIColor.lightGrayColor()

let subview2 = UIView(frame: CGRect(x: 20, y: 347, width: 335, height: 300))
subview2.backgroundColor = UIColor.lightGrayColor()

container.addSubview(subview1)
container.addSubview(subview2)
```

After creating the `container` view, the code of Listing 4-50 creates two more views and adds them as its subviews with the `addSubview()` method. The `subviews` property of `container` now has two elements: the `subview1` and `subview2` views. These subviews are of different color, creating two smaller rectangles of 335 by 300 points over the main view. The result is shown in Figure 4-5.

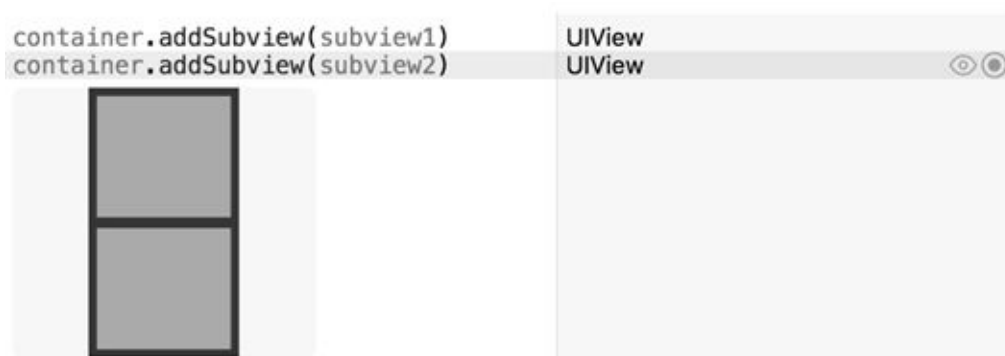


Figure 4-5: Views on Playground

Every subview is attached to its superview and positioned on the screen according to its coordinate system. When we set the values for the `x` and `y` parameters in the `UIView` initializer for every view, we have to consider the coordinates of their superviews, not the screen or other container views up the chain. For example, the position of the `subview1` inside the `container` view is 20, 20, but its own coordinate system starts again from 0, 0 at its top-left corner. The same happens with `subview2`. Any views added to these subviews will

have a position according to the coordinates of their superviews and not the coordinates of the **container** view or the window. Figure 4-6 illustrates these views and their coordinate systems.

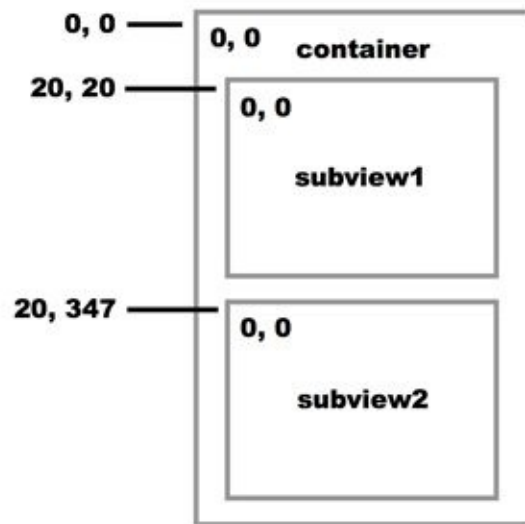


Figure 4-6: Subviews and their coordinates

These values are reflected in the **frame** and **bounds** properties of the **UIView** objects that represent each view. For example, the **x** and **y** values of the **frame** property of **subview1** are 20, 20, but the **x** and **y** values of the **bounds** property of **subview1** are 0, 0, because **bounds** refers to the internal frame.

UIView Subclasses

The views created from the **UIView** class are effective containers and organizers but don't present any content. The content has to be generated by subclasses of **UIView**. These subclasses overwrite some **UIView** methods and also provide their own properties and methods to draw graphics inside the view or respond to user's interaction. Considering how difficult and time consuming it is to create every single element of the interface from scratch, **UIKit** provides ready to use subclasses for the creation of standard elements. From labels and buttons to images and tables, there is a subclass to present on the screen everything our app needs. Figure 4-7 shows a scheme with the **UIView** class and its most important subclasses.

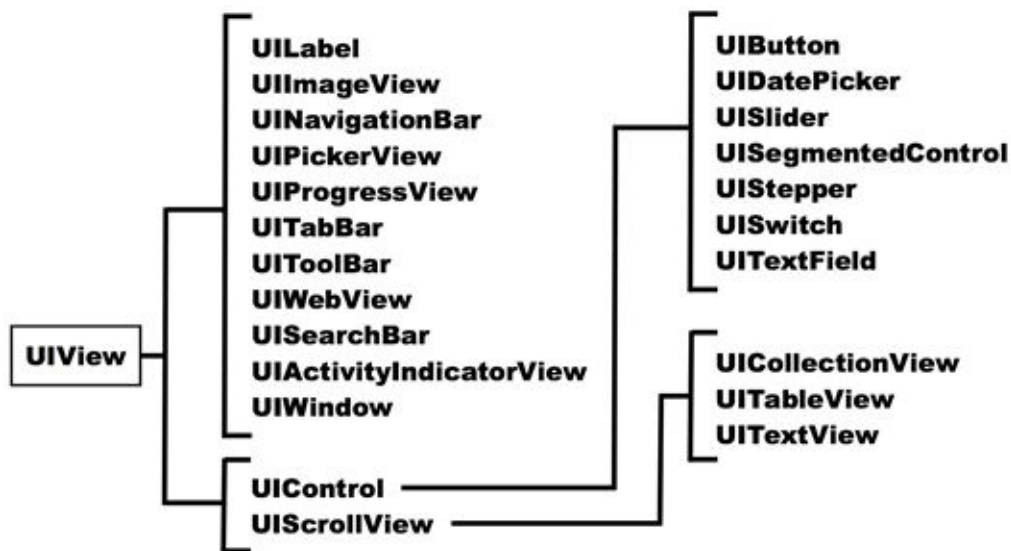


Figure 4-7: *UIView subclasses*

Some classes that add visual content and process basic user interaction, such as **UILabel** or **UIImageView**, inherit directly from **UIView**, while classes that require more complex functionality, such as **UIButton** or **UITableView**, inherit from intermediate subclasses. The two intermediate subclasses depicted in Figure 4-7, **UIControl** and **UIScrollView**, provide common functionality for further subclasses. **UIControl** adds to **UIView** the capacity to respond to user interaction and report events, and **UIScrollView** adds the possibility to create views with scrollable content. The subclasses created from these two classes inherit their characteristics and add more specific functionality. All these subclasses are extremely useful for app development. We will study some of the simple **UIView** subclasses next and explore more advanced subclasses in further chapters.

The Basics: In the scheme presented in Figure 4-7 you can see that the **UIWindow** class studied before is actually a subclass of **UIView**. This is how the class inherits the characteristics and functionality of views, such as the capacity to contain other views and essential properties like **frame** (this property is inherited by **UIWindow** from the **UIView** class).

UILabel

The **UILabel** class creates a view that draws one or multiple lines of text that can be used to present information or to identify other elements on the interface. The following are some of the properties defined by the class to configure the view and format the text.

text—This property sets or returns the text displayed by the label.

attributedText—This property sets or returns the formatted text displayed by the label. Its value is an object of the `NSAttributedString` class.

font—This property sets or returns the font used to display the label. Its value is an object of the `UIFont` class.

textColor—This property sets or returns the color of the text. Its value is an object of the `UIColor` class.

textAlignment—This property sets or returns the alignment of the text. It is an enumeration of type `NSTextAlignment` with the values `Left` (to the left side of the view), `Center` (to the center of the view), `Right` (to the right side of the view), `Justified` (the last line of the paragraph is aligned), and `Natural` (uses the alignment associated with the text).

lineBreakMode—This property sets or returns the mode used to display lines of text that get out of the boundaries of the view. It is an enumeration of type `NSLineBreakMode` with the values `ByWordWrapping` (the word that doesn't fit is moved to the next line), `ByCharWrapping` (the character that doesn't fit is moved to the next line), `ByClipping` (characters that doesn't fit are not drawn), `ByTruncatingHead` (the beginning of the text is replaced by ellipsis), `ByTruncatingTail` (the end of the text is replaced by ellipsis), `ByTruncatingMiddle` (the middle of the text is replaced by ellipsis).

numberOfLines—This property sets or returns the number of lines allowed for the text. If the text requires more lines than the number set by this property, it is truncated according to the mode selected by the `lineBreakMode` property. A value of 0 for this property declares a label of unlimited lines of text.

adjustFontSizeToFitWidth—This property sets or returns a Boolean value indicating whether or not the size of the font has to be reduced to fit the text inside the view. It only works when the `numberOfLines` property is set to 1.

enabled—This property sets or returns a Boolean value to indicate if the text is enabled. When the value of this property is set to `false`, the text is shown on the screen in a light color, usually light gray.

shadowColor—This property sets or returns the color of the shadow. It is an optional property of type `UIColor`. When its value is `nil`, the text is drawn with no shadow.

shadowOffset—This property sets or returns the offset of the shadow. Its value is a `CGSize` structure that establishes the vertical and horizontal displacement.

The content produced by `UILabel`, as any other subclass of `UIView`, is contained inside a view. Therefore, we have to define the position and size of the view when creating a `UILabel` object and then assign values to its properties to define the content. The class uses the same initializer of `UIView` to provide the information for the view.

Listing 4-51: Creating a label

```
import UIKit

let container = UIView(frame: CGRect(x: 0, y: 0, width: 375, height: 667))
container.backgroundColor = UIColor.lightGrayColor()

let label = UILabel(frame: CGRect(x: 20, y: 20, width: 150, height: 30))
label.text = "Hello World"

container.addSubview(label)
```

Labels, as any other graphic elements generated from subclasses of `UIView`, are inserted in container views to create the user's interface. In Listing 4-51, the container is created first and then a label is added as one of its subviews by the `addSubview()` method. The text for this label is assigned to the `text` property of the `UILabel` object as soon as the object is created. This is an optional property and its value by default is `nil`. This time, we assigned the value before the label view was added to the container, but we could have done it any time. The system updates the views on the screen over and over again after a few milliseconds, so the view will reflect the change right away every time a new value is assigned to this property.

The label is positioned in the container at the point 20, 20 and has a size of 150 points wide and 30 points tall. These are the measures of the rectangular area occupied by the view, not the size of the text. The size of the text is determined by the font type set by default in the system. If we want to set a different size or font type, we have to assign a new `UIFont` object to the `font` property of the `UILabel` object. `UIFont` objects are not only assigned to labels but also to other elements that present text on the screen. They contain properties and methods to store and manage fonts. The following are the most frequently used initializer and type methods included in the `UIFont` class to create these objects.

`UIFont(name: name, size: value)`—This initializer creates a `UIFont` object with the font referenced by the **name** attribute and a size determined by the **value** attribute. The **name** attribute is a string with the PostScript name of the font, and the **value** attribute is a `CGFloat` value.

`preferredFontForTextStyle(style)`—This type method returns a `UIFont` object with the font associated to the style specified by the **style** attribute and a size specified by the user in Settings. The **style** attribute is a constant that specifies a predefined style. The constants available are `UIFontTextStyleTitle1`, `UIFontTextStyleTitle2`, `UIFontTextStyleTitle3`, `UIFontTextStyleHeadline`, `UIFontTextStyleSubheadline`, `UIFontTextStyleBody`, `UIFontTextStyleFootnote`, `UIFontTextStyleCaption1`, `UIFontTextStyleCaption2` and `UIFontTextStyleCallout`.

systemFontSize(value)—This type method returns a **UIFont** object with the font set by default on the system, and a size determined by the **value** attribute (a **CGFloat** value).

boldSystemFontSize(value)—This type method returns a **UIFont** object with the font of type bold set by default on the system, and a size determined by the **value** attribute (a **CGFloat** value).

italicSystemFontSize(value)—This type method returns a **UIFont** object with the font of type italic set by default on the system, and a size determined by the **value** attribute (a **CGFloat** value).

labelFontSize()—This type method returns a **CGFloat** value with the size of the font used by the system on **UILabel** objects.

buttonFontSize()—This type method returns a **CGFloat** value with the size of the font used by the system on **UIButton** objects.

systemFontSize()—This type method returns a **CGFloat** value with the size of the font used by the system.

Although we can create a font object with the initializer or any of the methods above, Apple recommends to work only with the **preferredFontForTextStyle()** method. The fonts returned by this method are called *Dynamic Types* because they are selected by the system according to the style specified by the attribute and the size set by the user. The Settings app offers an option to change the font size for all the applications. If our application adopts dynamic types, it can respond to these changes and show the text on the screen with the size preferred by the user (the option is available in General / Accessibility / Large Text).

To take advantage of dynamic types we have to get a **UIFont** object for the style we need and then assign it to the label's **font** property.

Listing 4-52: Creating dynamic font types

```
import UIKit

var myfont = UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline)

var label = UILabel(frame: CGRect(x: 20, y: 20, width: 150, height: 50))
label.text = "Hello World"
label.font = myfont
```

The code of Listing 4-52 creates a new **UILabel** object with the text “Hello World” and a font determined by the **UIFontTextStyleHeadline** style and the size determined by the user.

The Basics: The example of Listing 4-52 doesn't add the label to a container view. Playground lets you check the result of executing your code independently of the context. When testing code on Playground, you can click on the Show Result button to see a representation of the views and their content on the screen without the need to create a proper structure. In a real application though, every view has to be added as a subview of another view.

Do It Yourself: The `UIFontTextStyleHeadline` style is the style used for titles and headers, but you can define any style you want depending on the purpose of the text. For example, for long texts and content the preferred style is `UIFontTextStyleBody`. Every style uses different font types and therefore they look different. Try different styles and click on the Show Result button on the last line of code to see the fonts they produce.

If we don't want to use dynamic types, we can create the font object with a specific family type and size using the `UIFont` initializer. The following example creates the `UIFont` object with a font called *Georgia-Italic* and a size of 30 points. Notice that because the size of this font is bigger than a normal font, we incremented the size of the label's frame to fit the text.

Listing 4-53: Creating a specific font

```
import UIKit

var myfont = UIFont(name: "Georgia-Italic", size: 30)

var label = UILabel(frame: CGRect(x: 20, y: 20, width: 300, height: 50))
label.text = "Hello World"
label.font = myfont
```

IMPORTANT: We always have to make sure that the size of the label's view is big enough to fit all the text or at least that the text won't be truncated in a way that becomes unintelligible for the user. In Chapter 6 we will study ways to set the system to ignore the width and height and expand the view automatically.

The Basics: The font names for the `UIFont` initializer are the PostScript names of the fonts. To find these names, open the Font Book application from your Applications folder, go to the View menu and select the option Show Font Info. On the right column of the window you will see the information for the selected font, starting by its PostScript name. Select the font you would like to include in your app to get its

name. Most of the font types in your computer are available in iOS, but you can also add your own fonts to the project. For more information, visit our website and follow the links for this chapter.

We can also use fonts set by default on the system that were carefully selected to optimize the design and produce a pleasant experience for the user. The `UIFont` class includes several methods to get an object with these settings.

Listing 4-54: Using the system’s standard font

```
import UIKit

var myfont = UIFont.systemFontSize(30)

var label = UILabel(frame: CGRect(x: 20, y: 20, width: 300, height: 50))
label.text = “Hello World”
label.font = myfont
```

The `systemFontSize()` method applied in Listing 4-54 returns a `UIFont` object with the standard system font but the size specified in its attribute. This time we declared a fix value of 30, but we can take advantage of some of the properties of the `UIFont` class to set the size as some of the default values defined by the system.

Listing 4-55: Using the font size used by default for buttons

```
import UIKit

var size = UIFont.buttonFontSize()
var myfont = UIFont.systemFontSize(size)

var label = UILabel(frame: CGRect(x: 20, y: 20, width: 300, height: 50))
label.text = “Hello World”
label.font = myfont
```

This last example calls the `buttonFontSize()` method to get the size by default of the font used by the system to write the text for buttons, and assign it as the size of the `myfont` object. This is helpful when we want to present text along with some buttons and we have to keep everything in proportion (we will study buttons later).

So far we have been working with short texts and making sure that the size of the label’s view is big enough to display the whole text on the screen, no matter its size. This is not always possible. Sometimes the size of the label’s view is permanent or there is not enough room on the interface to expand, but the text drawn inside the view may change.

We have to make sure that the system formats this text in a way that it remains readable and useful for our users all the time. The `UILabel` class provides the `lineBreakMode` property to establish how the text is going to be processed when it is longer than the size of its view. The value by default is `ByTruncatingTail`, which means the characters at the end of the text that don't fit inside the view are going to be replaced by ellipsis. Other frequent values are `ByTruncatingMiddle`, to truncate the middle of the text, or `ByClipping`, to hide the characters that exceed the boundaries of the view.

Listing 4-56: Clipping the label's text

```
import UIKit

var myfont = UIFont.systemFontOfSize(30)

var label = UILabel(frame: CGRect(x: 20, y: 20, width: 150, height: 30))
label.text = "This is a Beautiful Life"
label.lineBreakMode = NSLineBreakMode.ByClipping
```

In the code of Listing 4-56, the values for the label's view are restored to the ones we used at the beginning and a longer text is assigned to the `text` property to see the effects of the line break mode applied to the label.

Do It Yourself: Replace the code on your Playground's file by the code of Listing 4-56. Click the Show Result button corresponding to the last statement to see the label in the editor. Try every possible value for the `lineBreakMode` property to see how they work. Remember that Swift infers the type of value from the type of the property and you don't have to write the name of the structure every time a new value is assigned (e.g., `label.lineBreakMode = .ByClipping`).

The `UILabel` class includes more properties to customize the text, including changing its color by assigning a `UIColor` object to the `textColor` property, or even generating a shadow with the `shadowColor` and `shadowOffset` properties.

Listing 4-57: Customizing the label's text

```
import UIKit

var color = UIColor.blueColor()
var shadow = UIColor.lightGrayColor()
var myfont = UIFont.systemFontOfSize(30)

var label = UILabel(frame: CGRect(x: 20, y: 20, width: 150, height: 30))
```

```
label.text = "Hello World"
```

```
label.textColor = color
```

```
label.shadowColor = shadow
```

```
label.shadowOffset = CGSize(width: 2, height: 2)
```

These properties assign the styles to the whole text. Every time a new font, size or color is assigned to the `UILabel` object, all the characters are affected by the new values of these properties. If we want some parts of the text to have a different style, we have to create a separate label or assign the text to the `attributedText` property instead. This property takes a value of type `NSAttributedString`. The `NSAttributedString` class is a subclass of the `NSString` class defined to manage text with attributes (styles) that are assigned to an individual character or a range of characters. The `NSAttributedString` class includes the properties and methods to store an attributed string and return information about it, but it is through its `NSMutableAttributedString` subclass that we can assign attributes to a selected range of characters. The following list includes the initializer, properties and methods to work with this subclass.

`NSMutableAttributedString(string: value)`—This initializer creates an `NSMutableAttributedString` object with the text specified by the **value** attribute.

`string`—This property returns a `String` value with the text managed by the `NSMutableAttributedString` object.

`length`—This property returns the number of characters in the string.

`setAttributes(attributes, range: range)`—This method sets the attributes for a range of characters. The previous attributes are replaced. The **attributes** attribute is a dictionary containing constants that describe the attribute and values with the corresponding styles (font, color, etc.). The **range** attribute is an `NSRange` object defining the range of characters affected.

`addAttributes(attributes, range: range)`—This method adds attributes to a range of characters. The new attributes are added to the attributes already assigned to the characters in the range. The **attributes** attribute is a dictionary containing constants that describe the attribute and values with the corresponding styles (font, color, etc.). The **range** attribute is an `NSRange` object defining the range of characters affected.

`addAttribute(name, value: value, range: range)`—This method adds a single attribute to a range of characters. The new attribute is added to the attributes already assigned to the characters in the range. The **name** attribute is the constant that describes the attribute, the **value** attribute is the value of the corresponding style (font, color, etc.),

and the **range** attribute is an `NSRange` object defining the range of characters affected.

removeAttribute(name, range: range)—This method removes an attribute previously assigned to a range of characters. The **name** attribute is the constant that describes the attribute, and the **range** attribute is an `NSRange` object defining the range of characters affected.

The Basics: Unlike Swift, Objective-C doesn't have variables and constants. This condition is established by the definition of mutable and immutable classes. These are classes that determine if the values can be changed or not. Mutable classes are usually subclasses of the immutable ones, like in the case of the `NSAttributedString` class and its subclass `NSMutableAttributedString`.

The attributes are assigned to an attributed text by predefined constants. The list of constants available is extensive, with the most frequently used being `NSFontAttributeName` (font), `NSForegroundColorAttributeName` (color of the text), `NSStrokeColorAttributeName` (color of the stroke), `NSStrokeWidthAttributeName` (width of the stroke), and `NSShadowAttributeName` (shadow).

The first step we have to perform to show attributed text on the screen is to create the `NSMutableAttributedString` object, then assign the attributes from the object's methods, and finally assign the object to the label's `attributedText` property. One of the methods used to add attributes is `addAttribute()`.

***Listing 4-58:** Adding attributes one by one to a range of characters*

```
import UIKit

let myfont = UIFont.systemFontOfSize(30)
let myrange = NSRange(location: 0, length: 5)

var mytext = NSMutableAttributedString(string: "Hello World")
mytext.addAttribute(NSFontAttributeName, value: myfont, range: myrange)

var label = UILabel(frame: CGRect(x: 20, y: 20, width: 150, height: 30))
label.attributedText = mytext
```

The `addAttribute()` method takes a constant, the value for that constant, and an `NSRange` object to determine the range of characters we want to affect. In Listing 4-58 we assign a system font of size 30 to the first five characters of the string (the word "Hello").

Most of the constants require values of types we already know, such as `UIFont` (applied in the previous example) or `UIColor`, but some require more specific types. For example, the `NSShadowAttributeName` requires an object of the `NSShadow` class. This is a simple class

with a few properties to configure the characteristics of a shadow.

shadowColor—This property defines the color of the shadow. It takes a value of type `UIColor`.

shadowOffset—This property defines the offset of the shadow. It takes a value of type `CGSize` to determine the horizontal and vertical displacement.

shadowBlurRadius—This property defines the blur effect for the shadow. It takes a value of type `CGFloat`.

Effects like shadows are more noticeable when applied to big fonts. This requires us to assign multiple attributes to the same string. Although we can add all the attributes we need one by one with the `addAttribute()` method, the class offers a better method called `setAttributes()`. This method takes a dictionary of constants and values and assigns the attributes all at once.

Listing 4-59: Adding multiple attributes

```
import UIKit

let myfont = UIFont.systemFont(ofSize: 30)
let myrange = NSRange(location: 0, length: 5)
let color = UIColor.redColor()

let shadow = NSShadow()
shadow.shadowColor = UIColor.blackColor()
shadow.shadowOffset = CGSize(width: 2, height: 2)

let attributes = [NSFontAttributeName: myfont, NSForegroundColorAttributeName: color,
NSShadowAttributeName: shadow]

var mytext = NSMutableAttributedString(string: "Hello World")
mytext.setAttributes(attributes, range: myrange)

var label = UILabel(frame: CGRect(x: 20, y: 20, width: 150, height: 30))
label.attributedText = mytext
```

In the code of Listing 4-59 several attributes are assigned to the string all at once. Besides changing the font, as we did before, now we assign a new color to the text and a shadow. The values are defined first, including the object for the shadow, and then the `attributes` dictionary is created to include every constant and value we want to assign to the text. Finally, the `setAttributes()` method takes this dictionary and assigns the attributes to the

characters in the `myrange` range.

Do It Yourself: Replace the code on your Playground's file with the code of Listing 4-59. Remember to click the Show Result button corresponding to the statement you want to see represented in the editor.

IMPORTANT: There are more constants available. For a complete list, visit our website and follow the links for this chapter.

UIProgressView

The `UIProgressView` class provides properties and methods to create and define the style of a progress bar. This control was designed to depict the progress of a task over time. The class includes its own initializer and also a property and a method to set and read the progress.

UIProgressView(progressViewStyle: `style`)—This initializer creates a `UIProgressView` object with values by default and the style set by the `style` attribute. The attribute is an enumeration of type `UIProgressViewStyle` with the values `Default` (standard progress bar) and `Bar` (progress bar for Toolbars).

progress—This property sets or returns the progress of the task. The possible values are between 0.0 and 1.0 (the task was completed). The value by default is 0.0.

setProgress(progress, animated: `value`)—This method sets the value of the `progress` property. The `progress` attribute is a value between 0.0 and 1.0 (the task was completed), and the `value` attribute is a Boolean value to indicate if the transition will be animated.

The Progress View has a size by default, but it adapts to the size of its view. Using the `frame` property inherited from `UIView` we can resize the bar to our liking.

Listing 4-60: Creating a progress bar

```
import UIKit
```

```
var progressbar = UIProgressView(progressViewStyle: .Default)
```

```
progressbar.frame = CGRect(x: 50, y: 50, width: 200, height: 0)
```

```
progressbar.progress = 0.6
```


The bar is drawn with a standard height, so the `height` property of `frame` is ignored. Figure 4-8 illustrates the bar created by the code of Listing 4-60.

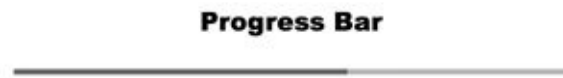


Figure 4-8: Progress View

The class also includes the following properties to configure the bar.

progressTintColor—This property sets or returns the color of the progress bar that is filled. It is an optional of type `UIColor`.

trackTintColor—This property sets or returns the color of the progress bar that is not filled. It is an optional of type `UIColor`.

progressImage—This property sets or returns the image used to illustrate the progress bar that is filled. It is an optional of type `UIImage`.

trackImage—This property sets or returns the image used to illustrate the progress bar that is not filled. It is an optional of type `UIImage`.

Listing 4-61: Customizing the progress bar

```
import UIKit

var progressbar = UIProgressView(progressViewStyle: UIProgressViewStyle.Default)
progressbar.progress = 0.2

progressbar.tintColor = UIColor.redColor()
progressbar.trackTintColor = UIColor.blueColor()
```

UIActivityIndicatorView

The `UIActivityIndicatorView` class provides properties and methods to create a spinning wheel that indicates a task is in progress. Unlike the progress bar studied before, this type of indicator has no implicit limitations. The class includes its own initializer.

UIActivityIndicatorView(activityIndicatorStyle: **style)**—This initializer creates a **UIActivityIndicatorView** object with the style set by the **style** attribute. The attribute is an enumeration of type **UIActivityIndicatorViewStyle** with the values **WhiteLarge** (large white indicator), **White** (standard white indicator), and **Gray** (standard gray indicator).

startAnimating()—This method starts the animation of the progress indicator.

stopAnimating()—This method stops the animation of the progress indicator.

isAnimating()—This method returns a Boolean value indicating whether the indicator is animating or not.

color—This property sets or returns the color of the indicator. It takes a value of type **UIColor**.

hidesWhenStopped—This property sets or returns a Boolean value to indicate if the indicator is going to be hidden when the animation stops.

Listing 4-62: Creating an activity indicator

```
import UIKit
```

```
var progress = UIActivityIndicatorView(activityIndicatorStyle: .Gray)
```

```
progress.frame.origin = CGPoint(x: 50, y: 50)
```

```
progress.color = UIColor.redColor()
```

```
progress.startAnimating()
```

```
if progress.isAnimating() {
```

```
    print("In Progress")
```

```
}
```

The code of Listing 4-62 creates a standard gray indicator but it changes its color to red. After the indicator is created, the **startAnimating()** method has to be called to start the animation. The code also checks the value returned by the **isAnimating()** method to confirm the indicator is animating. Figure 4-9 illustrates how a working indicator looks like.

Activity Indicator



Figure 4-9: Activity indicator

IMPORTANT: The Show Result button generates a window with a capture of the image produced by the control. Starting or stopping the wheel has no effect on this image. To see the indicator at work you have to include it in a real application.

UIControl

Some of the most valuable subclasses of `UIView` are actually subclasses of an intermediate subclass called `UIControl` (Figure 4-7). The `UIControl` subclass provides properties and methods to configure and retrieve information from the elements generated by its subclasses. The class includes things like properties to determine the current state of the element (selected, disabled, etc.) and also methods to respond to user interaction. The following are some of its properties and methods.

enabled—This property sets or returns a Boolean value indicating whether the element is enabled or not.

selected—This property sets or returns a Boolean value indicating whether the element is selected or not.

highlighted—This property sets or returns a Boolean value indicating whether the element is highlighted or not.

addTarget(target, action: method, forControlEvents: event)—This method configures a responder to an event. The **target** attribute is a reference to the object that receives the event (usually `self`), the **method** attribute is a selector (a string with the name of a method) indicating the method in the target object that is going to respond to the event, and the **event** attribute is the type of event the target will process.

removeTarget(target, action: method, forControlEvents: event)—This method removes a responder for an event. The values for the attributes are the same assigned by the `addTarget()` method.

The `UIControl` class does not create any element by itself, but serves as the basis for the construction of more complex classes that define visual controls capable of processing user interaction. Although the possibilities are unlimited, there are always common features every app requires. Considering this, UIKit includes predefined subclasses of `UIControl` to create standard controls that we can adapt to our needs. The rest of this chapter explores some of these subclasses and how to implement them.

UIButton

Probably the most simple but powerful subclass of `UIControl` is `UIButton`. This subclass creates a button composed by a text or an image that executes an action when the user taps on it. There are different kinds of buttons defined by the class, as illustrated in Figure 4-10.

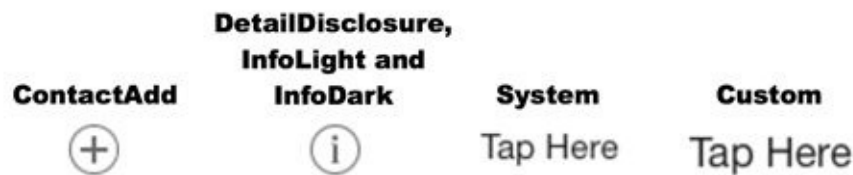


Figure 4-10: Button types

The graphic for the `DetailDisclosure`, `InfoLight`, and `InfoDark` buttons is the same, but the `InfoLight` button looks better on dark content and the `InfoDark` button looks better on light content. Additionally, the `System` and `Custom` buttons were design to show text, similar to a web link, but the `Custom` button can also present images and a custom appearance. The class uses the `UIView` initializer and a type method to create these buttons.

`UIButton(frame: frame)`—This initializer creates a `UIButton` object of `Custom` type and the size provided by the **`frame`** attribute. It takes a `CGRect` value to determine the area of the view.

`UIButton(type: type)`—This initializer creates a `UIButton` object configured according to the type specified by the **`type`** attribute. The attribute is an enumeration of type `UIButtonType` with the values `Custom`, `System`, `DetailDisclosure`, `InfoLight`, `InfoDark`, and `ContactAdd`.

`System` and `Custom` buttons have a title and also images and backgrounds that are fully customizable, but they are not permanent, they are associated to a specific state and change according to the condition of the button. For example, if the button is pressed, the state changes to **`Highlighted`** and the attributes associated to that state are assigned to the button. The possible states are defined by constants inside a structure called `UIControlState`. The constants available for buttons are **`Normal`** (the button is enabled and was not selected), **`Highlighted`** (the button is touched), **`Disabled`** (the button is disabled), and **`Selected`** (the button was selected). Every `UIButton` object includes the following properties and methods to modify the characteristics of the button in each of these states.

titleLabel—This property returns a `UILabel` object with the text of the button. The `font` property of this object can be modified to assign a new font to the text.

setTitle(title, forState: state)—This method sets the text of the button to the value specified by the `title` attribute and for the state specified by the `state` attribute.

setAttributedTitle(title, forState: state)—This method sets the text of the button to the value specified by the `title` attribute and for the state specified by the `state` attribute. The `title` attribute is an attributed string.

setTitleColor(color, forState: state)—This method sets the color of the text to the value specified by the `color` attribute and for the state specified by the `state` attribute. The `color` attribute is a `UIColor` object.

setImage(image, forState: state)—This method sets the image of the button to the one specified by the `image` attribute and for the state specified by the `state` attribute. Images are provided as `UIImage` objects.

setBackgroundImage(image, forState: state)—This method sets the background image of the button to the one specified by the `image` attribute and for the state specified by the `state` attribute. Images are provided as `UIImage` objects.

The simplest buttons to implement are the standard buttons, designed for a specific purpose. These buttons are shown on the screen with an image that has been configured to represent every possible state. The following example creates a button of type `ContactAdd` (see Figure 4-8).

Listing 4-63: *Creating a button of type `ContactAdd`*

```
import UIKit
```

```
let button = UIButton(type: .ContactAdd)
```

Do It Yourself: Replace the code on your Playground’s file by the code of Listing 4-63. Click on the Show Result button to see a representation of the button in between your code.

IMPORTANT: The Show Result button generates a window with a capture of the image produced by the views and their content. These images are incapable of processing user interaction and can’t adapt to new situations. In Chapter 5 we will study how to connect these objects to the rest of our code and process the actions performed by the user on these elements.

The area for the view of this type of button is determined by the size of the image, but the position may be declared by assigning the value to the **origin** property of the **frame** property inherited from the **UIView** class.

Listing 4-64: Positioning a standard button

```
import UIKit

let button = UIButton(type: .ContactAdd)
button.frame.origin = CGPoint(x: 50, y: 50)
```

The standard types are not used very often. Text buttons are more frequent and for that we have the **System** type. This type not only allows us to configure every aspect of the title of the button but also includes built-in functionality that makes it look professional (like a smooth transition between different states).

Listing 4-65: Customizing a text button

```
import UIKit

let button = UIButton(type: .System)
button.frame = CGRect(x: 50, y: 50, width: 100, height: 32)
button.titleLabel?.font = UIFont.systemFont(ofSize: 25)
button.setTitle("Tap Here", forState: UIControlState.Normal)
button.setTitleColor(UIColor.blueColor(), forState: UIControlState.Normal)
```

In Listing 4-65, the title and the color were assigned for the **Normal** state. This is the only one required. If no styles are specified for other states, the system defaults to **Normal**. We also change the font of the title's label. This affects every state. Notice that the **titleLabel** property returns an optional that we have to unwrap with **Optional Chaining**.

IMPORTANT: The code of Listing 4-65 is for illustration purposes only. At this moment Playground doesn't display **System** and **Custom** buttons and it is not possible to test button states and process user interaction. We will study how to create applications and implement these controls in real situations from Chapter 5.

UISegmentedControl

This `UISegmentedControl` class provides properties and methods to create a series of buttons that work together. Its initializer takes an array with `String` values to specify the title of each button.

`UISegmentedControl(items: values)`—This initializer creates a `UISegmentedControl` object with buttons created from the strings or images provided by the **`values`** attribute. The attribute is an array of `String` values or `UIImage` objects.

Each button of the segment is identified by an index starting from 0. The properties and methods in the class used this index to modify a specific button.

`numberOfSegments`—This property returns the number of buttons in the control.

`selectedSegmentIndex`—This property sets or returns the selected button in the control.

`setTitle(title, forSegmentAtIndex: index)`—This method sets the title for the button at the index specified by the **`index`** attribute. The **`title`** attribute is a `String` value.

`setImage(image, forSegmentAtIndex: index)`—This method sets the image for the button at the index specified by the **`index`** attribute. The **`image`** attribute is a `UIImage` object.

`insertSegmentWithTitle(title, atIndex: index, animated: value)`—This method inserts a button with the title specified by the **`title`** attribute at the index specified by the **`index`** attribute. The **`title`** attribute is a `String` value and the **`value`** attribute is a Boolean value to indicate if the process will be animated or not.

`insertSegmentWithImage(image, atIndex: index, animated: value)`—This method inserts a button with the image specified by the **`image`** attribute at the index specified by the **`index`** attribute. The **`image`** attribute is a `UIImage` object and the **`value`** attribute is a Boolean value to indicate if the process will be animated or not.

`removeSegmentAtIndex(index, animated: value)`—This method removes the button at the index specified by the **`index`** attribute. The **`value`** attribute is a Boolean value to indicate if the process will be animated or not.

`setEnabled(value, forSegmentAtIndex: index)`—This method sets the condition of the button at the index specified by the **`index`** attribute. The **`value`** attribute is a Boolean value to indicate whether the button is enabled (`true`) or disabled (`false`).

In a Segmented Control, only one button at a time is selected. When the user taps on a

button, the button tapped is selected and the rest are deselected. The change can also be performed from the code assigning a new value to the `selectedSegmentIndex` property. The following example creates a Segmented Control with two buttons, Yes and No, and sets the last button as selected.

Listing 4-66: Creating a Segmented Control

```
import UIKit

let segment = UISegmentedControl(items: ["Yes", "No"])
segment.frame.origin = CGPoint(x: 50, y: 50)
segment.selectedSegmentIndex = 1
```

Figure 4-11 illustrates the Segmented Control generated by the code of Listing 4-66 in two states, with none of the buttons selected and with the No button selected.



Figure 4-11: Segmented Control

The size of the control is determined automatically by the title of its buttons, the size of its view and the size of its superview. We can declare a specific size from the `frame` property.

Listing 4-67: Changing the size of a Segmented Control

```
import UIKit

let segment = UISegmentedControl(items: ["Yes", "No"])
segment.frame = CGRect(x: 50, y: 50, width: 200, height: 30)
```

The buttons and their titles are set by the initializer. They are created based on the number of elements in the array and their values. Sometimes, the options available change and the control has to adapt to the new situation. The `setTitle()` method let us assign a new title to a specific button after the control was already created.

Listing 4-68: Creating a Segmented Control

```
import UIKit
```



```
let segment = UISegmentedControl(items: ["Yes", "No"])
segment.frame = CGRect(x: 50, y: 50, width: 200, height: 30)
segment.setTitle("Agree", forSegmentAtIndex: 0)
```

Besides changing the titles we can also insert new buttons after the control was already created with the `insertSegmentWithTitle()` method. This method takes the title for the new button, the index where the button should be placed, and a Boolean value to animate the process.

Listing 4-69: Adding new buttons to a Segmented Control

```
import UIKit

let segment = UISegmentedControl(items: ["Yes", "No"])
segment.frame = CGRect(x: 50, y: 50, width: 200, height: 30)
segment.insertSegmentWithTitle("Not Sure", atIndex: 2, animated: false)
```

UISwitch

The `UISwitch` class provides properties and methods to create a switch to select between two possible values (`true` or `false`). The user selects the current value by swiping a round button to one side or another, turning the switch on and off. The class uses the `UIView` initializer to declare the area of the view, but the `size` component is ignored because the switch always has a size by default.

Listing 4-70: Creating a switch

```
import UIKit

let myswitch = UISwitch(frame: CGRect(x: 50, y: 50, width: 0, height: 0))
```

Switches are turned on and off by the user, but the class includes some tools to modify this state from the code.

on—This property sets or returns the state of the switch. It takes a Boolean value to set the switch on (`true`) or off (`false`).

setOn(state, animated: value)—This method sets the state of the switch. The **state** attribute is a Boolean value to set the switch on (`true`) or off (`false`), and the **value** attribute is a Boolean value to indicate if the transition will be animated.

Listing 4-71: Switching the switch on

```
import UIKit

let myswitch = UISwitch(frame: CGRect(x: 50, y: 50, width: 0, height: 0))
if !myswitch.on {
    myswitch.on = true
}
```

The code of Listing 4-71 reads the **on** property to check the state of the switch first, and if it is off it turns it on. Figure 4-12 illustrates the two possible states.



Figure 4-12: Switch

When the switch is turned on its background color changes. We can modify this color, the color of the border and the button using the following properties.

onTintColor—This property sets or returns the color of the switch’s background when it is turned on. It is an optional of type **UIColor**.

tintColor—This property sets or returns the color of the switch’s border when it is turned off. It is an optional of type **UIColor**.

thumbTintColor—This property sets or returns the color of the switch’s button. It is an optional of type **UIColor**.

Listing 4-72: Changing the colors of the switch

```
import UIKit

var colorBackground = UIColor(red: 0.9, green: 0.9, blue: 1.0, alpha:1.0)
var colorButton = UIColor(red: 0.5, green: 0.5, blue: 1.0, alpha: 1.0)

let myswitch = UISwitch(frame: CGRect(x: 50, y: 50, width: 0, height: 0))
myswitch.onTintColor = colorBackground
myswitch.tintColor = colorBackground
myswitch.thumbTintColor = colorButton
```

In this last example, the colors are declared first and then assigned to the corresponding properties. This time we used RGB colors to paint the switch with different tones of blue.

UISlider

The `UISlider` class provides properties and methods to create a control that lets the user select a value from a range of values. The control is presented as a horizontal bar with a round indicator to point to the value selected. The class uses the `UIView` initializer to set the view. The control adapts its size to the size of the view, but we must set minimum values for the graphics to fit inside.

Listing 4-73: Creating a Slider

```
import UIKit

let slider = UISlider(frame: CGRect(x: 50, y: 50, width: 200, height:31))
```

This presents a simple Slider with a range of values from 0.0 to 1.0 and the indicator pointing at the value 0.0. To adapt the control to our needs, we have to read and modify its properties.

value—This property sets or returns the selected value. When the value is set, the indicator is moved to the corresponding position.

minimumValue—This property sets or returns the minimum value for the Slider. It takes a value of type `Float`.

maximumValue— This property sets or returns the maximum value for the Slider. It takes a value of type `Float`.

setValue(selected, animated: value)—This method sets the selected value as the value specified by the **selected** attribute and moves the indicator to the corresponding position. The **value** attribute is a Boolean value to specify if the indicator will be animated or not.

Listing 4-74: Setting the values for the Slider

```
import UIKit
```

```
let slider = UISlider(frame: CGRect(x: 50, y: 50, width: 200, height:31))
```

```
slider.minimumValue = 0.0
```

```
slider.maximumValue = 10.0
```

```
slider.value = 6.0
```

The code of Listing 4-74 sets the minimum value at 0.0, the maximum value at 10.0, and the selected value as 6.0. The result is shown in Figure 4-13.



Figure 4-13: Slider

The class also provides properties and methods to customize the Slider, changing colors and adding images on the sides to represent minimum and maximum values.

minimumTrackTintColor—This property sets or returns the color of the bar on the minimum side. It is an optional of type **UIColor**.

maximumTrackTintColor—This property sets or returns the color of the bar on the maximum side. It is an optional of type **UIColor**.

minimumValueImage—This property sets or returns the image that represents the minimum value. It is an optional of type **UIImage**.

maximumValueImage—This property sets or returns the image that represents the maximum value. It is an optional of type **UIImage**.

Listing 4-75: Changing the colors of the Slider

```
import UIKit
```

```
let slider = UISlider(frame: CGRect(x: 50, y: 50, width: 200, height:31))
```

```
slider.minimumValue = 0.0
```

```
slider.maximumValue = 10.0
```

```
slider.value = 9.0
```

```
slider.minimumTrackTintColor = UIColor.redColor()
```

```
slider.maximumTrackTintColor = UIColor.blueColor()
```

UIStepper

The `UIStepper` class provides properties and methods to create a set of two buttons with the minus and plus symbols that allow users to increment or decrement a value. The class uses the `UIView` initializer to declare the area of the view, but the `size` component is ignored and the control is always drawn with a size by default.

Listing 4-76: Creating a Stepper

```
import UIKit
```

```
let stepper = UIStepper(frame: CGRect(x: 50, y: 50, width: 0, height: 0))
```

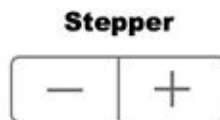


Figure 4-14: Stepper

The initializer used in the code of Listing 4-76 returns a Stepper with values by default. The minimum value is set to 0.0, the maximum value is set to 100.0, the initial value is set to 0.0 and the incremental value is set to 0.0. We can adapt the control to our needs from its properties.

value—This property sets or returns the current value of the Stepper.

minimumValue—This property sets or returns the minimum value for the Stepper. It takes a value of type **Double**.

maximumValue—This property sets or returns the maximum value for the Stepper. It takes a value of type **Double**.

stepValue—This property sets or returns the number by which the current value of the Stepper will be incremented or decremented. It takes a value of type **Double**.

The class also includes properties to configure the behaviour and appearance of the control.

autorepeat—This property sets or returns a Boolean value to indicate if the value is incremented automatically. If it is set to **true**, the value of the Stepper is incremented

repeatedly when the user presses and holds down the button.

continuous—This property sets or returns a Boolean value to indicate whether the updated values are report during the user interaction or only when the user interaction ends.

wraps—This property sets or returns a Boolean value to indicate how the value is processed when it reaches the minimum or maximum values allowed. If **true**, when the value goes beyond the limit the opposite limit is assigned to it and the value keeps decrementing or incrementing in a loop. On the other hand, when the property is **false**, the value stops decrementing or incrementing when a limit is reached.

tintColor—This property sets or returns the color of the control.

Listing 4-77: Setting the values for the Stepper

```
import UIKit

let stepper = UIStepper(frame: CGRect(x: 50, y: 50, width: 0, height: 0))
stepper.value = 4.0
stepper.minimumValue = 1.0
stepper.maximumValue = 9.0
stepper.stepValue = 0.5
```


Chapter 5

Building Applications

5.1 The Interface

Up to this point we have been working with a simplified interface called Playground. Playground is a recent addition to Xcode designed to reduce the learning curve and simplify testing of new and experimental code. It is good for learning and testing, but to build full applications we have to move to the Xcode's main interface. This is a toolset conformed by an editor, a visual interface, resource managers, configuration panels, and debugging and testing systems, all integrated in a single working space where we can create and modify our applications.

Xcode Projects

Applications are built from several files and resources, including our own codes, frameworks, images, databases, etc. Xcode organizes this information in projects. An Xcode project comprises all the information necessary to create one application. The welcome window, illustrated in Figure 1-2, presents a button called *Create a new Xcode project* to initiate a project. When we click on this button, a new window lets us choose the template we want to use to create our project.

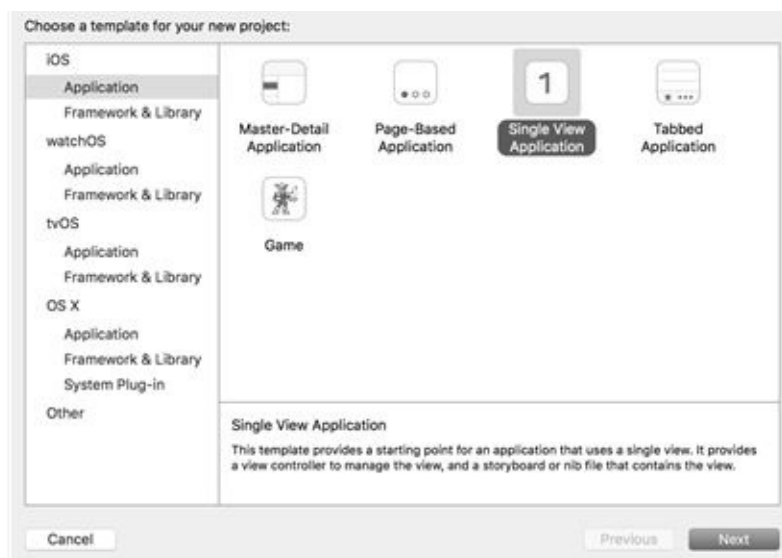


Figure 5-1: *Selecting the type of project*

The selection screen organizes all the options in several categories, iOS, watchOS, tvOS, and OS X, representing each operative system Apple has to offer (shown on the left column). To develop applications for Apple's mobile devices we have to select the options under the iOS category.

Xcode provides templates to give us something to start with. There are several

templates available for every platform and different types of projects, such as 2D and 3D video games, single or multiple windows and views, etc. Templates include some basic files with pre-programmed code that help us organize our own code according to the purpose of our app. The following are the options currently available for mobile applications.

Single View Application—This option creates a basic template with only one view. It is a very basic template but more views may be added later according to our needs.

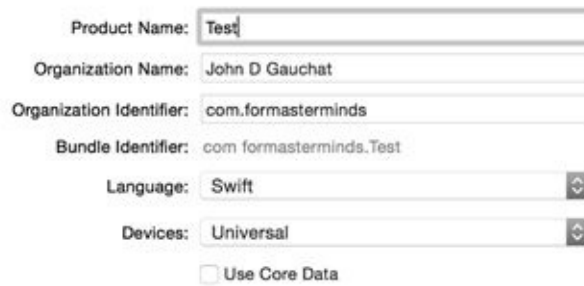
Master Detail Application—This option creates a structure of views necessary to divide the screen into two parts: the Master view (to contain menus or options) and the Detail view (to contain the most relevant information). This is a very useful template for iPad applications, but can also be used on iPhones and iPods.

Page Based Application—This option creates a structure where views are like the pages of a book. The views are replaced by swiping the fingers from one side to another.

Tabbed Application—This option creates a container view with other views inside and a bar with navigation buttons. The buttons allow users to select the view they want to see.

Games—This option creates a view and a few elemental files necessary to program a video game using specific libraries such as SceneKit, SpriteKit, OpenGL ES, or Metal.

After selecting a template, the next step is to insert information about the project. The information required depends on the type of project selected. For example, a project to create a mobile application asks for the name of the project, the name of our organization and also a company identifier (Figure 5-2 below). For some templates we will also have to select the programming language we want to use, the libraries that have to be included in our project, and the type of app we want to create: *Universal* for apps that run in every device and *iPhone* or *iPad* for a specific device. All this is personal information and does not demand anything in particular, except for the company's identifier. We can use any name we want for our product and declare our own name as the Organization's name, but the Company Identifier has to be unique. This is the reason why Apple recommends using an inverted domain. The window depicted in Figure 5-2 below shows an example. This is the configuration window for a Single View Application template. Here, the inverted domain `com.formasterminds` declared as the Company Identifier is applied automatically along with the app's name to create the Bundle identifier (`com.formasterminds.Test`). Using the inverted domain ensures that the identifier won't be duplicated.



Product Name: Test

Organization Name: John D Gauchat

Organization Identifier: com.formasterminds

Bundle Identifier: com.formasterminds.Test

Language: Swift

Devices: Universal

☐ Use Core Data

Figure 5-2: Project configuration window

IMPORTANT: Although it's not mandatory, you should get your own domain and website. Apple not only recommends the use of a domain to generate the Bundle Identifier, but at the moment of submitting your app to the App Store you will be asked to provide the web page used for promotion and where the users should go for support. We will study how to submit an app in Chapter 32.

Most templates offer an option at the bottom called *Use Core Data* to add a database to our project. Core Data is a database system designed for Apple applications. This option has to be left unmarked unless our application requires this system (we will study Core Data in Chapter 22).

When the configuration is over, we have to press the Next button and select the folder to store the project. Xcode creates a folder for each project, so we only need to designate the destination folder where we are going to store all of our projects and everything else is generated for us.

Xcode Tools

After all the information for the project is provided, Xcode creates the initial files according to the template selected and presents the main interface on the screen. Figure 5-3, next, shows what this interface looks like.

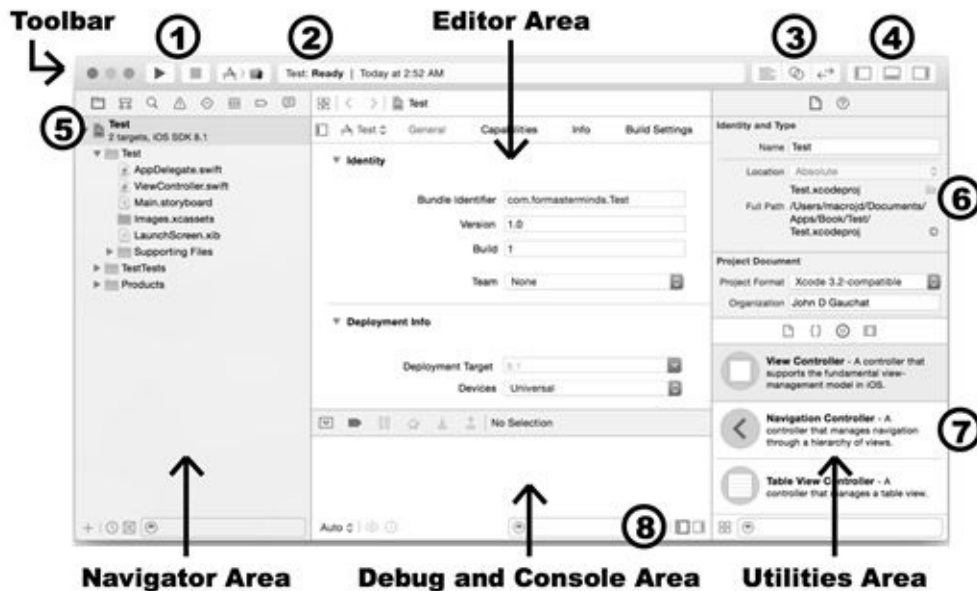


Figure 5-3: Xcode's interface

The window is organized in several areas. There is a Toolbar at the top with configuration buttons, an area for edition at the center called Editor Area, and three removable panels on the sides and bottom called Navigator Area, Debug and Console Area, and Utilities Area. Although we will study each tool later, at the moment they are required, it is a good idea to get familiar with the main parts of the Xcode's interface before working on our first project.

Toolbar—This is a horizontal bar at the top that lets us control the appearance of the rest of the interface and keeps us informed about warnings, errors, and the app's status during execution. It provides two buttons to run and stop the app (Figure 5-3, number 1), a display to show the status of the building and execution of the app (Figure 5-3, number 2), three buttons to split the Editor Area (Figure 5-3, number 3), and another three buttons at the top right corner to show or hide the removable areas of the interface (Figure 5-3, number 4).

Navigator Area—This removable area provides information related to the files that comprise the application and also tools for debugging (identify and remove errors in the code). From here we can select the files to edit, create groups to organize those files, add resources, check for errors, set breakpoints, etc. In addition to the files, this area shows an option at the top to configure the app (Figure 5-3, number 5).

Editor Area—This is the only non-removable area, and is the area where we will do much of the work. The content of the files is displayed here and also the app's configuration.

Debug and Console Area—This removable area splits into two sections that can be shown or hidden by the buttons at the bottom (Figure 5-3, number 8). The left section provides information for debugging, while the right section is a console to display the

results of the execution of our code, including warnings and errors (the strings printed by the `print()` method are shown here).

Utilities Area—This removable area provides additional information about the files and their content (Figure 5-3, number 6). It also presents tools to create the user interface (Figure 5-3, number 7). From here we can add elements to our app’s interface and edit their configuration.

Do It Yourself: Open Xcode and select the option *Create a new Xcode project* from the welcome window (Figure 1-2). On the template window, go to the Application menu for iOS and select the option *Single View Application* (Figure 5-1). Insert the information for your project as shown in Figure 5-2 (remember to set the language as “Swift” and the devices as “Universal”). Select the folder to store the project and click Create. You should see a window similar to Figure 5-3.

Running the Project

Xcode offers two ways to run our application: the simulator or a real device. The buttons to select these options, execute, and stop the app are at the left-hand side of the Toolbar.



Figure 5-4: Buttons to execute the app

Applications are always executed on a specific target. These targets could be different things, from real devices to windows or simulators, and have different configurations, including settings like the region where the target is located and the human language used by the app to display the information on the screen. Xcode let us define and configure the possible targets for our app using an arrangement called *Scheme*. When a project is created, a Scheme is automatically generated for us with options that are according to the type of project we are creating and our local settings (region and language set on the computer). For example, the Scheme of a project for a mobile application includes several iOS simulators and offers access to the devices currently connected to the computer. The target is selected by clicking on the Scheme button located next to the Play and Stop buttons (Figure 5-4). The option opens a list with the real devices at the top and the simulators at the bottom.



Figure 5-5: Options available to run the app

During development, the most common target is the simulator. The simulator is a program included in the Xcode's package that reproduces a specific device in a separate window. It is a very practical tool because it is fast, reproduces almost every feature of a real device, and can run fully functional applications. When we want to run the app, all we have to do is select the device we want to simulate from the scheme and then click Play. Figure 5-6 shows the simulator configured as an iPhone 4s, running the app created by the Single View Application template.



Figure 5-6: iPhone simulator running a template

The Basics: When a project is created from the Single View Application template the user's interface is declared as an empty view. That is why the simulator of Figure 5-6 only shows the status bar and an empty screen. The template leaves everything ready for us to start adding graphic elements and writing our code.

Running an app on the simulator doesn't present any complications, but do it on a real device has some requirements. Apple has strict policies regarding who can run an application and on which device. To identify who is running and developing the apps, we have to insert our Apple ID (email and password) in Xcode's configuration panel. The panel is accessible from the Xcode menu at the top of the screen. We have to select the

option Preferences and then select the Accounts tab, as shown in Figure 5-7, number 1.

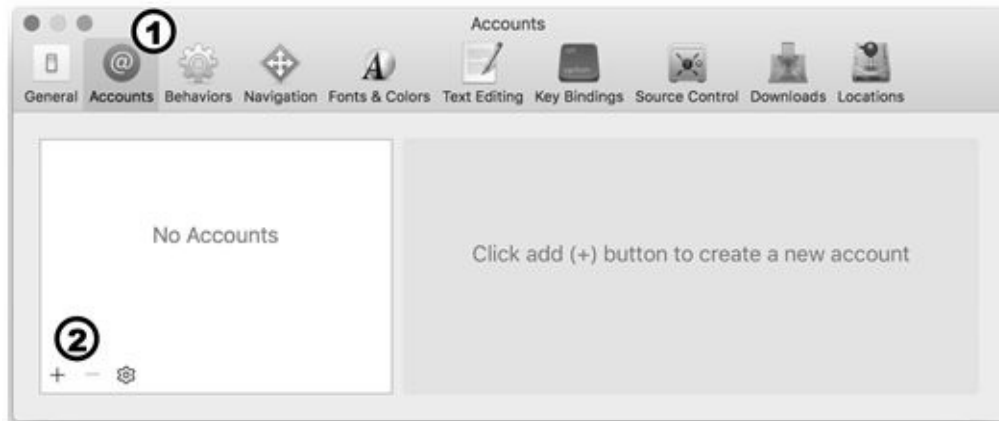


Figure 5-7: Xcode's Preferences window

The Accounts tab offers buttons to add and remove accounts (Figure 5-7, number 2). To add our account we have to click the plus button and select the option Add Apple ID in the popup menu that shows up. The window presents a form with fields to insert our email and password.



Figure 5-8: Registering our Apple ID in Xcode

Once we insert the information, the account is added to the list and configured to work with this copy of Xcode. The next step is to authorize the applications to run in our device. We have to get a certificate that validates our developer account for developing applications and a provisioning profile that allows our devices to run the applications. The easiest way to generate these values is to let Xcode do it for us. If we connect the device to the computer, select it on the Scheme (Figure 5-5) and press Play to run the application, Xcode will insert all the necessary signatures for our app to run and will make suggestions to fix the issues it finds. When all the issues are taken care of, the app is installed in the device and executed.

Do It Yourself: If you didn't do it yet, create the Test project following the steps described in the previous section. With the project opened, press the Scheme button (Figure 5-4) and select one of the iPhone's simulators. Press the Play button to run the app. After a few seconds, you should see the simulator's window on the screen (Figure 5-6).

IMPORTANT: Certificates and provisioning profiles are also necessary to publish the application in the App Store. If you find problems running your app or want to learn more about how to create certificates and provisioning profiles, read Chapter 32.

5.2 Templates

Templates are a simple way to start building our application. They include files with basic code to let us concentrate on what is important for our app right from the beginning. But to take advantage of these files we have to understand first how Xcode suggests applications should be built and the tools it provides for this purpose.

MVC

Xcode proposes an architectural paradigm that divides the application in three interconnected parts: the Model, the View, and the Controller (called MVC for short). Every part of the code performs a specific task and communicates to the other parts only the information strictly necessary. Figure 5-9 illustrates the elements involved.

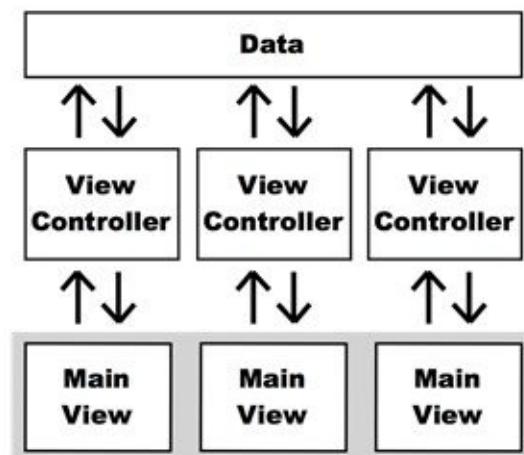


Figure 5-9: MVC architecture

The Model is the data the app has to process, the View is the user's interface, conformed by the window and its views, and the Controller are the objects in charge of reading and processing the data and also updating and receiving input from the views. The objects managing the data and the views are not connected with each other, they send the information to the controller and this code decides what to do with it. This allows reusability. We can use the same code to control different views or replace the user's interface entirely without having to change a line of code in the rest of the program. As long as they know what to do with the information received from other objects and return the right values in response, each part remains independent.

This structure is created with several objects. The interface is created from `UIView` objects. These objects and the objects managing the data for the app are connected to controller objects (one for each main view in the interface). Along with these objects,

there are also the **UIApplication** object keeping everything running and the **AppDelegate** object, used by the **UIApplication** object to communicate with our code.

The application loop and the window are created directly from the **UIApplication** and **UIWindow** classes studied in Chapter 4, but the rest of the objects are created from the definitions inside the files provided by the templates and our own code. Figure 5-10 shows the Navigator Area with the list of files generated by the Single View Application template.

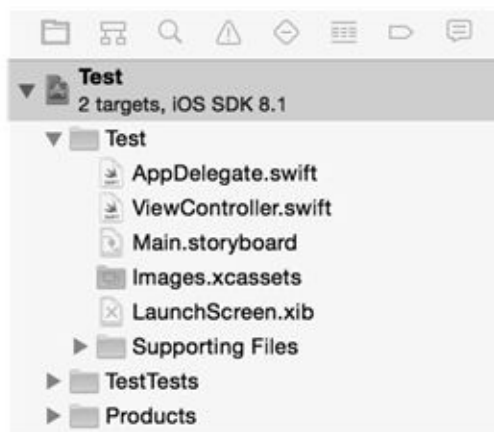


Figure 5-10: Template's basic files

The Basics: Xcode organizes the files in groups. Groups are not real directories. All the files for our app are stored in the same directory, but they are organized in groups to help us find them on the Navigator Area. If you can't see the files, click on the first button of the Toolbar at the top of the Navigator Area (Figure 5-10) or go to the View menu at the top of the screen, select the option Navigators, and click on Show Project Navigator.

A minimalistic app requires at least three files. There is a file defining the delegate of the **UIApplication** object that is going to establish the communication between the application loop and our code (**AppDelegate.swift**), a file defining the object that will control the initial main view and its content (**ViewController.swift**), and a file defining the objects that will create the views and represent every screen of the interface (**Main.storyboard**).

Application Delegate

One of the files available in every template is called **AppDelegate.swift**. This file defines a class called **AppDelegate** to create an object that will be the delegate of the **UIApplication** object. Once created, the **AppDelegate** object is assigned to the **delegate** property of the

UIApplication object and becomes the contact between this object and our code. The initial file contains a property called **window** and a few empty methods. The property is just a reference to the **UIWindow** object created for our app, and the methods are some of the methods defined by the **UIApplicationDelegate** protocol adopted by this class.

There are different states associated with an application. At a certain moment, the application may be Not Running, Inactive, Active, in the Background, or Suspended. For example, when the users tap on the app's icon, the application is loaded and becomes Active, but if the user later presses the Home button to open a different application, the app's state changes to Background. Every time the state of the application changes, the **UIApplication** object calls the corresponding method on the **AppDelegate** object to let us know what happened. The following are some of the methods defined by the **UIApplicationDelegate** protocol.

application(application, didFinishLaunchingWithOptions: launchOptions)

—This method is the first method to be executed. It is called to let us know that the basic objects were already instantiated and are ready to work. We can write inside this method all the code we need to prepare our app, such as declaring initial values, creating or opening a database, checking the current settings, etc.

applicationWillResignActive(application)—This method is called when the app becomes inactive, usually due to an incoming call or message.

applicationDidEnterBackground(application)—This method is called when the app is closed (it moves to the background).

applicationWillEnterForeground(application)—This method is called before the app becomes active again.

applicationDidBecomeActive(application)—This method is called when the app is launched and also when it comes back from the inactive state, usually due to an incoming call or message.

applicationWillTerminate(application)—This method is called before the app is being terminated. This state usually occurs when the system decides to terminate the app because it needs the memory for other applications.

All the methods in the **UIApplicationDelegate** protocol are optional. This means that we can implement only the methods our application needs. For example, some simple applications only implement the **application(application, didFinishLaunchingWithOptions: launchOptions)** method to initialize general properties.

Listing 5-1: *Defining our own AppDelegate.swift file*

```
import UIKit
```

```

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    var basicSalary: Double?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject:
    AnyObject]?) -> Bool {
        basicSalary = 30000.0
        return true
    }
}

```

The main purpose of the **AppDelegate** class defined in this file is to provide the methods that the **UIApplication** object will call to report changes in the state of the app. The methods are used to prepare the app and adapt it to new circumstances, but because the application's delegate is accessible from every part of the code through the **delegate** property of the **UIApplication** object, it is common practice to declare generic values inside this class and also initialize data containers such as databases or files. In the example of Listing 5-1, we have declared a single property to illustrate how the **AppDelegate** object and its methods work, but later we will see that the object is frequently used for more complex operations, such as opening and accessing files and databases and also set initial conditions for the app.

Do It Yourself: Click on the AppDelegate.swift file in the Navigator Area to see its content in the Editor Area. The code generated by the template includes all the methods listed above with comments explaining what the methods should be used for. Erase every one of them except the **application(application, didFinishLaunchingWithOptions: launchOptions)** method and add the declaration and initialization of the **basicSalary** property, as shown in Listing 5-1. We will work with this property later.

Storyboard

The user interface is built with views inside a single window. One view, called *main view* or *container view*, is responsible for organizing all the graphic elements on the screen. Several main views put together simulate virtual screens. This is how the app manages different graphic screens and expands the interface. The main views replace one another inside the window to give the impression of having more space than a single screen could offer. Figure 5-11 illustrates this organization.

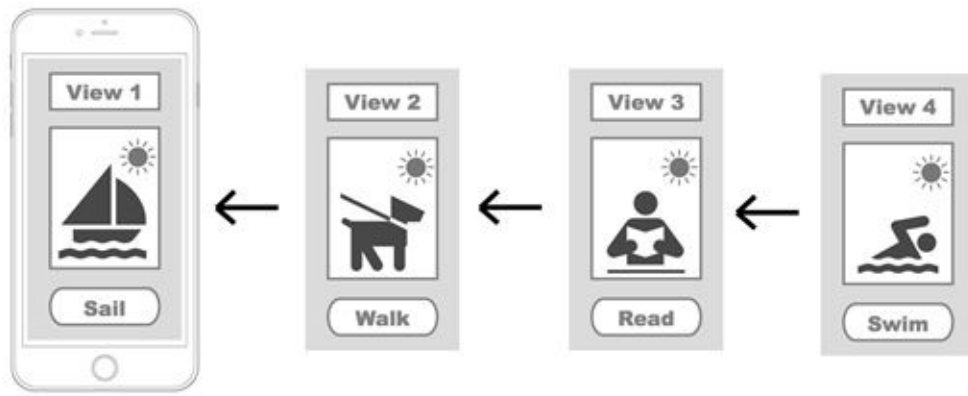


Figure 5-11: Multiple main views to expand the interface

Creating and organizing these views and their content from code could demand hours of meticulous work. It is very difficult to achieve the desired design and functionality for the user interface when we have to program every single element the way we did in Chapter 4. This is why Xcode introduces a graphic tool called *Storyboard*. The Storyboard produces a result similar to Figure 5-11. It lets us organize the main views and their content visually. Every component of the interface is displayed in the Editor Area, and we can drag and connect everything without writing a single line of code. All the information for the views created on the Storyboard is stored in an XML file. When the app is executed, this code is interpreted, the real objects are instantiated, and the views are drawn on the screen.

The Basics: An XML file is just a text file written in a language called *XML* that is easy to read for humans and machines. These types of files are frequently used to transfer documents from one server to another, but because of its characteristics it was adopted by Xcode to represent objects.

The files that store the information for the Storyboard have the **storyboard** extension. Templates always include a storyboard file with a few initial views. Figure 5-12 shows what we see on the Editor Area when we click on the Main.storyboard file included in the Single View Application template.

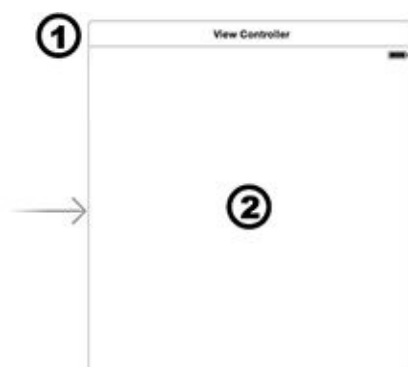


Figure 5-12: Main view on the Storyboard (*Main.storyboard*)

IMPORTANT: Recent versions of Xcode show the Editor Area partially covered by a panel on the left side when the storyboard file is selected. This panel is called Document Outline and its purpose is to offer quick access to the elements on the Storyboard. You can close this panel by clicking the button at the bottom-left corner of the Editor Area (see Chapter 6, Figure 6-3). We will learn more about the Document Outline panel in Chapter 6.

The Basics: The main views presented by default on the Storyboard have a square shape because they are a general representation of the interface that applies to any device. For the examples of this chapter, we recommend to position the subviews inside the main view close to its left side to be able to see them on any device and configuration. In Chapter 6 we will study how to adapt the size of the main views and the look of the interface to every device.

The Storyboard uses square boxes called *Scenes* to represent main views on the screen. A Scene contains the main view where the interface is created (Figure 5-12, number 2) and a small bar at the top with options to configure the Scene and the view controller (Figure 5-12, number 1). Clicking on the main area selects the main view, and clicking on the bar selects the Scene and the view controller. The Scene and the view controller may also be selected from the series of buttons shown on the bar when we click on it (Figure 5-13). The button on the left represents the view controller responsible for the Scene's main view, the button at the center provides options to configure the element designated as first responder (studied in Chapter 7), and the button on the right lets us connect main views together (studied in Chapter 13).



Figure 5-13: Buttons to configure the Scene

Every time an element on the Storyboard is selected (either the Scene, the main view, or its content), the Utilities Area on the right shows a series of panels for configuration. The panels are selected from the buttons at the top of the area. The most useful are the Attributes Inspector panel and the Size Inspector panel. From the Attributes Inspector panel we can modify properties related to the attributes of the selected element, and from the Size Inspector panel we can change its size and position. Figure 5-14 shows how to open these panels and some of the properties they include when the main view is selected.

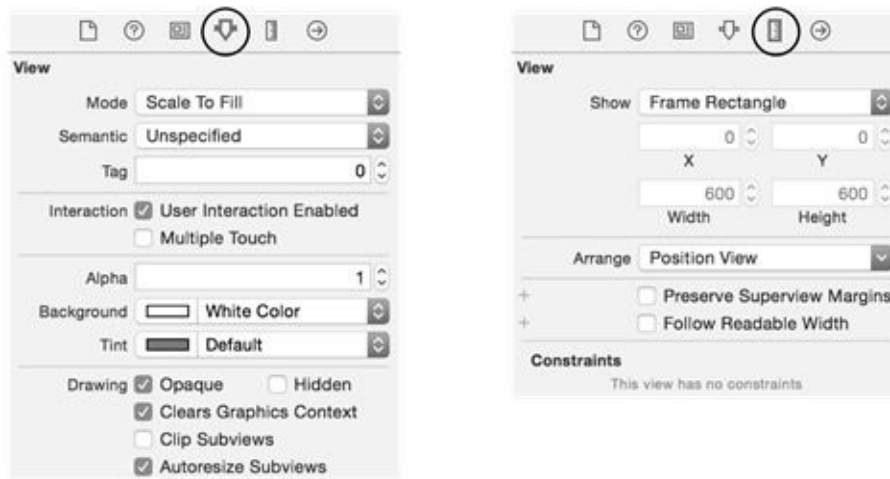


Figure 5-14: *Attributes Inspector panel and Size Inspector panel*

The values on each panel of Figure 5-14 correspond to the same properties studied in Chapter 4 for `UIView` objects. For example, the value defined for the Background color on the Attributes inspector panel will be assigned to the `backgroundColor` property of the `UIView` object responsible for the main view. The same happens with the rest of the properties displayed on the panels. All the values defined on these panels will be assigned to the corresponding properties of the object created later to represent the element. From here, we can visually modify all the values and configure the elements on the Storyboard without writing a single line of code.

Do It Yourself: Click on the `Main.storyboard` file in the Navigator Area to see its content in the Editor Area. When a storyboard file is selected, the Editor Area turns into a graphic tool that represents the components of the file. The `Main.storyboard` file of the Single View Application template includes a single main view with no content. When you click on this file, you should see something similar to Figure 5-12. Click on the main view or on the bar at the top of the Scene to see the configuration options in the Utilities Area. Select the main view and change the color of its background (Figure 5-14).

View Controllers

Each main view on the Storyboard is responsible for showing its content on the screen and adapting to new circumstances, which presents some challenges. First, each main view has content to manage and the code responsible to do it will be different for every one of them. Second, there is information we need to obtain about the view that is not related to the view itself but to the system, such as when the view becomes visible, when a transition from one view to another begins or ends, etc. And third, every view has to resize and reposition its content according to changes in the window, the orientation of the screen and

the device. These are complex tasks that require a close connection between our code and the rest of the application. With the purpose of controlling each view and performing any standard function they require, the UIKit framework includes the `UIViewController` class. This is a basic class with properties and methods to manage a single view and its content, and also respond to changes on its state. Although it provides everything we need to control a particular main view, we don't work directly with objects created from this class. Instead, we define a subclass of `UIViewController` for every main view on the Storyboard and add our own properties and methods to perform the tasks that are specific for that view and our app.

Most templates follow this pattern and always include files with subclasses of the `UIViewController` class to control the views they provide. For example, the `ViewController.swift` file created by the Single View Application template declares a class called `ViewController` that inherits from the `UIViewController` class and is in charge of controlling the only main view included by the template on the Storyboard. The initial file is mostly empty but it has two overwritten methods of its superclass: `viewDidLoad()` and `didReceiveMemoryWarning()`. These are methods of the `UIViewController` class that are executed to report changes in the condition of the view. We have to overwrite these methods in our subclass if we want to program our own response. Along with these two, the class includes a few more methods to report these states.

`viewDidLoad()`—This method is called after the main view and its content is loaded in memory. It is the first method called on the `UIViewController` object to indicate that the main view is ready to be shown on the screen.

`viewWillAppear()`—This method is called after the `viewDidLoad()` method and before the main view is shown on the screen.

`viewDidAppear()`—This method is called after the main view is shown on the screen.

`viewWillDisappear()`—This method is called before the main view is removed from the screen.

`viewDidDisappear()`—This method is called after the main view has been removed from the screen.

`didReceiveMemoryWarning()`—This method is called when the app receives a memory warning from the system. It is used to free memory and avoid our app being terminated.

Usually we don't need other methods than `viewDidLoad()` to set up the view and the `didReceiveMemoryWarning()` to take care of memory. This is why these are the only methods declared by the template. The next example reads the property we added to the `UIApplication` delegate in Listing 5-1 and prints its value on the console as soon as the view controlled

by the `ViewController` object is loaded.

Listing 5-2: *Adding our own code to the `ViewController.swift` file*

```
import UIKit
```

```
class ViewController: UIViewController {  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        let app = UIApplication.sharedApplication()  
        let mydelegate = app.delegate as! AppDelegate  
        if let salary = mydelegate.basicSalary {  
            print("Basic Salary is \(salary)")  
        }  
    }  
}
```

The first thing we do inside the `viewDidLoad()` method in Listing 5-2 is call the same method on the superview. When a method is overwritten, the original method is not executed anymore. Some classes, like the `UIViewController` class, include methods that execute essential tasks necessary for the app to work properly. If we don't call them, those tasks are never performed.

After the method is called in the superview, we can include our own code. In this example, we get a reference to the `UIApplication` object to read the properties of its delegate. First we obtain the reference by calling the `sharedApplication()` method in the class. Next, we read the object's `delegate` property and assign it to the `mydelegate` constant (this is the object created from the class declared in the `AppDelegate.swift` file). Notice that we have to cast this object as `AppDelegate` with the `as!` operator because the `AppDelegate` object is a subclass of the `UIResponder` class (a basic class in `UIKit`) and the `delegate` property returns a value of this type. Finally, we use Optional Binding to read the `basicSalary` property of the delegate and print its value on the console.

When the app is executed, the system creates all the basic objects, including the `UIApplication` object and its delegate, and then calls the `application(application, didFinishLaunchingWithOptions: launchOptions)` method of the `AppDelegate` object. In this method we define and initialize the `basicSalary` property. After the execution of this method is over, the system looks for the main view that has to show first and calls the `viewDidLoad()` method of its view controller. In this method we get the value of the `basicSalary` property defined in the delegate object and print it on the console.

Do It Yourself: The last example assumes you have already declared the `basicSalary` property in the app's delegate as shown in Listing 5-1. Replace the code in your

ViewController.swift file by the code of Listing 5-2. Press the Play button to execute the app (Figure 5-4). The value of the `basicSalary` property is printed on the console, at the bottom of the Xcode's interface (Figure 5-3, Debug and Console Area). If you can't see the console, press the middle button at the right-hand side of Xcode's Toolbar (Figure 5-3, number 4) and check the console is not hidden by the buttons at the bottom of the panel (Figure 5-3, number 8).

IMPORTANT: The main views defined in the Storyboard conform a path the user may follow to achieve a goal (e.g., access information, find a picture, etc.). The starting point of the path is determined by assigning the view controller object that controls the initial main view to a property of the `UIWindow` object called `rootViewController`. This may be done programmatically, but you can also do it from the Storyboard. All you have to do is point the arrow on the screen to the main view you want to show first (Figure 5-12). If the arrow is not visible, you can select the Scene you want to show first, go to the Utilities Area, open the Attributes Inspector panel (Figure 5-14), and activate the option *Is Initial View Controller*.

A `UIViewController` object not only manages the main view but also its content. The class includes the `view` property to store a reference to the main view the object controls. After the main view associated with the view controller is created from the Storyboard a reference is assigned to its `view` property. Using this property we can manage the views' content, add subviews, etc. The following example adds a `UILabel` view to the main view created by the template.

Listing 5-3: *Adding a subview to the main view from the `ViewController` object*

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        let app = UIApplication.sharedApplication()
        let mydelegate = app.delegate as! AppDelegate
        if let salary = mydelegate.basicSalary {
            let title = UILabel(frame:CGRect(x:20, y:20, width:300, height:50))
            title.text = "Basic Salary is \(salary)"
            view.addSubview(title)
        }
    }
}
```

This new version of our `ViewController` class creates a `UILabel` view with a string that includes the value of the `basicSalary` property defined in the application's delegate. The label is added to the main view next by calling its `addSubview()` method. Remember that the `view` property of every view controller has a reference to the main view in the Storyboard so we can add anything we want to the view from this property and this way modify the user interface.

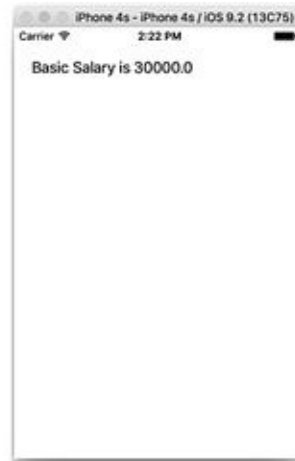


Figure 5-15: Label added to the initial screen

Do It Yourself: Replace the code in your `ViewController.swift` file by the code of Listing 5-3. Select an iPhone simulator from the Schemes. Press the Play button on Xcode's Toolbar to execute the app in the simulator. You will see the text “Basic Salary is 30000.0” at the top of the screen, as shown in Figure 5-15.

IMPORTANT: The `viewDidLoad()` method is called after the views are loaded in memory and before they are shown on the screen. This means that all the code defined inside this method is executed before the user is able to see the view. If the code takes time to finish, the user may have the impression that the application crashed. To avoid this situation, sometimes it is better to also implement the `viewDidAppear()` method and move some of the initialization code to this method. The method is called after the main view and its content is shown on the screen, and therefore from the user's perspective there is no lag or interruption. In further chapters we will see when the use of this and other initialization methods may be appropriate.

Creating the initial interface by adding subviews to the main view from our code, as we did in the last example, may be appropriate for very small applications, but as the interface grows and new elements and functionality are introduced, the code becomes difficult to develop and maintain. This is especially true with mobile applications, where we not only have to define the initial interface but also adapt it to new conditions produced by the

rotation of the screen or the app running on different devices. This problem was solved by Xcode with the introduction of Interface Builder.

5.3 Interface Builder

Interface Builder was actually the first visual tool introduced by Xcode. At the beginning, we were able to design the user's interface one main view at a time. Later, Storyboards were introduced to let us connect the main views together and develop the whole interface in one place. Now we use the Storyboard to create and connect the main views and Interface Builder to generate their content.

Object Library

Interface Builder is an integral part of Xcode; all we have to do to take advantage of this tool is click on the storyboard file and start dragging elements inside the views. The elements, including views, main views (also called view controllers), and all the `UIView`'s subviews studied in Chapter 4, are listed at the bottom of the Utilities Area (Figure 5-3, number 7) in a panel called *Object Library*. We can scroll down the list to see familiar elements like labels and buttons, or use the search field at the bottom to quickly find an element.

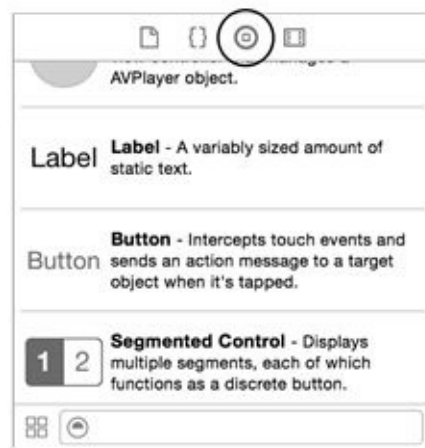


Figure 5-16: *Object Library*

The Basics: If you can't see the Object Library, click on the third button of the Toolbar at the top of the area (circled in Figure 5-16) or go to the View menu at the top of the screen, select the option Utilities, and click on Show Object Library.

The elements from the Object Library are incorporated into the view by dragging and dropping, as illustrated in Figure 5-17.

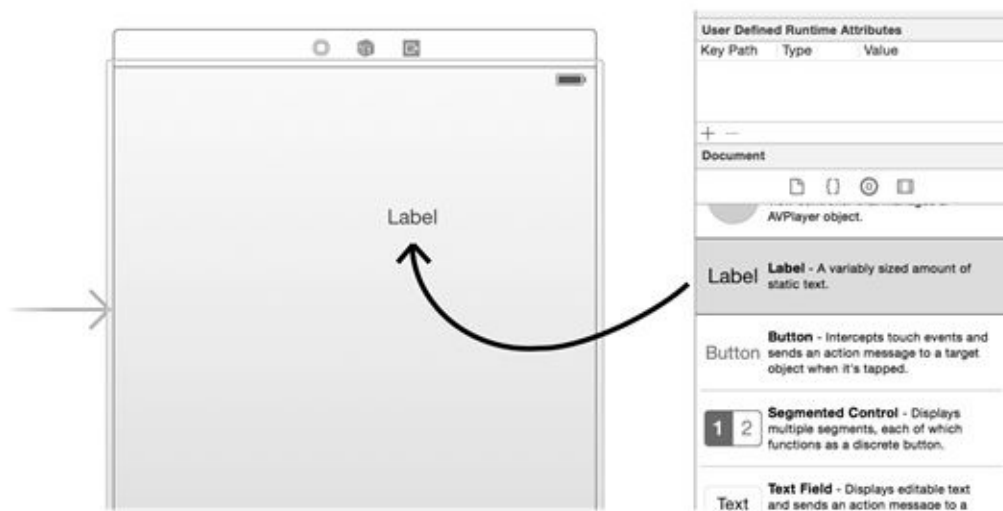


Figure 5-17: *Adding elements to the view*

Once the element is dropped inside the view it becomes the selected element. Selected elements are surrounded by little squares that we can drag to change the size of their views. We can also drag the element to a new position and select another element by clicking on it. Several elements may be selected at the same time by pressing and holding down the Shift key on the keyboard. Some elements, such as labels and buttons, allow us to double click on them to edit their content.

The element dropped inside a view becomes one of its subviews. It's the same process performed by the `addSubview()` method of the `UIView` object, with the difference that this time everything is done visually. All the views and subviews generated by this tool are stored as XML code in the storyboard file and then the real objects are reconstructed when the application is executed.

Guidelines

While the element is being dragged over a view, Xcode shows guidelines to help us find the right place to drop it. There are lines to indicate the horizontal and vertical center of the view, the superior, inferior and lateral margins, and also lines to indicate the position of one element relative to another. Figure 5-18 shows what we see when we drag a new label into the corner of the main view.

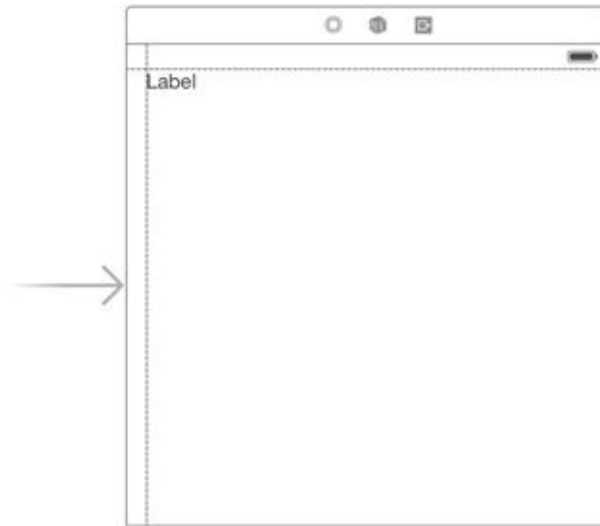


Figure 5-18: *View guidelines*

These lines become visible again when we move the elements around to find the right location and when we expand the area of the element's view by dragging the little squares around it.

Do It Yourself: Drag some of the elements you already know to the view in your Storyboard and increment the size of their views, position them side by side, or move them close to the border to understand how the guidelines work and how to build your interface.

IMPORTANT: For the examples of this chapter you will need to position the elements close to the left edge of the view to be able to see them on the screen of the simulator. We will study how to position elements and adapt the user interface to every device in Chapter 6.

Properties

As well as the main view, the elements in the interface may be selected and configured from the panels in the Utilities Area. Most of the properties studied in Chapter 4 are available from these panels. Figure 5-19 shows what the Attributes Inspector panel and the Size Inspector panel look like after a label is selected.

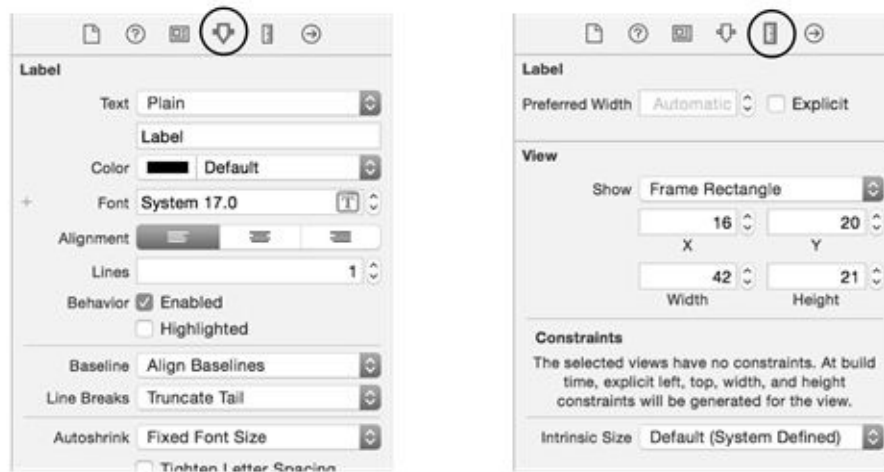


Figure 5-19: *Editing the properties of the label from the Utilities Area*

Most of the properties of an element are available from this panel. For example, we can modify the text, color, and font of a label. The panel even includes the styles for dynamic type fonts studied in Chapter 4, so we can create labels with a specific style that adapt to the size determined by the user.

Classes like `UILabel` are subclasses of the `UIView` class. Other similar classes like `UIButton`, for example, are subclasses of `UIControl`, which is a subclass of `UIView`. Therefore, they not only have access to their own properties, but also to other important properties they inherit from their superclasses, such as the `background` property of the `UIView` class we can use to set the element's background color, or the `enabled` property of the `UIControl` class we can use to enable or disable the element. All these options are also available on the Attributes Inspector panel under the sections Control and View.

Do It Yourself: Drag and drop several labels to the view. Select the labels one at a time and modify their values from the Utilities Area. When the text inserted for the label is longer than the size of its view, it gets truncated. If you want to show the whole text, you have to expand the label's view dragging the little squares around it or modifying the values in the Size Inspector.

5.4 Connections

As we already mentioned, the elements generated in the Storyboard and Interface Builder, including views, subviews, and their properties, are stored as XML code in the storyboard file. Right after the app is executed, the objects for the main views and their content are created from this code. The objects that represent the main views are connected to their view controllers through the `view` property, but the objects that represent the subviews inside these main views (labels, buttons, etc.) are not connected to our code in any way. They didn't exist when we were programming our app (they were just XML code in the storyboard file), so we can't reference these objects from our code. To solve this problem, Xcode introduces the concepts of Outlets and Actions.

Outlets

An Outlet is a property referencing the object that represents an element in the user interface. These are normal properties but prefixed by the `@IBOutlet` attribute to indicate to Xcode that they are related to an element in the Storyboard. When the app is executed, the objects are created from the Storyboard and then assigned to their Outlets in our code. From these Outlets we can read and set the values of their properties.

There are different ways to create an Outlet and generate the code for the connection. The most simple is to drag a line from the element in the Storyboard to our view controller and let Xcode generate the codes on both ends. For this, we need to divide the Editor Area in two parts by clicking the Assistant Editor button (Figure 5-3, number 3). Figure 5-20, next, illustrates what the Editor Area looks like after we open the storyboard file and click this button.

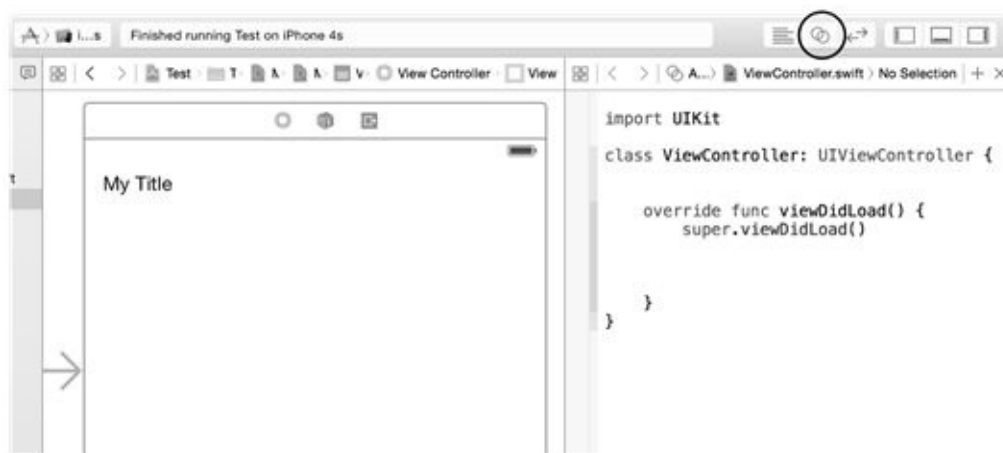


Figure 5-20: Editor Area with the Assistant Editor opened

When we click the Assistant Editor button (circled at the top of Figure 5-20), the Editor Area splits in two parts. The content of the file currently selected is shown on the left and the content of a related file is shown on the right. If the file opened on the left is the Storyboard, the file on the right will be the view controller of the currently selected main view. The connection between the elements in the main view on the left and our code on the right is established by dragging a line from the element to the class while pressing and holding down the Control key on the keyboard. Xcode generates a blue line on the screen to show the connection that is being created and to indicate where the property is going to be placed in our file when the mouse's button is released. The figure below is a capture of this process.



Figure 5-21: Connection between a label and its view controller

Outlets, as well as properties, should always be created at the beginning of the class. When we reach this place and release the mouse's button, Xcode shows a popup window to configure the connection.

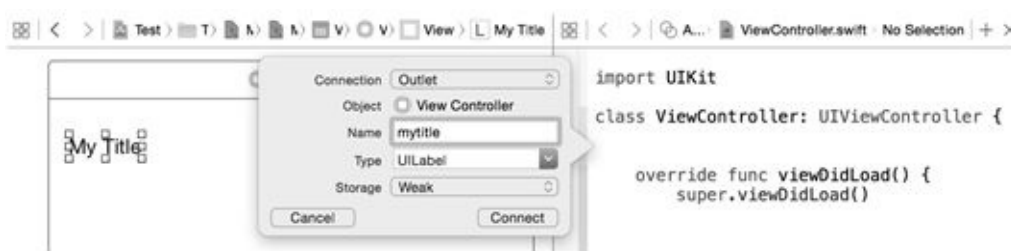


Figure 5-22: Configuration of the connection

The window asks for the type of the connection (Outlet or Action) and additional information to generate the code. To create an Outlet we have to select the Outlet connection, insert the name for the property, its data type, and the value that determines how the memory is going to be managed. The latter is usually specified as `weak`, because its container (the `UIView` object that represents the main view) already has a strong reference to the object (see Memory Management in Chapter 3).

In the example depicted in Figures 5-20, 5-21, and 5-22, a label with the text My Title

was added to the view and then connected to the view controller with an Outlet called **mytitle**. After the Connect button is clicked, Xcode generates the code for the Outlet. The following example shows what the code looks like when the Outlet is added to an empty view controller class.

Listing 5-4: Adding an Outlet for a label to the view controller

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var mytitle: UILabel!
    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

The code added to the class by the connection is a property prefixed by the **@IBOutlet** attribute. The attribute doesn't perform any task; it is there to indicate to Xcode that the property contains a reference to an element on the Storyboard. The attribute is followed by a keyword that determines the type of the reference (usually specified as **weak**) and the declaration of the property.

Do It Yourself: Drag and drop a label from the Object Library into the view in the Storyboard. Double-click the label to change its text to “My Title” and expand the label's view size if necessary to fit the text. Click on the button to open the Assistant Editor (Figure 5-20). Position the mouse over the label and press and hold down the Control key on your keyboard. Press the mouse's button and drag a line from the label to the beginning of the **ViewController** class. Release the mouse's button when you see something similar to Figure 5-21. A popup window similar to Figure 5-22 will show up. Define the Connection as Outlet, the name as **mytitle**, the type as **UILabel**, and the Storage as **weak**. Click the Connect button to generate the code. Your view controller should look like Listing 5-4.

IMPORTANT: There are some names that are already in use by the UIKit classes or are reserved by the Swift language and we can't use in our own properties or methods. For example, you can't give a property the name **title** because the **UIViewController** class already defines a stored property with that name. Every time there is a conflict with the name of a property or a method, Xcode will show you an error. Use more descriptive names such as **mytitle** or **titleLabel**.

Notice that the properties for Outlets are declared as implicitly unwrapped optionals.

We studied these types of optionals in Chapter 2. They are useful when we don't have the initial value for the variable or the property, but we know that once the value is assigned, they will never be empty again. This fits perfectly into the methodology applied to the creation of views and subviews. The objects that represent these views don't exist during development, but they are created later and assigned to the Outlets as soon as the app is executed. The reason why we used these types of optionals instead of regular optionals (declared with the question mark) is because we don't want to unwrap their values every time we have to use them. This is demonstrated in the following example where we access the `mytitle` property as if it were a normal `UILabel` property and change its color to red.

Listing 5-5: Modifying the label's properties

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var mytitle: UILabel!
    override func viewDidLoad() {
        super.viewDidLoad()
        mytitle.textColor = UIColor.redColor()
    }
}
```

When the app is executed, the objects from the Storyboard are created and a reference to the `UILabel` object that represents the label My Title is assigned to the `mytitle` property. As soon as the main view and the label are loaded into memory, the `viewDidLoad()` method is called. Inside this method we use the `mytitle` property to access the `textColor` property of the `UILabel` object and change the color of the label to red.

Connections in the Storyboard

When an Outlet or an Action is created, Xcode generates code in our view controller and also in the storyboard file. A common mistake we have to be aware of is to think that deleting the code for an Outlet or an Action in the view controller is enough to break the connection. This only deletes the connection in our code, but it does nothing to the code in the storyboard file. Interface Builder offers a shortcut menu to manage the element's connections and events. If we click the mouse's right button (also known as the Secondary click) when the pointer is over an element in the Storyboard, Xcode shows a popup window similar to the one illustrated in Figure 5-23.



Figure 5-23: Menu to manage connections on the storyboard file

This window shows all the Outlets and Actions created for the selected element. The name of the property is on the left and the name of the object is on the right. The name of the object includes a small x on the left that is highlighted by a circle when we hover the mouse over. Clicking on this x erases the corresponding XML code in the storyboard file and breaks the connection.

IMPORTANT: Always remember to erase the Outlets and Actions in your code and then the XML code in the storyboard file when you want to cancel a connection. If you don't need the element to be part of the interface anymore, you can delete it by selecting the element and pressing the Delete key on your keyboard. The XML code and all the connections are completely removed from the storyboard file.

Actions

Some elements are not limited to show graphic content on the screen; they can also process user interaction. We have studied some of them in Chapter 4. Buttons, Switches, Segmented Controls, and others are designed to show a visual response to the user and also fire an event to report the activity to the rest of the code. The way they do the latter is through a process called *target-action*. This is a programming pattern that allows an object to call a method on another object when a specific event occurs (e.g., a button is pressed, a switch is turned off, etc.). For this purpose, these elements implement a method inherited from the `UIControl` class called `addTarget()` (see `UIControl` class in Chapter 4).

The `addTarget()` method adds to its object a target and an action for a specific event. The action is the method to be called when the event occurs, the target is the object that the method belongs to, and the event is a specific type of activity that the user performs on the element (pressing a button, incrementing a value, etc.).

The implementation of this pattern requires an element that can perform an action (such as a button), an Outlet connected to that element (or a reference if the element is created programmatically), and a custom method to process the response. As an example, we are

going to insert a label and a button in our view using Interface Builder and declare a method in our view controller called `showCounter()` to count and print the times the button is pressed. Figure 5-24 shows what the view looks like after the addition of the label and the button.

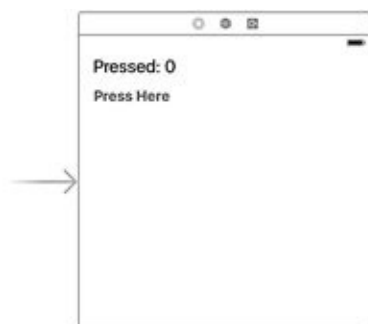


Figure 5-24: Adding a button

The text for the label and the title of the button were changed to “Pressed: 0” and “Press Here”, respectively. We can do this by double-clicking the elements or selecting each element and modifying their properties from the Attributes inspector panel in the Utilities Area (Figure 5-19). After adding the elements, they have to be connected to Outlets in our view controller.

Listing 5-6: Adding a target-action to a button

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var counterLabel: UILabel!
    @IBOutlet weak var pressHere: UIButton!

    var counter: Int = 0

    override func viewDidLoad() {
        super.viewDidLoad()

        pressHere.addTarget(self, action: "showCounter:", forControlEvents: UIControlEvents.TouchUpInside)
    }

    func showCounter(sender: UIButton) {
        counterLabel.text = "Pressed: \(++counter)"
    }
}
```

The first statement of the `ViewController` class of Listing 5-6 defines a property called `counter` to keep track of the times the button is pressed. Right after this property the Outlets

for the label and the button are declared with the names `counterLabel` and `pressHere`. Inside the `viewDidLoad()` method the `pressHere` property is used to call the `addTarget()` method on the `UIButton` object and add the target to the button. The syntax for this method is `addTarget(target, action, event)`. The target in this example was declared as `self` (the `ViewController` object), the action was declared as the `showCounter()` method defined below, and the event was specified as `TouchUpInside` (this event is fired after the button is pressed and released). What the `addTarget()` method is doing is telling the `UIButton` object that represents the button on the screen to call the `showCounter()` method in the `ViewController` object when the button is pressed and released. When this happens the `showCounter()` method assigns to `counterLabel` a string composed by the characters “Pressed: ” and the result of adding 1 to the current value of `counter` (the `++` operator adds 1 to the current value of the property and returns the new value afterwards).

Do It Yourself: Delete all the elements you have added to your view by selecting them and pressing the Delete key on your keyboard. Clean the code in your view controller, including the Outlets. Drag and drop a label and a button from the Object Library into the view. Double-click the label and the button to change the text and title. Expand the label’s view size if necessary to fit the text. Create an Outlet for both elements and call them `counterLabel` and `pressHere`. Copy the rest of the code of Listing 5-6 into the `ViewController` class. Run the application in the simulator. Every time the button is pressed you should see the number on the label incremented by one unit.

IMPORTANT: The value assigned to the **action** attribute is called *Selector*. Selectors are just the name of the method we want to execute. They may be declared as strings, as we did in Listing 5-6, or using the `Selector` instruction, as in `Selector(“nameofmethod”)`. We can’t send any value to the method from a selector, but the methods that use selectors send a reference to the object that performed the action. For example, the `addTarget()` of Listing 5-6 sends a reference to the `UIButton` object that triggered the event. To send this attribute we have to write a colon at the end of the method’s name and then specify the parameter in the method’s declaration (`sender: UIButton`).

The Basics: The types of events are defined as constants in the `UIControlEvents` structure. There are several constants available, but the most frequently used are `TouchUpInside` (a button is pressed and released) and `ValueChanged` (the value of the control changes). For a complete list, visit our website and follow the links for this chapter.

As illustrated by the code of Listing 5-6, to add a target and respond to an event the view controller has to include an Outlet for the control, call the `addTarget()` method to set the target for the object, and define the method that responds to the action. This protocol is always the same, only the method that processes the response differs every time. This is

the reason why Xcode introduced Actions.

Actions, like Outlets, are connections between the elements on the Storyboard and our code, but unlike Outlets, they connect the elements to methods, not properties. The purpose is to move the definition of the action and the target to the Storyboard. When a control is connected to our code by an Action, the definition of the `addTarget()` method is moved to the Storyboard and we only have to define in our view controller the method that will produce the response.

Adding an Action for an element is similar to adding an Outlet. We have to drag a line from the control to the view controller class while pressing and holding down the Control key and then provide the information for the connection.

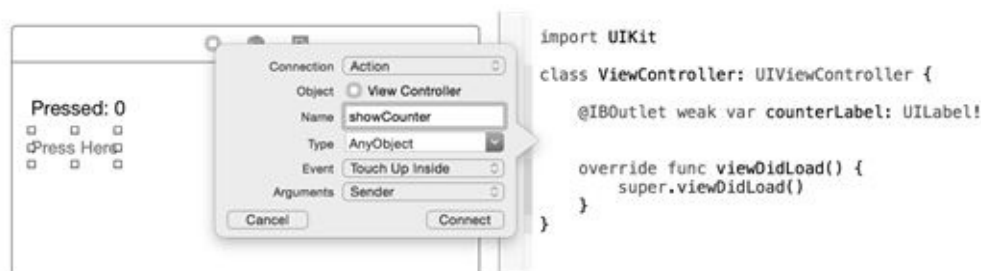


Figure 5-25: *Configuring the Action*

The first thing we have to do in the popup window is to select the type of the connection as Action. The options to configure the Action are similar to those defined for the `addTarget()` method. We need to insert the name of the method, the type of the object, and the type of event that will trigger the action. The last option, Arguments, determines the values we want to send to the method. The option includes three possible values: None (no value is sent to the method), Sender (a reference to the object is sent to the method), and Sender and Event (a reference to the object and information about the event that triggered the action are sent to the method). By default the value is set to Sender, which will create a parameter for the method called `sender` to receive the reference to the object.

IMPORTANT: Selecting the data type of the object is important when we need to access the object's properties and methods. If you set this value as `AnyObject`, multiple elements can be connected to the same Action, but Swift won't be able to find the properties and methods of specific types like `UIButton`.

After all the information is declared, Xcode generates the codes in both ends. The declaration of the method in our view controller is similar to the method in the previous example, but this time it is prefixed by the `@IBAction` attribute to establish its relationship with the element on the Storyboard. The rest of the code for the class remains the same.

Listing 5-7: *Defining an Action for the button*

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var counterLabel: UILabel!
    var counter: Int = 0

    @IBAction func showCounter(sender: UIButton) {
        counterLabel.text = "Pressed: \(++counter)"
    }
}
```

From this example we can clearly see the advantage of using Actions. The code of Listing 5-7 has been greatly simplified. This time the objects in the Storyboard take care of defining the action and the target, and we only have to declare the method that is going to be executed when the event is fired. The process produced by this code is exactly the same as the previous example. When the button is pressed, the `UIButton` object created from the Storyboard calls the `showCounter()` method in our view controller and the text in the `counterLabel` label is updated.

Do It Yourself: Delete the elements inside the view by selecting them and pressing the Delete key on your keyboard. Clean the code in your view controller, including the Outlets. Drag and drop a new label and a button from the Object Library into the view (Figure 5-23). Double-click the label and the button to change the text and title. Create an Outlet for the label called `counterLabel`. Create an Action for the button with the name `showCounter()`, the event Touch Up Inside, and keep the rest of the values by default. Complete the `ViewController` class with the code of Listing 5-7. Run the application in the simulator. Every time the button is pressed you should see the number on the label incremented by one unit.

IMPORTANT: The use of methods like `viewDidLoad()` is not required. We only have to overwrite them when we need to perform a task or initialize information during their execution. In the example of Listing 5-7 (and the following examples), we didn't include this method because the process is done by the method created for the Action. The requirement of this and any other `UIViewController` method depends on your application.

Actions are the best way to prepare our code to process user interaction. Every control defined in UIKit can process this type of connection, although the events fired are not always the same. For example, the Segmented Control fires an event called *ValueChanged* every time the user selects a new segment. Figure 5-26, next, shows what our

view looks like when the button is replaced by this type of control.

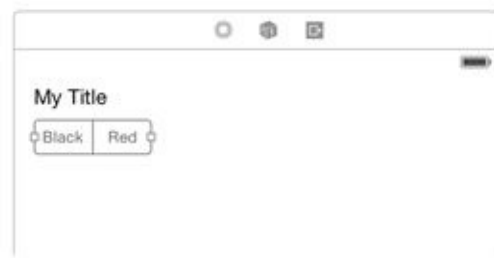


Figure 5-26: *Segmented Control with two options*

The Segmented Control used in this example was defined with two segments (buttons) with the titles Black and Red, representing colors for the label. This is the number of segments presented by default, but we can add more from the Attributes inspector panel in the Utilities Area.

The way this control works is by keeping track of which segment is currently selected. An index value is assigned to each segment in correlative order starting from 0. When a new segment is selected, the value of its index is assigned to the `selectedSegmentIndex` property of the `UISegmentedControl` object that represents the element and the Value Changed event is fired. By creating an Action that responds to this event we can check the `selectedSegmentIndex` property and execute the corresponding code according to its value.

Listing 5-8: *Responding to the Value Changed event of a Segmented Control*

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var mytitle: UILabel!

    @IBAction func selectColor(sender: UISegmentedControl) {
        if sender.selectedSegmentIndex == 0 {
            mytitle.textColor = .blackColor()
        } else if sender.selectedSegmentIndex == 1 {
            mytitle.textColor = .redColor()
        }
    }
}
```

The `ViewController` class of Listing 5-8 defines a method called `selectColor()` assigned as the Action for the Segmented Control illustrated in Figure 5-25. When the user selects a new segment by pressing the button, the Value Changed event is fired and the `selectColor()` method is executed. Inside this method the value of the `selectedSegmentIndex` of the `sender`

object is checked and the color of the label is changed accordingly (black for the segment at index 0 and red for the segment at index 1).

Do It Yourself: Delete the elements inside the view and clean the code in your view controller, including the Outlets and the Actions. Drag and drop a label and a Segmented Control from the Object Library into the view on the Storyboard (as illustrated in Figure 5-26). Double-click on the label and each segment to change the text and the titles. Create an Outlet for the label called `mytitle` and an Action for the control. In the popup window for the Action, select the Action type, define the name for the method as `selectColor`, select the type for the value to be sent to the method as `UISegmentedControl`, select the Value Changed event and the Sender argument. Click the Connect button to generate the connection and copy inside the method the statements presented in Listing 5-8. After executing the application you will be able to change the color of the label by pressing the buttons.

A similar procedure can be performed by the Switch control. This type of control has always two positions: on or off. By checking the value of its `on` property we can determine the current state and perform the operations we want. In the following example, if the `on` property is `true`, the color of the label is changed to black, otherwise it is changed to red.

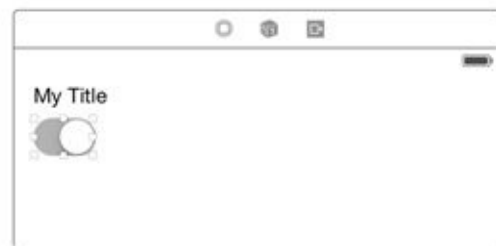


Figure 5-27: Switch on the Storyboard

Listing 5-9: Changing the color with a switch

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var mytitle: UILabel!

    @IBAction func switchColor(sender: UISwitch) {
        if sender.on {
            mytitle.textColor = .blackColor()
        } else {
            mytitle.textColor = .redColor()
        }
    }
}
```

```
}  
}
```

Do It Yourself: Delete the elements inside the view and clean the code in your view controller, including the Outlets and the Actions. Drag and drop a label and a Switch from the Object Library into the view in the Storyboard (as illustrated in Figure 5-27). Create an Outlet for the label called `mytitle` and the Action for the switch. In the popup window for the Action, select the Action type, define the name for the method as `switchColor`, select the type for the value to be sent to the method as `UISwitch`, select the Value Changed event and the Sender argument. Click the Connect button to generate the connection and copy inside the method the statements presented in Listing 5-9. Turning the switch on and off should change the color of the label.

Other controls that fire the Value Changed event to perform an Action are the Stepper (`UIStepper`) and the Slider (`UISlider`). The Stepper lets the user increment or decrement a value by a specific number of units in a predetermined range. The following example adds a Stepper to the view and a label to show the current value.



Figure 5-28: Stepper in the Storyboard

As with any other element, the values of the Stepper's properties may be modified from the Attribute inspector panel. By default, the range goes from 0 to 100, it starts counting from 0 and the value is incremented by 1. For this example, we leave the values by default and define an Action that takes the current value from the `value` property of the `UIStepper` object and assigns it to the label.

Listing 5-10: *Incrementing or decrementing a value with a Stepper*

```
import UIKit  
  
class ViewController: UIViewController {  
    @IBOutlet weak var counterLabel: UILabel!  
    @IBAction func increment(sender: UIStepper) {  
        counterLabel.text = String(sender.value)  
    }  
}
```

}

Do It Yourself: Delete the elements inside the view and clean the code in your view controller, including the Outlets and the Actions. Drag and drop a new label and a Stepper from the Object Library into the view on the Storyboard. Double-click the label to change its text to “0.0”, as illustrated in Figure 5-28. Create an Outlet for the label called `counterLabel` and an Action for the Stepper called `increment()` with the same options used before. Click the Connect button to generate the connection and copy inside the method the statement presented in Listing 5-10. The value in the label should be incremented or decremented every time you press the buttons of the Stepper.

The Slider is slightly similar to the Stepper, but its value is incremented or decremented according to how far the pointer is displaced. This type of control is usually applied to analog systems where a specific value is not required (e.g., volume control, brightness, etc.).



Figure 5-29: A Progress View controlled by a Slider

For this example we use a Progress View element to show the values selected from the Slider in a progress bar. When the user moves the pointer of the Slider, the value in its `value` property changes and the `increment()` method is executed. Inside this method, the value of the `value` property is assigned to the `progress` property of the `UIProgressView` object to update the progress bar.

Listing 5-11: Incrementing or decrementing a value with a Slider

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var progress: UIProgressView!
    @IBAction func incrementSlider(sender: UISlider) {
        progress.progress = sender.value
    }
}
```

Do It Yourself: Delete the elements inside the view and clean the code in your view controller, including the Outlets and the Actions. Drag and drop a Progress View and a Slider from the Object Library into the view on the Storyboard (as illustrated in Figure 5-29). Create an Outlet for the progress bar called `progress` and an Action for the Slider called `incrementSlider()`. Click the Connect button to generate the connection and copy inside the method the statement presented in Listing 5-11. The progress bar should change according to the position of the Slider.

The Basics: The progress bar and the Slider work by default with the same range of values (0.0 to 1.0). If you change the range for the Slider you will have to adapt the number returned by this element to the Progress View range to be able to show the correct level of progress.

Outlet Collections

An Outlet Collection is a type of Outlet that can reference one or more elements. It is helpful when several elements of the interface have to be modified at the same time. The property created for the Outlet is an array containing references to each element. Iterating through the array we can reach every element connected to the Outlet and access its properties.

The following example presents a very simple interface with three buttons. The Say Hello and Say Goodbye buttons print messages on the console and the Disable button is used to disable the first two.

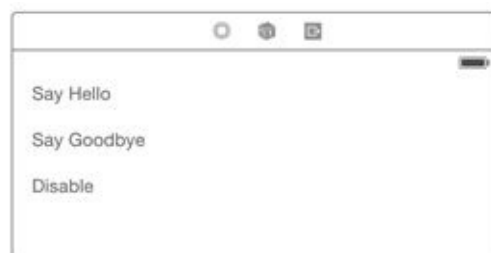


Figure 5-30: A simple interface to test Outlet Collections

To create the functionality for the interface we have to define an Action for every button and an Outlet Collection for the first two. The Outlet Collection is created dragging a line from the elements to the view controller one by one. The first connection asks for the name of the Outlet, as always, but the subsequent Outlets are created by pointing the mouse to the definition of the property, as shown in Figure 5-31.

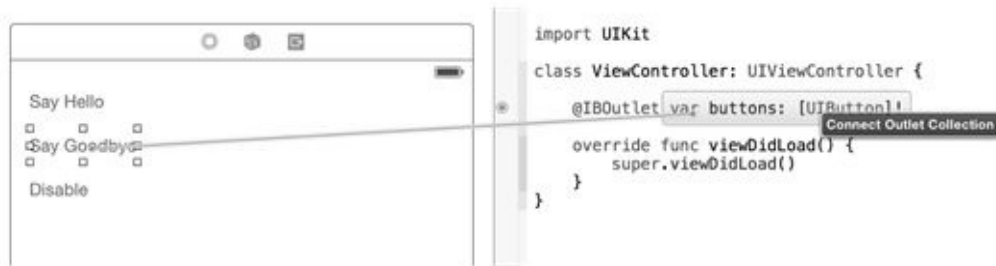


Figure 5-31: The subsequent Outlets have to be connected to the definition of the property

After the Outlet Collection is defined, we have to create an Action for every button. Listing 5-12, next, shows the view controller with the connections and the code for every method.

Listing 5-12: Implementing an Outlet Collection

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet var buttons: [UIButton]!

    @IBAction func sayHello(sender: AnyObject) {
        print("Hello!")
    }

    @IBAction func sayGoodbye(sender: AnyObject) {
        print("Goodbye!")
    }

    @IBAction func disableButtons(sender: AnyObject) {
        for button in buttons {
            button.enabled = false
        }
    }
}
```

The methods for the Say Hello and Say Goodbye buttons (`sayHello()` and `sayGoodbye()`) print messages on the console, but the method for the Disable button (`disableButtons()`) takes advantage of the `enabled` property inherited by the `UIButton` class from the `UIControl` class to deactivate the other two buttons (see `UIControl` in Chapter 4). To access the property of every button we iterate through the `buttons` array produced by the Outlet Collection. Every time the Disable button is pressed, the other two buttons are disabled (they don't respond to user interaction anymore).

Do It Yourself: Delete previous elements and code in your project. Drag and drop three buttons to the view to create the interface of Figure 5-29. Connect the first two buttons to an Outlet Collection called `buttons` (the type of the connection has to be selected as Outlet Collection in the popup window). Create the Actions for every button and add the code of Listing 5-12 to every method. Every time you press any of the first two buttons, a message is printed on the console. Press the third button to disable the first two.

Real-Life Application

The same way there are Outlet Collections to connect several elements to the same Outlet, there is also the possibility to connect several elements to the same Action (the same method). The following example builds a calculator to demonstrate how to implement these types of Actions and how to program an application with the features studied so far.

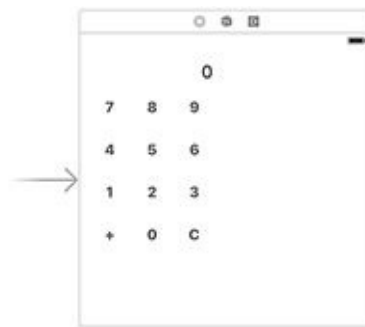


Figure 5-32: Interface for a basic calculator

Our calculator's interface is depicted in Figure 5-32 above. We need one label at the top to represent the display (to show the input and the results), and also 12 buttons to let the user insert new numbers, add the values, erase the counter, and clean the display (our calculator only adds integers). The label's view is wide enough to fit 12 digits and its text is aligned to the right. The values for the properties we need to modify to get this design are available in the Attribute Inspector panel (Figure 5-19). To get the same design of Figure 5-32 we just have to drag and drop the elements to the view and modify their properties.

The Basics: The Storyboard allows you to copy and paste elements in the view. To create the buttons for the calculator you can drag and drop one button, change the title to the corresponding number and then copy and paste the button (pressing Command + C and Command + V on your keyboard). Position the new button at the right place, change its title and repeat the process until you have created all the buttons you need. You can also drag an element while holding down the Option key on your keyboard

to create a duplicate.

The buttons for the numbers (0 to 9) are all connected to one single Action we called `addNumber()`. We have to create this Action from one button and then drag the line from the rest of the buttons to this same method (the whole method turns blue to indicate the connection). The challenge presented by various buttons calling the same method is how to identify which button performed the call. There are at least two ways to do this. We could read the `currentTitle` property of the `sender` parameter (the `UIButton` object) and check its value to detect which button was pressed, or we can use the `tag` property. Every subclass of `UIView` inherits the `tag` property from its superclass. This property stores a value of type `Int` to identify the view. Assigning an integer value to this property we are able to identify the view later. For this purpose, we have to define the value of the `tag` property of every button from the Attributes Inspector panel. We have to select a button and scroll down this panel until we find the Tag value (in our example, we only need to do this for the buttons with numbers). We can assign the same number of the button's title to its `tag` property (number 0 for button 0, number 1 for button 1, and so on), as shown in Figure 5-33.



Figure 5-33: Properties for the buttons

The rest of the buttons (+ and C) are connected to their own Actions to perform the addition and clean the display, respectively. Listing 5-13, next, presents the view controller with the necessary Outlets, Actions, and code to control the whole process.

Listing 5-13: Programming the calculator's view controller

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var display: UILabel!
    var current: Double = 0
    var total: Double = 0
    let digits = 12
    var format = NSNumberFormatter()

    @IBAction func addNumber(sender: UIButton) {
```

```

let number = Double(sender.tag)
let text = display.text!
if current == 0 {
    current = number
} else if text.characters.count < digits {
    current = (current * 10) + number
}
display.text = format.stringFromNumber(current)
}

@IBAction func calculate(sender: UIButton) {
    total += current
    display.text = format.stringFromNumber(total)
    current = 0
}

@IBAction func clean(sender: UIButton) {
    current = 0
    total = 0
    display.text = "0"
}
}

```

The code starts by defining the necessary properties to store the numbers and show them on the display. The **current** property stores the current value typed by the user (one digit at a time), the **total** property stores the result of the last operation, the **digits** property indicates the maximum number of digits available on the display, and the **format** property is an **NSNumberFormatter** object to format the numbers before they are printed on the screen.

The view controller is completed by the **display** Outlet for the label and three Actions: **addNumber()**, **calculate()**, and **clean()**. The **addNumber()** method adds the number of the button pressed by the user to the value of the **current** property, the **calculate()** method adds the current number inserted by the user to the previous result, and the **clean()** method cleans the display and the values of the properties to start again from scratch.

The calculator adds a new integer number to the previous result. This only presents one small complication. The number inserted by the user is not always the final value. This is because the number is inserted in sequence and it may be composed by more than one digit. For example, when the user presses the button 5 and later the button 4, the value to be processed is 54. We have to add the number 4 to the number 5 as a new digit to get the final number. The **addNumber()** method takes care of this by calculating and storing the total value in the **current** property. First, it reads the value of the **tag** property, converts it to a **Double** type, and stores it in the **number** constant (**let number = Double(sender.tag)**). The conversion is necessary because the **tag** property returns an integer value and we are working with properties of type **Double** (operations can only be performed on values of the same type). The **number** constant is used next to calculate the total value inserted by the

user. If the value of `current` is equal to 0, the number is assigned directly to the `current` property; otherwise we have to check if the digits already inserted are not over the limit. This limit was defined in the `digits` property. We count the amount of digits in the display using the `count` property and compare it with the value of `digits` (`text.characters.count < digits`). If the current digits are less than the limit, we add the value of the `number` property as a new digit of the `current` property (the new digits are added by multiplying the current value times 10 (`current = (current * 10) + number`)). At the end, the new value of the `current` property is processed by the `stringFromNumber()` method of the `format` object to eliminate any decimal numbers and the string returned is assigned to the label to show the new value on the display (`display.text = format.stringFromNumber(current)`).

The Basics: The `format` object was configured with values by default. This configuration rounds the decimal number to the nearest integer and returns a string containing an integer value. For more information on `NSNumberFormatter`, see Chapter 4.

The codes for the `calculate()` and `clean()` methods are simpler. The `calculate()` method is executed when the `+` button is pressed to perform the addition. This method adds the value of the `current` property to the value of the `total` property and shows it on the calculator's display. When this is over, it assigns the value 0 to the `current` property to get it ready to receive new values. On the other hand, the `clean()` method is executed when the `C` button is pressed. All we have to do here is to clean the display and the properties to start all over again.

Do It Yourself: Delete any element in your view and the code in your `ViewController` class from previous examples. Drag and drop a label and all the buttons necessary to get the design illustrated in Figure 5-32 (position everything close to the left edge of the view to be able to see it on any version of the simulator). Expand the label's view to fit at least 12 characters, align its text to the right and change it to "0". Modify the titles of the buttons to represent the buttons of the calculator. Generate the Outlet for the label and the Actions for the buttons with the names declared in Listing 5-13 (the buttons for the numbers are all connected to the `addNumber()` method). Select the buttons for the numbers one by one and change the value of their `tag` property in the Attributes Inspector panel to match the numbers in their titles (Figure 5-33). Complete the `ViewController` class with the code of Listing 5-13. Run the application. Every time you want to add a new value or see the result you have to press the `+` button.

Taking advantage of what we already know, we can modify the elements to get a more professional design. For example, the design shown in Figure 5-34 was achieved by adding a few extra views to delimit the display and the whole calculator.

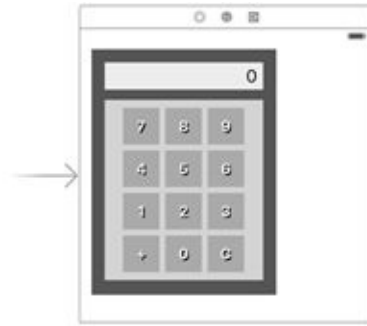


Figure 5-34: *A better design for the calculator*

This design includes three empty views: one with a dark background to represent the calculator's box, another with a lighter background to delimit the area of the buttons, and a smaller one to represent the display. The label and the buttons were added as subviews of the smaller views, and the smaller views were added in turn as subviews of the view representing the box. The color for the titles of the buttons was changed to white and their background color to gray. There is also a black shadow generated for the titles to create a simple tridimensional effect. All the properties to apply these styles are available in the Attributes Inspector panel.

Empty views are included in the list of elements offered by the Object Library, as shown in Figure 5-35. When a new view is dragged to the Storyboard, it takes the size of its container, but we can easily change its values from the Size Inspector panel or dragging the little squares around it.

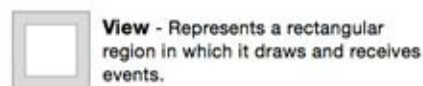


Figure 5-35: *Views in the Object Library*

IMPORTANT: All the projects presented in this book and links to videos illustrating the procedures are available in our website. Go to www.formasterminds.com and click on Videos or Sample Codes.

Chapter 6

Adaptivity

6.1 Adaptivity

Adaptivity is a term that describes the capacity of the user interface to adapt to changes in the size of the screen, either because the app is running on devices with different screens or the device has been rotated. Adaptivity is just an idea, accomplished by the combination of old and new systems that turn the Storyboard into a multipurpose graphic tool. The Storyboard is still the centerpiece of this intricate system, but it is complemented by the combination of two technologies called *Auto Layout* and *Size Classes*. *Auto Layout* is an improved old system that lets us define a set of rules to determine how the elements are going to be laid out inside the main view. *Size Classes*, on the other hand, is a new system that allows us to switch to different versions of the Storyboard to prepare the interface for all available screen sizes. Combining these tools we are now able to develop an interface that adapts to multiple devices and orientations.

6.2 Auto Layout

When an element is added as a subview of another view, it has a position and size determined by the values of its `frame` property. Every time the app is executed, these values are used by the system to know where the element should be drawn on the screen. This is an intuitive way to organize the interface but presents a problem. If the screen is rotated or the app is executed in a different device, the characteristics of the screen change but the values of the subview remain the same. Figure 6-1 shows what happens in these situations.

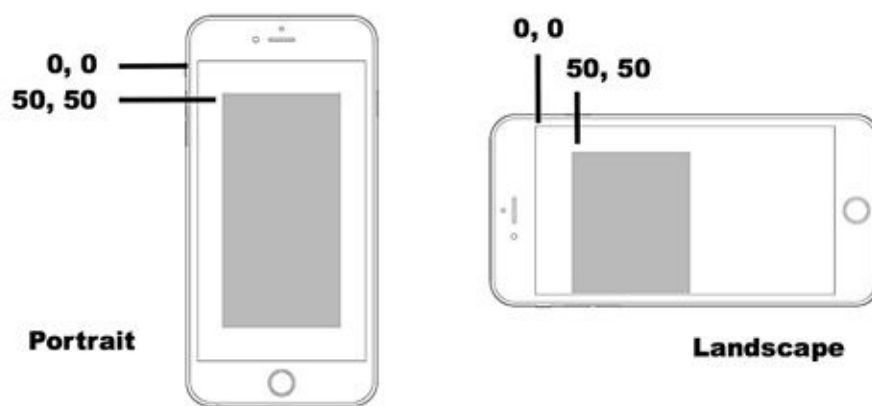


Figure 6-1: View positioned with absolute values

In this example, the gray view is added to the main view with values according to the vertical orientation (portrait). When the device is rotated, the values of the position and size of the view remain the same, but the coordinate system changes because of the rotation. In landscape orientation, half of the gray box generated by the view disappears at the bottom of the screen. Its coordinates are the same, but the horizontal and vertical spaces available are not. Assigning new values to the `frame` property every time something changes is one possible solution, but it also introduces a difficult challenge. All the values for every element of the interface have to be modified not only when the screen is rotated but also for every type of screen available. And the list of possible scenarios grows every time a new device enters the market. Looking for a solution to this dilemma, Apple developers realized that instead of using absolute values it was better to determine how the views have to be laid out and let the system calculate the positions and sizes. The result of their work is a system called Auto Layout.

Auto Layout is a system that uses a set of rules to organize the elements of the user interface. The rules establish the relationship between elements and between the elements and their container. They are simple statements that determine things like how far a view should be from another view, or how a set of views should be aligned. As an example, Figure 6-2, next, presents a container with a button in a position and with a size determined by four rules. These specific rules tell the system how far the button should be from the sides of its container. The example on the left is what we would see during

implementation, when all the rules are assigned to the element. The illustration on the right represents these same elements during execution, when the position and size of the button have already been modified to abide to the rules.

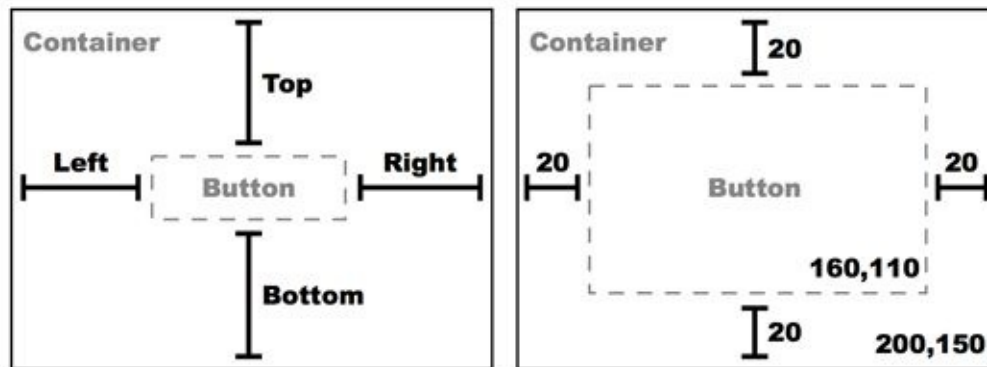


Figure 6-2: Rules applied to a button

The rules applied to the button depicted on the right side of Figure 6-2 are saying things like “this button has to always be 20 points from the top of its container” and “this button has to always be 20 points from the left side of its container”. Every rule determines one aspect of the relationship between the button and its container to help the system define its position and size. Because of this, we no longer have to declare explicit values to layout the elements; they are calculated by the system after all the rules have being followed.

IMPORTANT: The values assigned to the `frame` property of every view in the Storyboard are not considered by the system anymore. Assigning new values to this property has no effect on the position and size of the element on the screen. But there are situations in which the values of this property are still useful. We will see some examples in further chapters.

Constraints

Computers cannot understand human expressions like “position the view 50 points from the edge of the container”. For this reason, Auto Layout defines a set of specific rules that we can configure and combine to organize the interface. Auto Layout rules are called *Constraints*, and they are comprehensive enough to let us create any design we want.

There are two main groups of constraints: Pin constraints and Align constraints. Pin constraints define space, such as the space between a view and its container, while Align constraints define how the views should be aligned with respect to other views or their containers. The following list describes the Pin constraints available:

Top Space—This constraint defines the space between the top of the element and its nearest neighbour. If there is no other element at the top, the limit is determined by the edge of the container.

Bottom Space—This constraint defines the space between the bottom of the element and its nearest neighbour. If there is no other element at the bottom, the limit is determined by the edge of the container.

Leading Space—This constraint defines the space between the left side of the element and its nearest neighbour. If there is no other element on the left, the limit is determined by the edge of the container.

Trailing Space—This constraint defines the space between the right side of the element and its nearest neighbour. If there is no other element on the right, the limit is determined by the edge of the container.

Width—This constraint defines the width of the element.

Height—This constraint defines the height of the element.

Aspect Ratio—This constraint defines the aspect ratio of the element.

Equal Widths—This constraint declares that two elements will have the same width. If the width of one of the elements changes, the same value is assigned to the other element affected by the constraint.

Equal Heights—This constraint declares that two elements will have the same height. If the height of one of the elements changes, the same value is assigned to the other element affected by the constraint.

The following are the constraints for alignment:

Horizontal in Container—This constraint aligns the center of the element with the center of its container in the horizontal axis.

Vertical in Container—This constraint aligns the center of the element with the center of its container in the vertical axis.

Leading Edges—This constraint aligns two elements to their left side.

Trailing Edges—This constraint aligns two elements to their right side.

Top Edges—This constraint aligns two elements to their top edge.

Bottom Edges—This constraint aligns two elements to their bottom edge.

Horizontal Centers—This constraint aligns two elements to their horizontal center.

Vertical Centers—This constraint aligns two elements to their vertical centers.

Baselines—This constraint aligns two elements to their baselines.

Most of these constraints are straightforward, but some of them adapt to the circumstances. For example, the constraints that affect the left or right side of the elements are called *Leading* and *Trailing* because they affect a side according to the language set in the system. In Arabic languages, for example, the leading space is on the right side of the element and the trailing space is on the left. Also, each constraint was defined to affect a specific number of elements. There are constraints for one element and others that associate two elements together, but we can generate several constraints for multiple elements at the same time by selecting the elements and assigning the constraints from the tools provided by Xcode.

Assigning Constraints

Every element on the interface requires constraints to define its position and size. Xcode offers a series of buttons at the bottom of the Storyboard to add and manage constraints. Figure 6-3 illustrates all the options available.



Figure 6-3: Buttons to work with Auto Layout and Size Classes

These buttons have different functions, from adding constraints to helping us in solving the issues that come up along the way. The following list describes the options available.

Document Outline (1)—This button opens a panel on the left side of the Editor Area with a list of the elements inside the main views (view controllers) and their associated constraints. From this list we can edit the names, select and modify the constraints, or even generate new ones.

Size Classes (2)—This button displays a small window to set the Storyboard according to a specific Size Class (we will study Size Classes in the next section of this chapter).

Align (3)—This button opens a quick access menu with options to add constraints for alignment.

Pin (4)—This button opens a quick access menu with options to add constraints for space.

Resolve Auto Layout Issues (5)—This button opens a quick access menu with options that help us solve common constraints issues.

To add a constraint to an element we first have to select the element and then click the button to open the Align or Pin menus (buttons 3 and 4 in Figure 6-3). Figure 6-4 shows the menus displayed by these buttons.

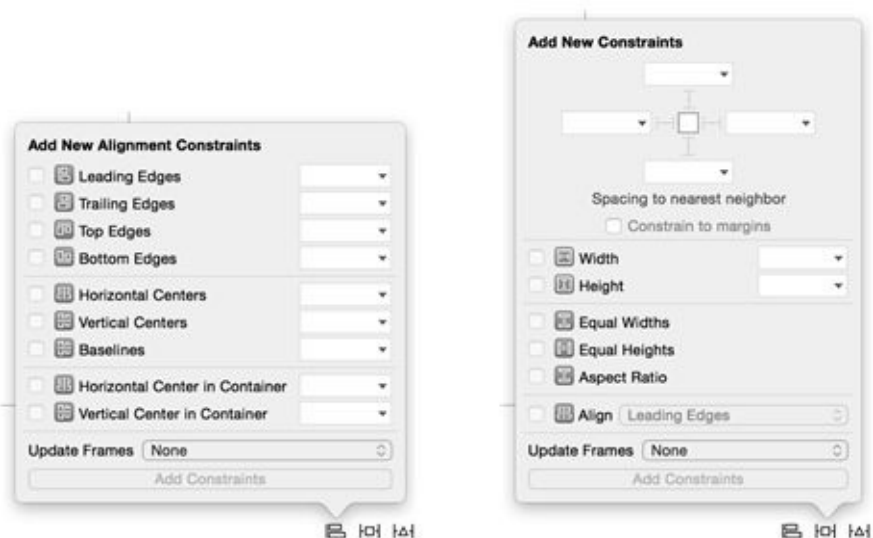


Figure 6-4: Align and Pin menus

The menus list all the constraint types described above, but they adjust the options to the types and amount of elements selected. For example, those constraints that require two elements are disabled when only one element is selected. After we activate the constraint we want to add, the Add Constraints button at the bottom of the menu is highlighted. Clicking this button adds the constraint to the view. Figure 6-5 shows the process of adding a Top Space constraint to a view.

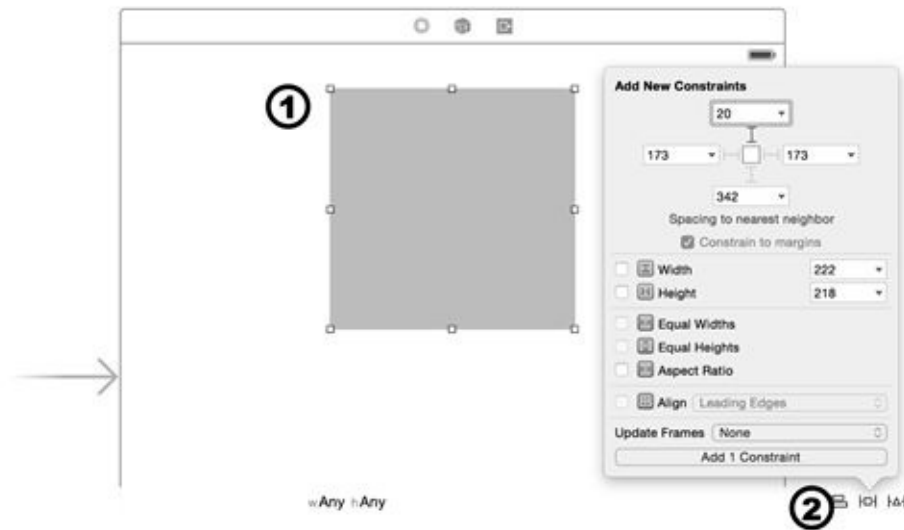


Figure 6-5: Adding a Top Space constraint to a single view

These types of constraints are added from the Pin menu. We have to select the view (Figure 6-5, number 1), click the Pin button at the bottom of the Editor Area to open the Pin menu (Figure 6-5, number 2), and activate the constraint we want to add. For space constraints, the options are at the top of the menu, as shown in Figure 6-6.

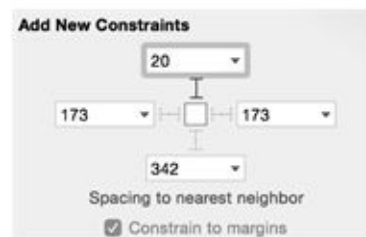


Figure 6-6: Menu for Space constraints

Space constraints are selected by clicking the red line between their value and the small square in the middle that represents the view (Figure 6-6). The value for the constraint is set by default as the value defined in the Storyboard, but we can change it to any value we want. In our example, we have activated the Top Space constraint with a value of 20. This means when the system reads this constraint it will position the top of the view 20 points from its nearest neighbour. If there are no other elements at the top, the distance is determined from the top of the element to the top edge of the container. One or more space constraints may be activated at the same time, but we always have to click on the red lines first to activate the constraints before modifying their values.

After the constraints have been selected and configured, we have to press the Add Constraints button at the bottom of the menu to effectively add the constraints to the view. The menu is closed and Xcode shows lines on the Storyboard representing the constraints. Every time an element is selected, the lines that represent its constraints are shown on the screen, as illustrated in Figure 6-7.

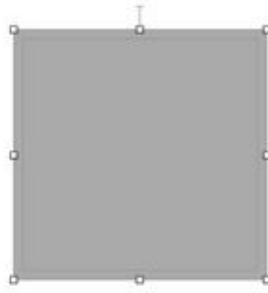


Figure 6-7: Subview with a Top Space constraint

Do It Yourself: Create a new Single View Application project. Click on the storyboard file to show the Storyboard in the Editor Area. Drag a view from the Object Library to the main view. Assign a background color, a size, and a position similar to the view depicted in Figure 6-5. With the view selected, open the Pin menu and add a Top Space constraint (Figure 6-6). Keep the value for the constraint as suggested by Xcode (this is the current distance between your view and the top of the main view). You should see the lines representing the constraints as shown in Figure 6-7. If you want the distance to be between the view and the edge of its container, deactivate the option *Constrain to margins*.

The Basics: The Pin and Align menus allow the selection of multiple constraints. The title of the Add Constraints button is automatically updated to reflect the number of constraints selected. When you press this button, all the selected constraints are added at once.

Although a constraint determines a specific rule, one constraint alone can't provide enough information for the system to determine the position and size of the element. When this happens, Xcode shows the constraint on the Storyboard in orange. The color indicates that the constraints assigned to the element are ambiguous or insufficient. There are a total of three colors: blue, orange and red. The constraints are shown in blue when the system has enough information to display the element, orange when there is not enough information or the information is ambiguous, and red when some constraints are in conflict (the system can't satisfy all the constraints at the same time).

In our example, the constraint is orange because the system knows about the vertical position of the view (the value of the *y* coordinate), but we didn't specify yet how it should be positioned horizontally and how its size should be determined. There are several options available, depending on what we want to do. For instance, if we want to position the view in the center of the main view, we have to add an alignment constraint.

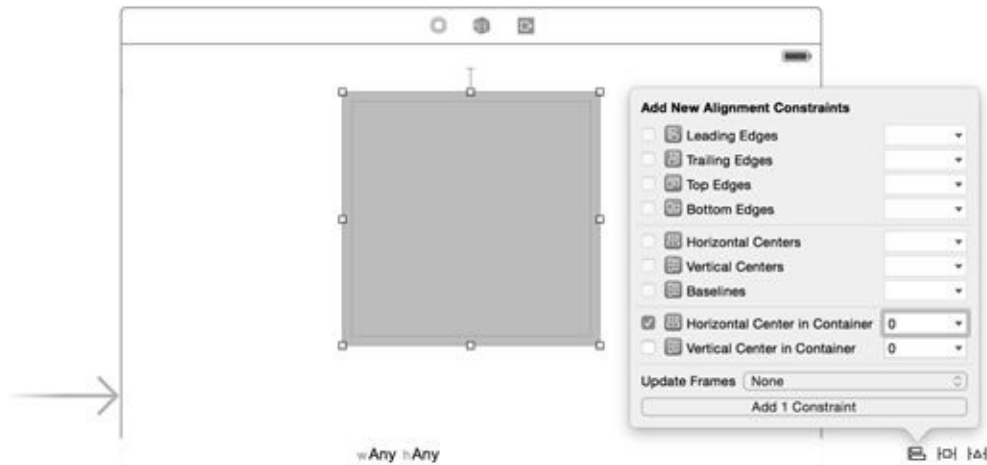


Figure 6-8: Adding an alignment constraint

These types of constraints are added from the Align menu, as illustrated in Figure 6-8. The constraints available for single elements are the Horizontal Center in Container (to center the element horizontally) and the Vertical Center in Container (to center the element vertically). The rest of the options are only available for multiple elements. All the options on this menu also have a value that determines the offset of the constraint. By default this value is 0, but we can change it to move the view to the left (positive values) or the right (negative values). In this example, we have selected the Horizontal Center in Container constraint with an offset value of 0 to center the view horizontally and right in the middle of the container.

Now the view is centered on the screen, but the constraints are still orange. The two constraints we have added so far (the Top Space constraint and the Horizontal Center in Container constraint) tell the system that the top of the view has to be 20 points from the top of the container and aligned with its center. This is enough to establish the position, but we still have to provide more information for the size. The box could stretch or shrink to always be at a certain distance from the sides of the container, as we did with the button in Figure 6-2, or it could have a specific size. For this example we decided to go with the last option and give our view a specific size.

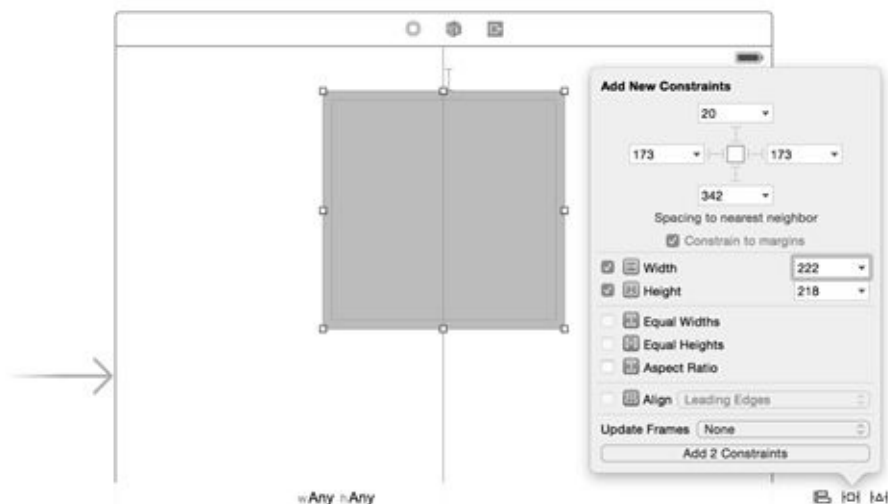


Figure 6-9: *Adding the Width and Height constraints*

The Width and Height constraints tell the system that the view always has to have a specific width and height. The size is determined by the values declared for each constraint in the Pin menu, as shown in Figure 6-9. By default, the menu shows the current values defined in the Storyboard, but we can change them to declare the size we want. For our example, we assigned the Width and Height constraints with the values by default. Figure 6-10 illustrates our view with all of its constraints.

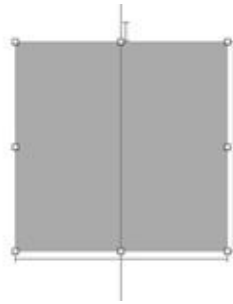


Figure 6-10: *View with all the necessary constraints*

All the constraints are now blue because the system has enough information to draw the view. The Top Space and Horizontal Center in Container constraints establish the position while the Width and Height constraints determine the size. No matter what the orientation or the size of the screen are, our view will be drawn at 20 points from the top margin of the main view, in the center of the screen, and with the same size established in the Storyboard, as illustrated by Figure 6-11.

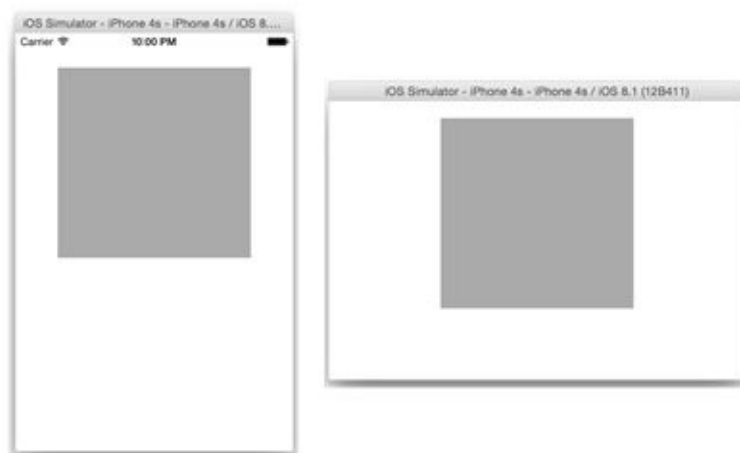


Figure 6-11: *Interface in portrait and landscape*

Do It Yourself: Add to the view in your Storyboard the Horizontal Center in Container constraint as shown in Figure 6-8 and the Width and Height constraints as shown in Figure 6-9. Run the application in the simulator. Rotate the window to see

how the view is repositioned on the screen.

The Basics: The simulator can simulate different devices and can also be rotated to show the application in landscape mode. The options to select the device and rotate the window are available in the Hardware menu at the top of the screen. You can also press the Command key along with the left or right arrows on your keyboard to perform the rotation.

Unless we have a reason to set a specific size for the element, we should always let the size of the elements be determined by the relation between the element and its siblings or the container. For instance, if we want the box of our example to always be in the center of the screen, it is better to set Leading and Trailing constraints to define an absolute space on the sides and let the system determine its size, as illustrated in Figure 6-12.

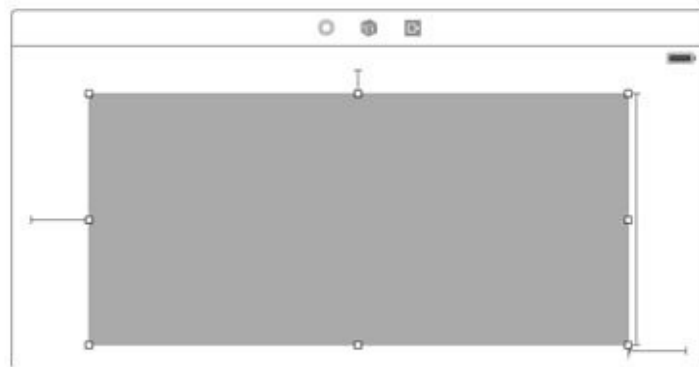


Figure 6-12: *Leading and Trailing constraints to define the width of the element*

In this example, the position and size of the view are determined by the Top Space, Height, Leading Space, and Trailing Space constraints. The Leading Space and Trailing Space constraints tell the system that the view has to be at a certain distance of the edge of the container and it should be shrunk or expanded to always be at that distance.

The view of Figure 6-12 has an absolute height determined by the Height constraint, but we could also change that assigning a Bottom space constraint or an Aspect Ratio constraint.

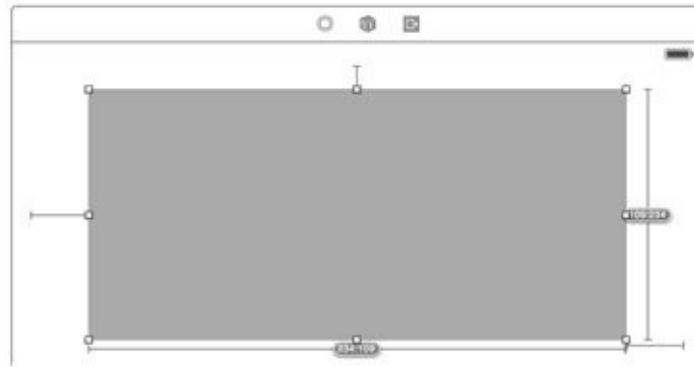


Figure 6-13: *The Aspect Ratio constraint maintains the proportions of the view*

The example of Figure 6-13 uses an Aspect Ratio constraint to determine the height of the view. The width is determined by the Leading Space and Trailing Space constraints, and the height is defined to be proportional to the current width by the Aspect Ratio constraint. When the view gets wider because of the Space constraints on the sides, it also gets proportionally taller. Because of these constraints, the view always has the same rectangular shape.

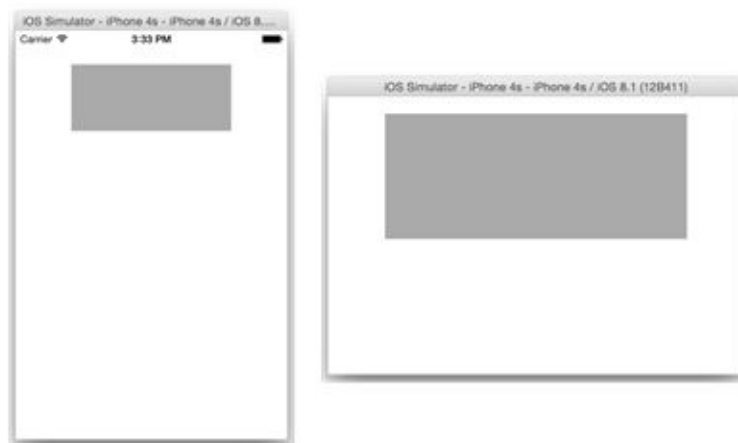


Figure 6-14: *The view's aspect ratio is the same in portrait and landscape*

Do It Yourself: Select and erase the view on your Storyboard to start from scratch. Add a rectangular view in the center of the main view, similar to the one shown in Figure 6-13. Assign a background color to be able to see it. Open the Pin menu and add a Top constraint of 20 points, a Leading constraint of 50 points and a Trailing constraint of 50 points (remember to click on the red lines to activate the constraints, Figure 6-6). Click on the Aspect Ratio constraint on this same menu to also add it to the view. Run the application, you should see something similar to Figure 6-14.

The Basics: The aspect ratio is a value that determines how many points on one side correspond to how many points on the other side, keeping the same proportions for the shape when it is shrunk or expanded. For example, a ratio of 1:1 determines that

for each point on one side, the other side should have one point. The Aspect Ratio constraint sets this value according to the current ratio of the view. Next, we will see how to modify this value and specify our own ratio.

Editing Constraints

Every constraint has associated values to determine its properties, such as the distance, the width of the element, etc. Assigning new values to the constraints changes how they affect the elements and how the interface is built. The Align and Pin menus only allow us to add new constraints, not to edit the ones already created. Xcode offers different alternatives to access and edit the constraints. The most simple is to select the element and open the Size Inspector panel in the Utilities Area (Figure 5-19). All the constraints associated with the element are listed at the bottom, as shown in Figure 6-15.

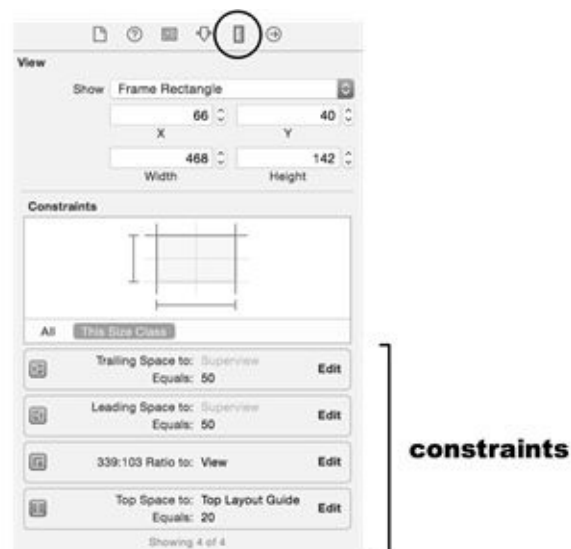


Figure 6-15: The list of constraints on the Size Inspector panel

The constraints are listed like buttons at the bottom of the panel. Figure 6-15 shows the four constraints added to the view of our last example (Trailing Space, Leading Space, Aspect Ratio, and Top Space, in this order). Each constraint in the list presents an Edit button to edit its values. Figure 6-16 shows the small window that opens when we click on this button. Changing the values in this window automatically modifies the constraint.

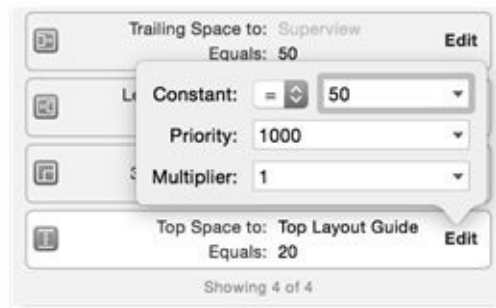


Figure 6-16: Popup window to edit the values of a constraint

Do It Yourself: Select the view in the Storyboard. Click on the button to show the Size Inspector panel (circled in Figure 6-15). Click on the Edit button of any of the constraints to change its value. Try, for example, to assign a value of 50 to the Top Space constraint, as shown in Figure 6-16. The changes will be immediately reflected in the Storyboard when possible.

IMPORTANT: There are different options available to configure the constraint. The option that determines the value of the constraint, such as the distance or the width, is called *Constant*. We will study how to configure the rest of the options later in this chapter.

Constraints have their own panel with more configuration options. To open this panel we can double click the constraint on the Size Inspector panel (Figure 6-15), select the constraint from the Storyboard, or click on the item that represents the constraint inside the Document Outline panel. When a constraint is selected, its values are shown in the Utilities Area. Figure 6-17 is a capture of this panel after the Top Space constraint of the view from our example was selected. From this panel we can change all the constraint's values and also some of the aspects of the elements it is connected to, as we will see later.

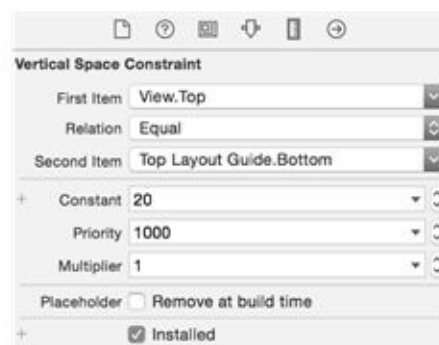


Figure 6-17: Constraint editor in the Utilities Area

Standard Values

The system defines standard values for the distance between elements and between the elements and their container to help us organize the interface. We don't have to use them, but their use is recommended because they adapt automatically to the device according to styles defined by Apple.

There are different ways to work with these values. For example, the standard space between a view and its container may be defined on the Pin menu. When a space constraint is activated, we can set the constraint to be between the element and the virtual margins defined for the device by selecting the option *Constrain to margins* (Figure 6-6).

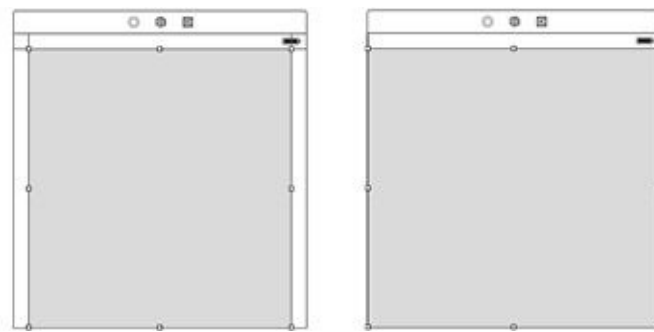


Figure 6-18: *Constrain to margins option activated and deactivated*

The view on the left side of Figure 6-18 was positioned by Space constraints with a value of 0 and the Constrain to margins option activated. This positions the view a few points away from the edge of the main view. On the other hand, the view on the right was affected by the same constraints but now with the Constrain to margins option deactivated. Because of this, the sides of the views reach the edge of the container (no margin is created).

When we use the Constrain to margins option, the constant value of the constraint determines the distance from the margin, not from the edge of the container. For example, if we declare a value of 20 for the constant and the margin is set as 10 points by the system, the distance between the elements will be 30 points. If we want to change this and modify the constraint to consider only the constant value, we have to remove the value of the margin from the constraint's panel in the Utilities Area deactivating the option *Relative to margin* (Figure 6-19, number 2).

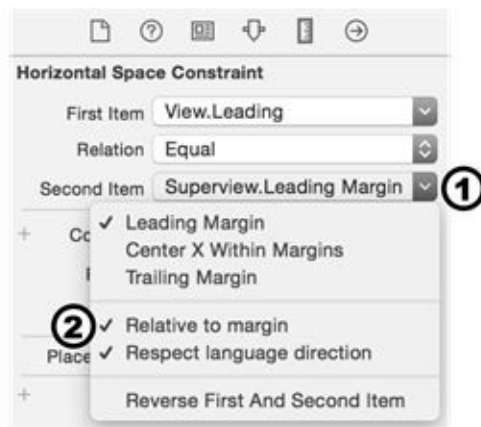


Figure 6-19: Constraint's configuration

The panel shows the two elements associated with the constraint. The element on the side of the constraint that considers the margin includes the word Margin in the description (Figure 6-19, number 1). If we click the arrow on the right, Xcode opens a menu with an option called Relative to margin. After we deactivate this option, the constraint won't consider the margin anymore. Figure 6-20 shows what happens with the interface of our previous example after the Leading constraint is modified.

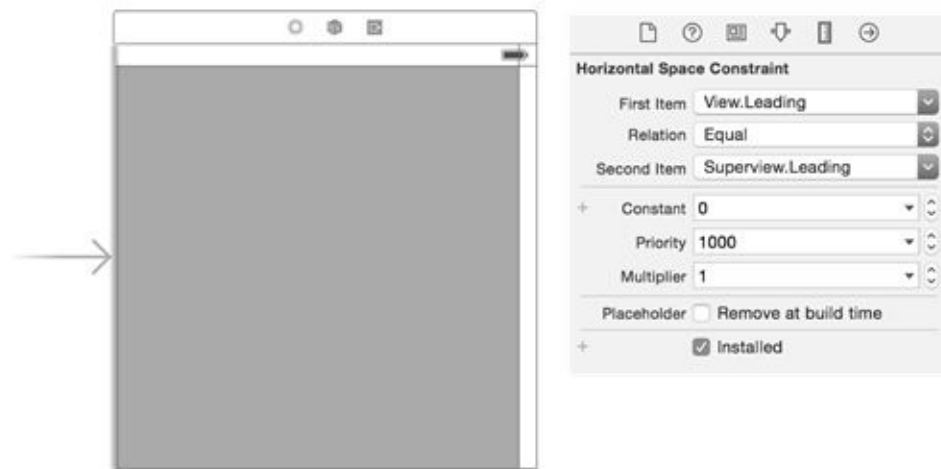


Figure 6-20: Leading constraint without considering the margin

These examples take advantage of the Constrain to margin option included in the Pin menu, but we can also set standard values for the constraints from the popup menu displayed when we click on the arrow at the right side of the constraint's value, as shown in Figure 6-21.

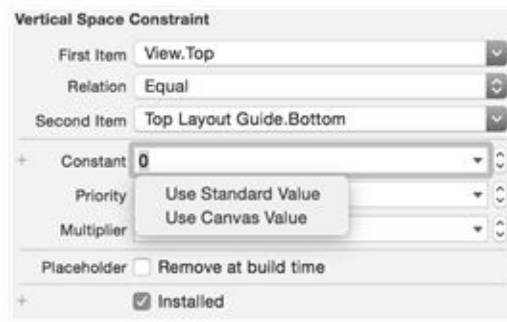


Figure 6-21: Use Standard Value option

The option we have to select to set a standard value for the constraint is called *Use Standard Value*. Every time we want our interface to follow Apple’s guidelines, instead of declaring an absolute value for the constraint we have to select this option. It is available everywhere we are able to modify the constraint’s value. If this value is declared, the system assigns automatically a value corresponding to the device where the app is running.

The Basics: The use of standard values is recommended because it ensures that our app is going to look good in every device available. It is considered best practice and you should use them when possible, especially when defining the distance between two elements, as we will see later.

Resolving Auto Layout Issues

As we mentioned before, Xcode shows the constraints in different colors depending on the information they provide: orange when they don’t provide enough information and red when the information is ambiguous. But there is another situation when although the constraints are sufficient Xcode can’t determine the new place and size of the views in the Storyboard. There is a mismatch between what we see in the Storyboard and what we will see after the constraints are applied. This situation may occur because Xcode decides that is better to show the view with the current position and size to make easy for us to edit the interface, or it simply can’t determine the new position and size under the current conditions. To help us identify the problem, Xcode shows dashed lines demarking the area where the view will be after the constraints are applied.

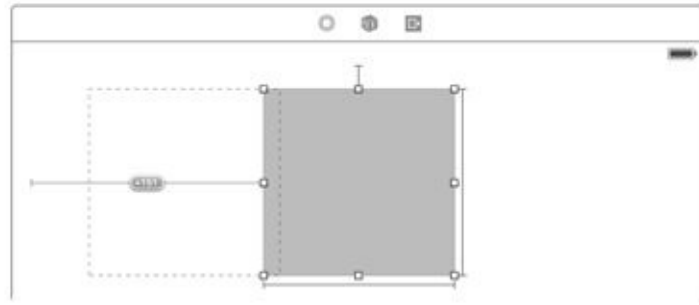


Figure 6-22: *Element in the wrong position*

The example of Figure 6-22 presents a simple view with a Top Space constraint and Width and Height constraints to determine its size. The horizontal position was specified by a Leading Space constraint added at the end. This last constraint determines that the view should be 50 points from the left edge of the container, but Xcode decided that it was better to preserve the view in its original place and show dashed lines demarking the new position instead. This is just an indication and we could keep adding or modifying the constraints until the design is finished, but if we need the Storyboard to represent the final interface, we can update the views from the Resolve Auto Layout Issues menu (Figure 6-3, button number 5).

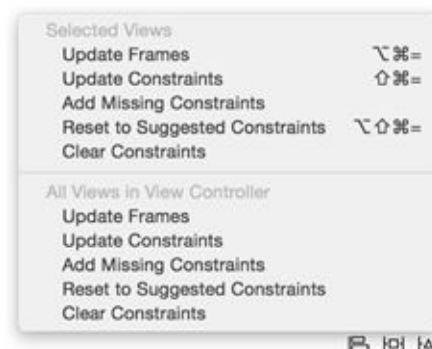


Figure 6-23: *Resolve Auto Layout Issues*

The Resolve Auto Layout Issues menu not only allows us to solve this situation but it can also help us determine what the right constraints are for our interface. The following are the options available.

Update Frames—This option updates the values of the view’s frame (position and size) to match the ones established by the constraints.

Update Constraints—This option updates the values of the constraints to match the current view’s position and size.

Add Missing Constraints—This option adds the constraints for us. Xcode guesses how the constraints should be according to the current position and size of the view on

the Storyboard and automatically adds those constraints. It is useful as a starting point.

Reset to Suggested Constraints—This option modifies the constraints to take them back to the state established by Xcode when the suggested constraints were generated.

Clear Constraints—This option erases the constraints. We can select this option when the constraints are not working and we decide to start from scratch, but it is particularly useful when working with Size Classes, as we will see later.

The options are available twice, for the selected views and also for all the views inside the current main view. Figure 6-24 shows the view of our example after its frame is updated.

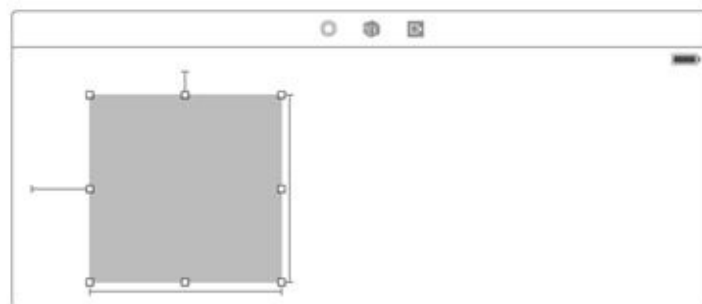


Figure 6-24: Frame updated

Do It Yourself: Erase all the views in your main view. Add a new view similar to Figure 6-22 at the center of the main view. Assign a Top Space constraint of 20 points and also the Width and Height constraints to set its size. Finally, add a Leading Space constraint of 50 points. Xcode will show the dashed lines to indicate the final position and a value on the line representing the difference between the current distance and the real one. Select the option Update Frame or Update Constraints to update the interface. As with any other software, you can always press the Command-Z keys to undo and try again.

Xcode offers tools with more specific information to help us identify and solve problems with constraints. One easy way to get this information is from the Error button at the top of the Editor Area (Figure 6-25, number 1).



Figure 6-25: Errors and Warnings

This button displays a list of the errors and warnings that are currently affecting our application. Figure 6-26 shows the warning displayed when a view is not in the position determined by the constraints.

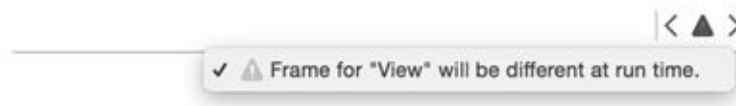


Figure 6-26: Current issues with our constraints

Another way to get a detailed explanation of the issues regarding our constraints is from the Document Outline panel (Figure 6-3, number 1). This panel displays a list of the elements and constraints in the interface and also a small button at the top right corner whenever an issue occurs with the constraints (Figure 6-27, number 1).



Figure 6-27: Get the list of issues from the Document Outline panel

When we click this button, the list of views and constraints is replaced by the list of issues our interface currently has.



Figure 6-28: List of issues in the Document Outline panel

The issues are listed one over another with detailed information, such as the current and expected values, the constraints needed, etc. Every item of the list also includes

another button (Figure 6-28, number 1) to open a popup menu with suggestions and solutions.

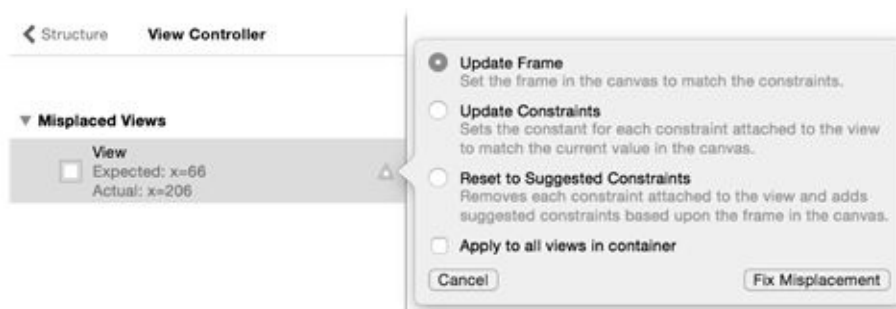


Figure 6-29: Automatic solutions for constraint issues

Some of the options in this menu are available in the Resolve Auto Layout Issues menu, such as Update Frames or Update Constraints, but they adapt to the issue we are dealing with.

Intrinsic Content Size

Some elements like labels and buttons generate their own content. This content defines a minimum size that the element adopts when no other size is determined by the constraints. For example, the intrinsic content size of a label is determined by its text. Because of this property, sometimes all we need for an element is to declare enough constraints to determine its position.



Figure 6-30: Label defined with intrinsic content size

The label of figure 6-30 has only two constraints, Top Space of 20 points and Leading Space of 50 points. There are no requirements for its size. If no constraints are defined, the label automatically adopts the size of its text. This is useful when working with localization (text that adapts to the human language set on the device), as we will see in Chapter 31.

The Basics: Elements with intrinsic content size always adopt the size of their

content when they are created. If for some reason we modify their content or their frame, we can go to the Editor menu at the top of the screen and select the option *Size to Fit Content* to force the view to go back to its intrinsic content size.

If we need to know what the current intrinsic content size of a view is, the `UIView` class includes the following method to return this value.

`intrinsicContentSize()`—This method returns a `CGSize` value with the intrinsic content size.

The following example connects a label with an Outlet called `mytitle`, assigns a new text, and prints the new values of its intrinsic content size on the console.

Listing 6-1: Reading the intrinsic content size

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var mytitle: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        mytitle.text = "Description: My New Label"

        let labelSize = mytitle.intrinsicContentSize()
        print("New Width: \(labelSize.width)")
        print("New Height: \(labelSize.height)")
    }
}
```

Do It Yourself: Erase the views in the main view and any previous code in the view controller. Add a new label to the main view with a Top Space and a Leading Space constraint to define its position. Create an Outlet for the label called `mytitle`. Copy and paste the code from Listing 6-1 into your view controller. Run the app. The text for the label is modified as soon as the app is executed and the new values for its intrinsic content size are printed in Xcode’s console.

Multiple Views Constraints

The Pin and Align menu add the constraints to every selected element. For one element the addition is simple, but when two or more elements are selected the constraints added are the same for every one of them. Xcode provides an additional way to add constraints that makes it easy to connect elements with each other and define individual constraints. This is the same process we use to create Outlets and Actions, but instead of pressing the Control key and dragging the line to the view controller we do it to a view, as shown in Figure 6-31.

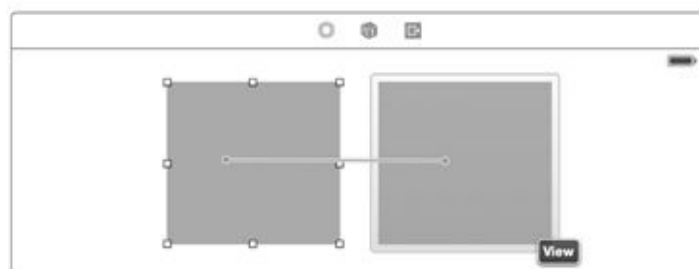


Figure 6-31: Control-drag to add a constraint

For this example we have created two square views in the center of the main view. What we want to do is to add a constraint to determine the space between the views. The process is as simple as control-dragging a line from one view to another. The target view is highlighted in blue to indicate which element is going to be associated with the constraint. When the mouse button is released, Xcode shows a popup menu over this target to select the constraint we want to add.



Figure 6-32: Quick menu to add constraints

We can use this system to add all the same constraints applied before. For example, control-dragging from the view on the left to the main view we can add a Top Space constraint for each view.

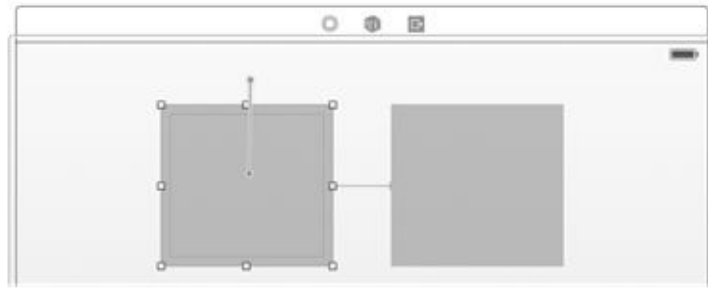


Figure 6-33: *Adding a Top Space constraint*

The Basics: If you hold down the Command key or the Shift key, you can select multiple constraints from the same menu. After all the constraints are selected, press the Enter key or click outside the menu to add them to the views.

When we drag a line inside the same view, the menu displayed by Xcode includes the Width or Height constraint depending on the direction of the line. Figure 6-34 shows the options available after the horizontal line is dragged from and into the left view.

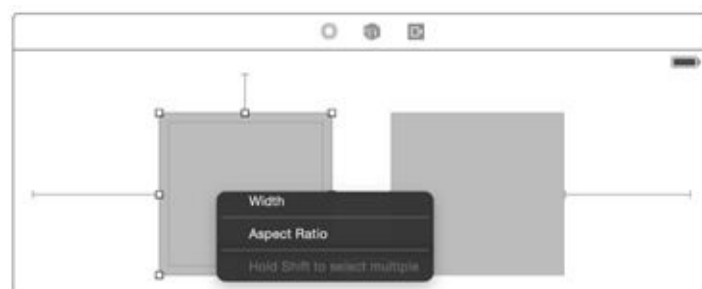


Figure 6-34: *Popup menu with the Width constraint*

When working with two views connected with each other, we have to make sure that the constraints provide enough information for the system to position and size the views. For example, if we only add Space constraints on the sides and between the views, as we did in the example of Figure 6-34, the system won't be able to determine the width of the views, because it won't know which one has to be expanded or shrink to occupy the remaining space. There are different options available, depending on what we want to achieve with our interface. An alternative is to define the width for only one view and let the system determine the width of the other. In the example of Figure 6-35, next, we pinned the views to the sides and between each other, but only added a Width constraint for the view on the left.

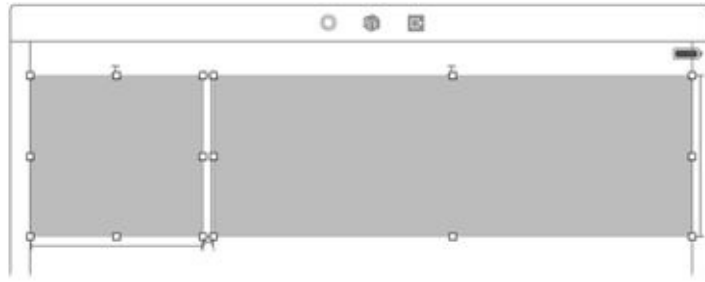


Figure 6-35: *Standard values for Space constraints*

Because we defined a fixed width for one view only, the view on the left is always the same size while the view on the right expands or shrinks to fill the space available. If we had defined both views with a fixed size, the system wouldn't have been able to adapt the interface to the size of the screen and we would have received an error message.

Do It Yourself: Delete the views in your Storyboard. Add two square views as in Figure 6-31. Control-drag from the left view to the right view to add a Space constraint (Figures 6-31 and 6-32). Control-drag from the left view to the top of the main view to add a Top Space constraint. Repeat the process for the right view. Control-drag from the left view to the left side of the main view to create a Leading Space constraint. Repeat the process with the view on the right to create a Trailing Space constraint (Figure 6-34). Control-drag a horizontal line inside the view on the left to create a Width constraint (Figures 6-34). Control-drag a vertical line inside each view to create Height constraints. Select the views and modify the values of their Space constraints to the standard value. Update the frames and run the application. Rotate the simulator to see how the views adapt to the space available.

Besides the Width and Height constraints there are other constraints we can use to set the size and achieve a different purpose. For example, applying an Aspect Ratio constraint to one view and the Equal Width and Equal Height constraints to both of them we get two views of equal size that shrink and expand according to the space available. The Aspect Ratio constraint may be added from the Pin menu or control-dragging inside the view. The Equal Width and Equal Height constraints may be added from the Pin menu or control-dragging from one view to another.



Figure 6-36: Menu to add Equal Width and Equal Height constraints

In the example of Figure 6-36, the Width and Height constraints added before were erased and an Aspect Ratio constraint was added to the view on the left. The figure shows the popup menu we get after we control-drag a line from one view to another to add the Equal Width and Equal Height constraints.

The final design includes a total of 8 constraints. We have two Top Space constraints to pin the views to the top of the container, a Leading Space constraint to pin the view on the left to the left side of the container, a Trailing Space constraint to pin the view on the right to the right side of the container, another Space constraint added to both views defining the space between them, an Aspect Ratio constraint to define the size of the view on the left and the Equal Width and Equal Height constraints to determine that both views should always be of equal sizes. The result is an interface with two views of the same size that shrink or expand in equal proportions to meet the requirements.



Figure 6-37: Equal Width and Equal Height constraints on the simulator

Do It Yourself: Delete the Width and Height constraints in both views (select them from the Storyboard or the Document Outline panel and press the Delete key on your keyboard). Add an Aspect Ratio constraint to the view on the left and control-drag from one view to another to add the Equal Width and Equal Height constraints. Update the frames if necessary and run the application. The simulator should show something similar to Figure 6-37.

Constraints were developed to affect single properties, such as a specific dimension of an element or the distance from one element to another. Because of this, designing an interface, even a simple one, requires an intricate network of constraints interconnecting the elements together. Fortunately, this elaborated system can be simplified with the application of some basic tricks. One alternative is to add our views and controllers to other views that act as containers, which distributes the constraints between different

levels of the hierarchy.

In the following example, we have positioned a square view in the center of our main view and added a label inside. The constraints for the view are associated with the main view, but the constraints for the label are associated with its own container (the square view).



Figure 6-38: Views inside views

Figure 6-38 shows the constraints for the view and the label. The view has a Width and a Height constraint to define its size and a Top and a Leading Space constraint to determine its position. The label, on the other hand, has Horizontal and Vertical alignment constraints to determine its position inside the view. Because of this distribution, the view adapts to the space available on the screen, but the label is attached to its container and adapts only to that space (the label is always at the center of the container, no matter what the container's position is).

Do It Yourself: Delete the previous views on the Storyboard. Add a square view at the center of the main view with a Top Space constraint, a Leading Space constraint and the Width and Height constraints shown in Figure 6-38. Add a label inside with a Horizontal and Vertical Center in Container constraints (called *Center X* and *Center Y* in the constraints quick menu). Run the application and rotate the screen to see how the constraints keep the label at the center.

Another possible trick involves the use of invisible views to help us organize the rest of the views. The following example centers two labels in the main view, using a third view as a spacer.

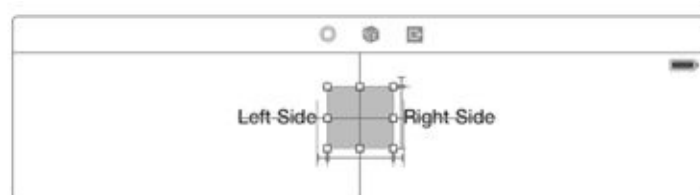


Figure 6-39: Labels centered with a view

The view is centered in the main view and it has a fixed width and height, and the labels are pinned to the sides of the view. The constraints necessary to create this design are a Top Space and a Horizontal Center in Container constraint to position the view at the center of the main view, and Width and Height constraints to determine its size. The position of the labels is determined by a Vertical Center constraint and Leading and Trailing constraints connected to the view. These constraints determine that the labels are going to be positioned on the sides of the view and share the same vertical center.

The spacer is sometimes styled with a background color during development and then turned transparent when the app is ready. Figure 6-40 illustrates what the final interface looks like. The labels are centered and always at the same distance from each other.

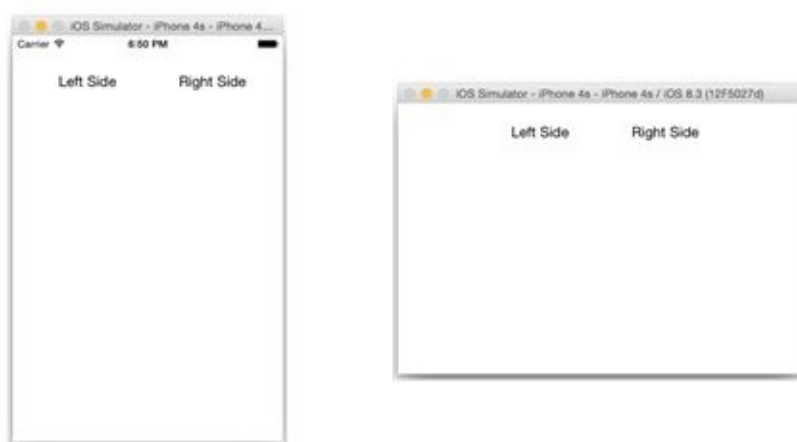


Figure 6-40: *Spacer between elements*

Do It Yourself: Delete the previous views in the Storyboard. Add a square view at the center of the main view with a Top Space constraint, a Horizontal Center in Container constraint and the Width and Height constraints shown in Figure 6-39. Add two labels on each side of the view. Add Trailing and Leading constraints between the labels and the view. Add a Vertical Center constraint between each label and the view. Run the application in the simulator. You should see something similar to Figure 6-40.

Beside the Vertical Center constraint applied in the last example to center the labels vertically with the view, there are more alignment constraints available to align two or more elements together. These types of constraints are added from the Align menu and their application is straightforward, as shown in Figure 6-41.

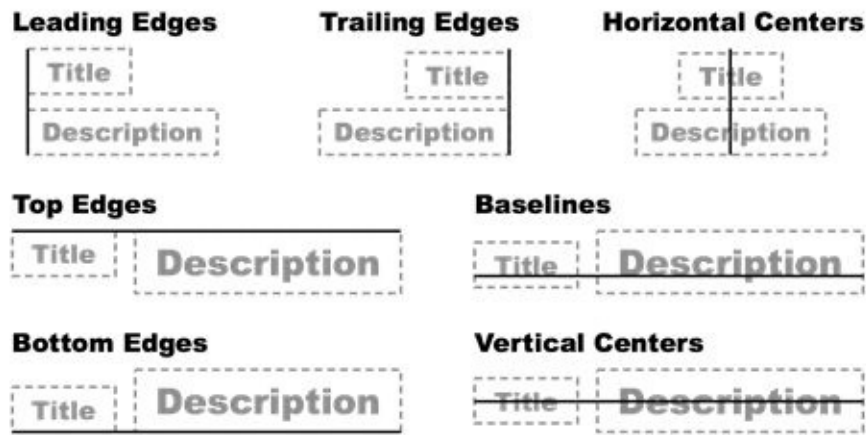


Figure 6-41: Alignment constraints for multiple views

These types of constraints provide the alternative to build more complex layouts. For example, they can help us determine the size of an element according to the space occupied by two or more other elements. Figure 6-42 illustrates this situation.



Figure 6-42: Width determined by a Trailing Edges constraint

This example includes two labels and a button. All the elements have Space constraints to the margins of the container and between them to establish their positions. Their sizes are determined by their intrinsic content size, except for the label at the top. The width of this label is determined by a Trailing Edge constraint connected to the button (called *Right* on the quick menu). Because of this, the label at the top always extends to reach the right side of the button, independently of the size of the rest of the elements.

Do It Yourself: Delete the previous views in the Storyboard. Add two labels and a button. Assign a background color to the label at the top to be able to see the size of its view. Add Space constraints to pin the labels to their container and with each other. The label at the top needs a Top Space and a Leading Space constraint to the container. The label at the bottom needs a Top Space constraint to the other label and a Leading Space constraint to its container. The button needs a Leading Space constraint and also a Vertical Center constraint to the label (Horizontal Spacing and Center Y on the quick menu). Finally, add a Trailing Edges constraint between the top label and the button (called *Right* on the quick menu). Update the frames if necessary. You should see something similar to Figure 6-42.

Relations and Priorities

The constraints we have defined so far are constant. This means that their values don't change and the system has to satisfy every one of them or return an error in case of failure. But some situations require more flexibility. Sometimes constraints need only an approximate value or they are not required all the time. For these situations, constraints include two more configuration values called *Relation* and *Priority*. The Relation determines the relationship between the constraint and its constant value, while the priority determines which constraint is more important. Important constraints are satisfied first and then the system tries to satisfy the rest as close as possible. The possible relations are the following:

- **Equal** determines that the constraint's value must be equal to the constant value. This is the relation by default.
- **Less Than or Equal** determines that the constraint's value can be equal or less than the constant value.
- **Greater Than or Equal** determines that the constraint's value can be equal or greater than the constant value.

On the other hand, the priorities are defined by an integer value between 1 and 1000. For common situations, Xcode declares three predefined states:

- **Low (250)** determines a low priority.
- **High (750)** determines a high priority.
- **Required (1000)** determines that the constraint is required.

In complex interfaces, these properties usually work together, but they are also useful in simple situations. For example, the interface of Figure 6-43 contains a label that is always at the same distance from the view on the right. The view shrinks or expands, but the label remains at the same position and with the same size.

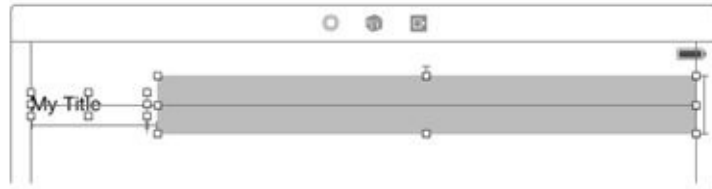


Figure 6-43: *Label with a fixed size*

In this particular example, the label was defined with a Width constraint of 100, a Leading Space constraint that pins it to the left side of the main view and a Vertical Center constraint that keeps it vertically centered with the gray view. The view has a Height constraint to define its height and Space constraints with standard values to define its position and the distance between its left side and the label, but its width is determined by the space available. The result is that the view expands or shrinks but the label always occupies 100 points.



Figure 6-44: *The view expands on landscape*

The problem arises when a text longer than 100 points is assigned to the label. Because the constraint determines a fixed size of 100 points for the label, the new text is clipped.

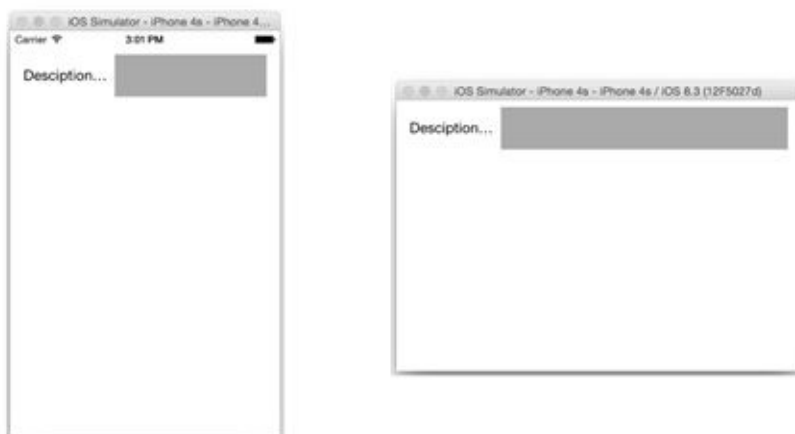


Figure 6-45: Longer text gets clipped

What we want is for the label to stay at a size of 100 points but to grow when the space is not enough. We need the value of the Width constraint to be Greater Than or Equal to 100 points. To change the relation we have to select the constraint, click the button on the right side of the Relation value (Figure 6-46, number 1), and select the option from the quick menu (Figure 6-46, number 2).

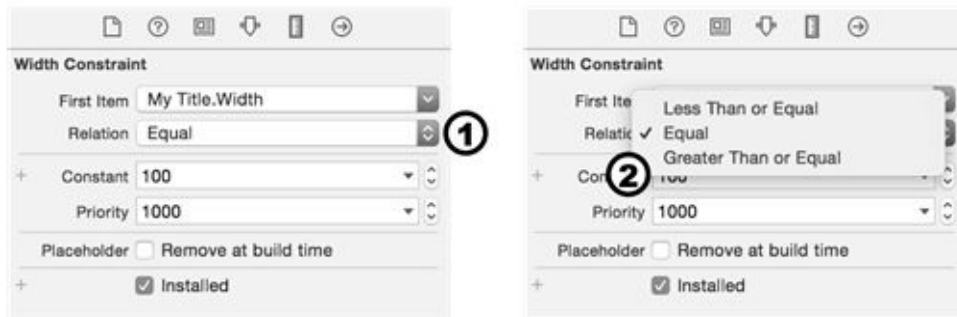


Figure 6-46: New Relation for the constraint

After the Greater Than or Equal relation is selected the system tries to keep its width at 100 points whenever possible, but it lets the label expand if necessary. Now the text is not clipped and the view shrinks according to the size of the label.

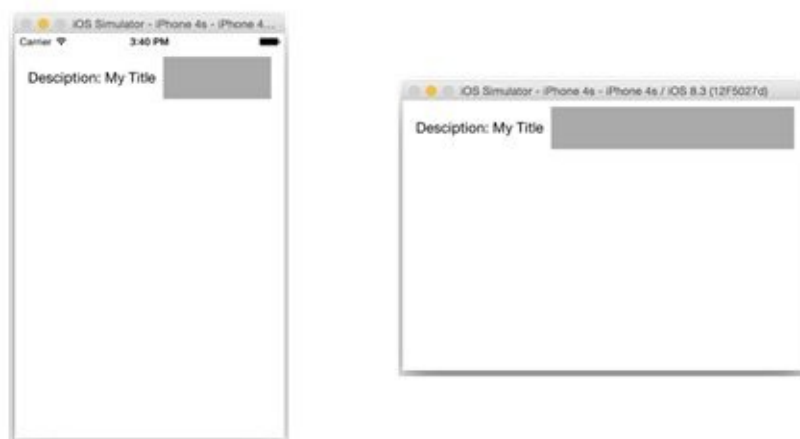


Figure 6-47: The label is expanded

Do It Yourself: Delete the previous views in the Storyboard. Add a label on the left side of the main view with the text “My Title” and a long view on the right with a gray background, as shown in Figure 6-43. Add a Leading Space constraint and a Trailing Space constraint to the label with standard values. Add a Width constraint to the label, select it and change its constant value to 100 and the relation to Greater Than or Equal. Add a Top Space constraint and a Trailing Space constraint to the

view with standard values. Add a Height constraint to the view to define its height. And lastly, add a Vertical Center constraint between the view and the label. Run the application on the iPhone 4 simulator. You should see something similar to Figure 6-44. Create an Outlet for the label in the view controller and change the label's text to "Description: My Title", as we did in Listing 6-1. Run the application again. You should see something similar to Figure 6-47.

There are times when two or more constraints have to be constant but they can't be completely satisfied. In cases like this, the system needs to know which one is more important. In the interface of Figure 6-48, two views are on the sides of a label. The views expand to occupy the space around the label, but they can't reduce their sizes further than 50 points (the reason could be that they are going to contain an image that is 50 points wide, so their sizes can't be less than the size of the image).



Figure 6-48: *Conflicting constraints*

In our example, the label was defined with a Width constraint Equal to 208 points. The views are pinned to the main view and to the label by Space constraints and they also have a Width constraint but this time Greater Than or Equal to 50 (the views are connected to each other by an Equal Width constraint to make their widths equal all the time). This means that the label will always be of the same size, but the views will expand or shrink to fill the space available. When there is plenty of space, the constraints work as expected. The views expand from the label to the margins of the main view to fill the space available. The problem emerges when there is not enough space to simultaneously satisfy all the Width constraints. The label has to always be 208 points wide and the views' width can't be less than 50 points. As we can see from Figure 6-49, below, the app doesn't crash and the system solves the problem, but it is not what we want (the views are now smaller than 50 points in portrait).



Figure 6-49: *Unable to satisfy constraints*

IMPORTANT: In these situations, the app won’t crash but the system will print an error on the console with the message “Unable to simultaneously satisfy constraints” followed by a list of all the constraints that couldn’t be satisfied. To find the solution you have to modify the constraints involved until no error is returned.

We have defined the Width constraints for the views to be Greater Than or Equal to 50 because it is important for us that they are never smaller than 50 points. But all the constraints were defined by default with a priority of 1000. For the system, the Width constraint of the label is equally important to the Width constraints of the views and it doesn’t know what to do when there is not enough space to satisfy all of them. The solution is simple. We have to communicate our priorities to the system. We have to tell the system that it is important for us that the width of the views is never less than 50 points. And this is achieved by reducing the priority for the label’s width.

The constraints with a higher-priority value are satisfied first, and then the system tries to satisfy the rest as close as possible. Figure 6-50 shows what we see in the simulator if we assign a priority of, for example, 750 to the label’s Width constraint.

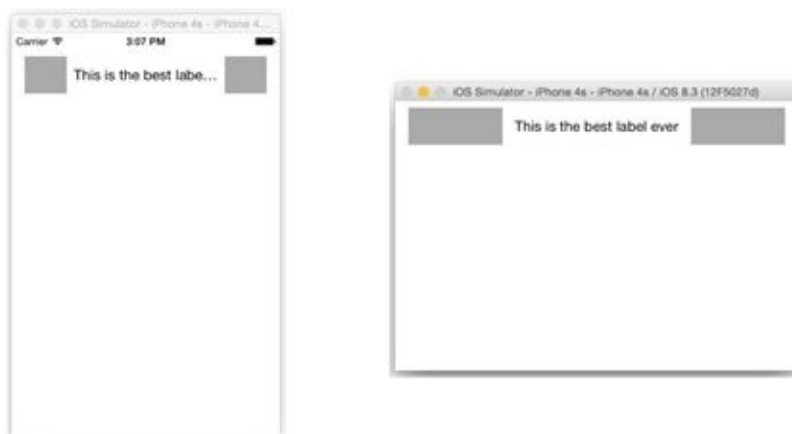


Figure 6-50: *Label’s Width constraint with a lower priority*

Do It Yourself: Delete the previous views in the Storyboard. Add a label at the center of the main view with the text “This is the best label ever”. Add two views on the sides as shown in Figure 6-48. The constraints for the view on the left are a Leading Space and a Top Space to the main view with a standard value, a Trailing Space to the label with standard value, a Height to define its height and a Width constraint Greater Than or Equal 50. The constraints for the view on the right are a Trailing Space and a Top Space to the main view with standard value, a Leading Space to the label with standard value, a Height to define its height, and an Equal Width constraint connected to the view on the left (they have to always be the same width). The constraints for the label are a Width of 208 and a Vertical Center constraint to center the label with the views. Run the application on the iPhone 4 simulator. You should see something similar to Figure 6-49. Change the priority of the label’s Width constraint to 750 (Figure 6-46). Run the application again. You should see something similar to Figure 6-50.

IMPORTANT: Besides Constant, Relation, and Priority, there is an additional value available to configure constraints called *Multiplier*. The multiplier is multiplied by the value of the constraint to get a proportional value. For example, when the Multiplier value of an Equal Width constraint is set to 2, the element will have a double width than the original. The system also uses the Multiplier to set the value for the Aspect Ratio constraint.

Elements with intrinsic content size, such as labels and buttons, present two more types of priorities called *Content Hugging* priority and *Content Compression Resistance* priority. These are special priorities that show up on the Size Inspector panel as soon as a constraint is added to the element. Figure 6-51 shows the controls available on the Size Inspector to modify their values.

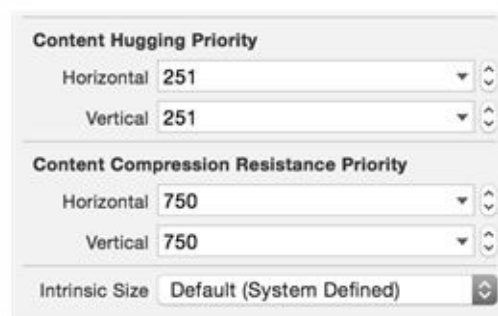


Figure 6-51: Controls to modify content priorities

The Content Hugging priority determines how much the element wants to stay of the size of its content and avoid expanding (it hugs to its content). The Content Compression

Resistance priority, on the other hand, determines how much the element wants to stay of the size of its content and avoid shrinking (resist compression). Figure 6-52 presents an example of two labels with a visible background to demonstrate how this works.



Figure 6-52: *Labels with content priorities*

The labels are pinned to the main view and to each other with Space constraints and standard values. Their widths were not defined, so they will shrink or expand according to the space available. Which label expands and which one shrinks is decided by the system considering their priorities. For this example we have decided to assign to the label on the right higher values for its priorities. Its horizontal Content Hugging priority was declared as 750 and its Content Compression Resistance priority was declared as 1000. This won't let the label expand or shrink so it is always the label on the left that adapts to the space available.



Figure 6-53: *Labels with different content priorities*

If, for example, we want the label on the right to shrink when there is not enough space, we can change its Content Compression Resistance priority to a value smaller than the value of the label on the left. Figure 6-54 shows the effect of assigning a value of 250 to the Content Compression Resistance priority for the label on the right and a value of 750 for the label on the left. The label on the left keeps expanding when there is more space available, but now the label on the right shrinks to make some room when the space is not enough to fit both.

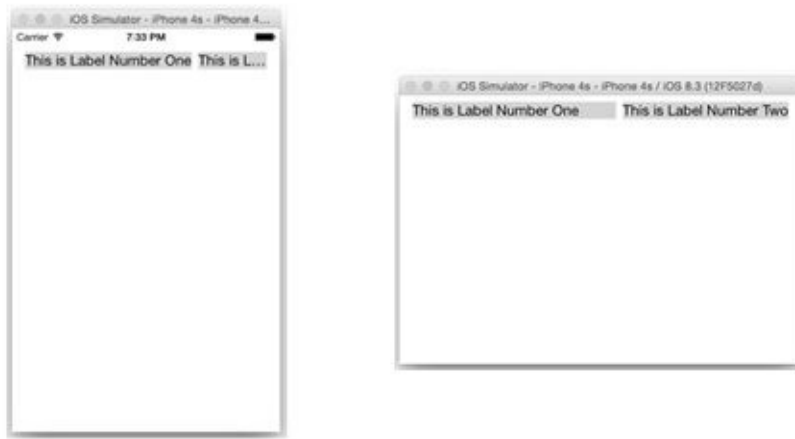


Figure 6-54: Labels with different compression values

The Basics: The priorities can be set for both the horizontal and vertical axis. The horizontal values usually apply to elements like labels and buttons, while the vertical values apply to Text Fields or images.

Constraints in the Document Outline Panel

The Document Outline panel (Figure 6-3, number 1) is a shortcut to the elements and constraints in the Storyboard. The panel allows us to select an element or a constraint to modify their values, connect them to Outlets and Actions in our code, change their names to make them easy to find, move them up or down in the hierarchy, delete them, and also create new constraints. Figure 6-55 illustrates how to create a new constraint between the gray view of our example and the main view.

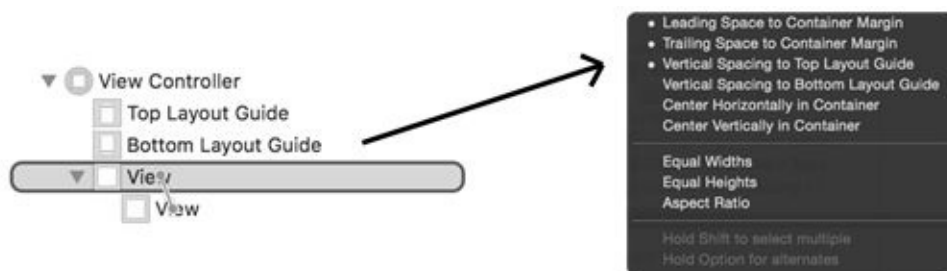


Figure 6-55: Adding constraints from the Document Outline panel

Instead of control-dragging a line from one element to another in the Storyboard, we have to drag the line from one item to another in the Document Outline panel and select the type of constraint we want to add from the popup menu (Figure 6-55). The same result is produced when we drag a line from an element in the Storyboard to an item in the panel. This technique is particularly useful when some of the elements are not visible. A very

common scenario is to have a view as the background of our interface that covers the main view's area but contain some elements that need to be pinned to the main view (e.g., an image used as a background). Figure 6-56 illustrates how to create constraints between a label inside a view and the main view.

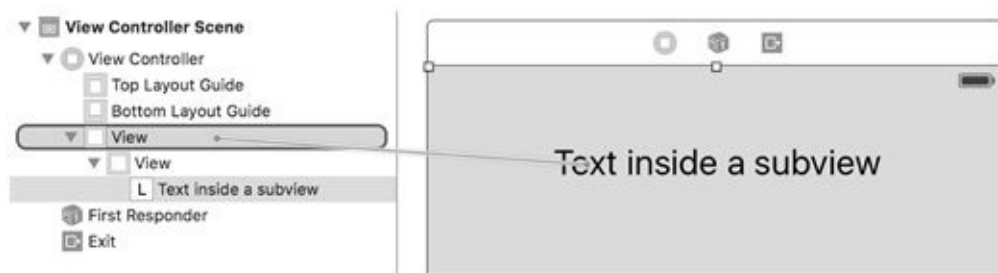


Figure 6-56: Constraints between elements and items in the Document Outline panel

The Basics: Connecting views from different levels in the hierarchy, as we did in Figure 6-56, is required when working with more complex interfaces that combine multiple elements or include images or views based on the `UIScrollView` class. We will study how to work with images in Chapter 8 and how to create Scroll Views in Chapter 11.

The label in Figure 6-56 is a subview of the view inside the main view, but in our example it was connected with constraints to the main view. This means that its position is not determined by its container but by the main view. If the container changes position or size, the label will remain in the same place. This is useful in some circumstances but not always. If what we need is just to position the label over the element and not inside, we can change its place in the hierarchy by dragging the element in the Document Outline panel to the new place, as illustrated by Figure 6-57.

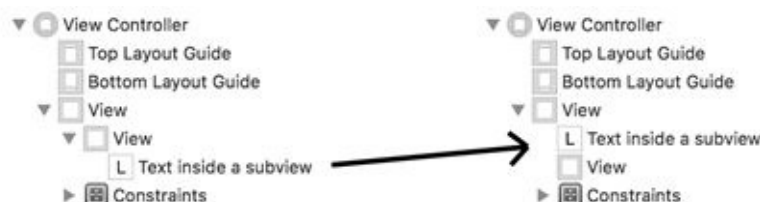


Figure 6-57: The views hierarchy was modified

Now the label is inside the main view instead of the subview. But there is still a problem. Views are drawn in the order they are presented in the Document Outline panel. In the example of Figure 6-57, the label will be drawn first and the subview next. This means that the subview will cover the label. For the user to be able to see the label, we have to move it down the list by dragging its item again. The result is shown in Figure 6-58.



Figure 6-58: *The view's place was modified*

Once the label is outside its container and in the right place, we can assign the constraints again, but this time we can use the Pin and Align menus because the label's container is now the main view.

When working with elements and constraints inside the Document Outline panel it is difficult to remember which item corresponds to an element in the Storyboard. To simplify our work, we can change the names of the items (elements or constraints) by selecting the item and pressing the Enter key. For instance, in our example, we added two constraints to the label, a Top Space and Horizontal in Container constraints, to position the label at the top and in the center of the main view. Xcode uses the text on the label to name these constraints. The names are so long that they don't even fit inside the panel. Figure 6-59 shows what they look like after we shorten their names.



Figure 6-59: *New names for the constraints*

NSLayoutConstraint

The information for the constraints is stored as XML language in the storyboard file along with the rest of the information for the interface. Once the application is executed, objects that represent the constraints are created. These objects are instances of the `NSLayoutConstraint` class. The objects defined in the Storyboard are automatically created from this class, but we can also use it to create our own constraints from code. The class includes the following initializer and a type method.

`NSLayoutConstraint(item: view1, attribute: attribute1, relatedBy: relation, toItem: view2, attribute: attribute2, multiplier: multiplier, constant: constant)`
 —This initializer creates an `NSLayoutConstraint` object from the information provided by

the attributes. The **view1** and **view2** attributes are the views affected by the constraint. The **attribute1** and **attribute2** attributes are `NSLayoutAttribute` values to specify the type of constraint. And the **relation**, **multiplier**, and **constant** attributes provide the respective values for the constraint.

constraintsWithVisualFormat(string, options: options, metrics: metrics, views: views)—This type method returns an array with all the constraints defined by the **string** attribute. The constraints are declared in a visual format language. The **options** attribute is an `NSLayoutFormatOptions` value to specify the alignment of multiple views. The **metrics** attribute is a dictionary used to assign names to the values in **string**. And the **views** attribute is another dictionary used to assign names to the views involved.

The `constraintsWithVisualFormat()` method is probably the most practical way to create `NSLayoutConstraint` objects. The method is relatively simple and it can create several constraints at once. The constraints are defined inside a string of characters written with a specific formatting language. The following are the characters used by the language to represent views, constraints, and their values.

- **H:** declares all the constraints defined in the string to be horizontal (from left to right). This is the orientation by default.
- **V:** declares all the constraints defined in the string to be vertical (from top to bottom).
- **[name]** it represents a view. The **name** attribute is the name that represents the view.
- **|** it represents the container.
- **-** it represents a Space constraint. A single hyphen represents a constraint with the standard value. To declare other values we have to write the number between hyphens (e.g., -25-).
- **(value@priority)** it declares the value and the priority of the constraint. It could be used after the name of the view to declare Width and Height constraints and also for Space constraints.

The definition of the constraints begins with the declaration of the orientation, horizontal or vertical, and continues with the definition of the elements and their constraints from left to right or top to bottom. For example, if we want to declare a Leading Space constraint between a label and its container, the string would be **H:|-[label]**. The **H:** indicates that we are declaring horizontal constraints, the **|** character represents the container, the **-** represents a space with standard value, and the word **label** between square

brackets represents the `UILabel` object. Including a Trailing Space constraint is as simple as adding the connection on the right side `H:|[label]-|`.

The string with the visual format language defines the constraints, but there are three more attributes in the `constraintsWithVisualFormat()` method to configure the constraints and prepare the values for the string. The **control** attribute, for example, aligns the views involved. It has to be specified as a set of properties of the `NSLayoutFormatOptions` structure (or as an empty set if the elements don't require alignment). The possible values are the following:

- **AlignAllLeft** aligns elements to the left.
- **AlignAllRight** aligns elements to the right.
- **AlignAllTop** aligns elements to the top.
- **AlignAllBottom** aligns elements to the bottom.
- **AlignAllLeading** aligns elements to the leading edge (left or right depending on the human language).
- **AlignAllTrailing** aligns elements to the trailing edge (left or right depending on the human language).
- **AlignAllCenterX** aligns elements to their horizontal center.
- **AlignAllCenterY** aligns elements to their vertical center.
- **AlignAllBaseline** aligns elements to their baselines.
- **DirectionLeadingToTrailing** arranges elements from the leading edge to the trailing edge.
- **DirectionLeftToRight** arranges elements from left to right.
- **DirectionRightToLeft** arranges elements from right to left.

The **metrics** attribute, on the other hand, is a dictionary with elements that represent values in the string. It is used to replace common values by names. For example, if we have the declaration of constraints `H:|-20-[label]-20-[label]-20-|`, which creates Leading and Trailing constraints of 20 points between two labels and their container, we can replace the number 20 by a word. We provide to the **metrics** attribute a dictionary like `["space": 20]` and then change the declaration to `H:|-space-[label]-space-[label]-space-|`.

The **views** attribute does the same, but for the views. It takes a dictionary with the names for the views (the elements) and the reference to their objects. In the previous examples we used the word `label` to declare the view, so the dictionary for the **views** attribute would be something like `["label": object]`, where `object` is the property in our code that contains a reference to the `UILabel` object.

The `NSLayoutConstraint()` initializer and the `constraintsWithVisualFormat()` type method return `NSLayoutConstraint` objects but don't add the constraints to the view. The constraints are added and managed by the following methods of the `UIView` class.

`constraints()`—This method returns an array containing references to the constraints managed by the view.

`addConstraint(constraint)`—This method adds a constraint to the view. The **`constraint`** attribute is an `NSLayoutConstraint` object.

`addConstraints(constraints)`—This method adds an array of constraints to the view. The **`constraints`** attribute is the array returned by the `constraintsWithVisualFormat()` method.

`removeConstraint(constraint)`—This method removes a constraint from the view. The **`constraint`** attribute is a reference to the `NSLayoutConstraint` object that represents the constraint.

`removeConstraints(constraints)`—This method removes constraints from the view. The **`constraints`** attribute is an array containing the `NSLayoutConstraint` objects that represents the constraints to be removed.

The constraints can be added to a view on the Storyboard (through an Outlet) or to a new view created from code. The following example creates a `UILabel` object and adds it to the main view.

Listing 6-2: Adding a label with constraints

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let mylabel = UILabel(frame: CGRectZero)
        mylabel.text = "Hello World"
        mylabel.backgroundColor = UIColor.lightGrayColor()
        mylabel.translatesAutoresizingMaskIntoConstraints = false
        view.addSubview(mylabel)

        var constraints = NSLayoutConstraint.constraintsWithVisualFormat("H:|-[mylabel]", options: [], metrics: nil,
views: ["mylabel": mylabel])
        view.addConstraints(constraints)

        constraints = NSLayoutConstraint.constraintsWithVisualFormat("V:|-20-[mylabel]", options: [], metrics: nil,
views: ["mylabel": mylabel])
        view.addConstraints(constraints)
    }
}
```



```
}
```

The `UILabel` object is created the same way we did before, but because of compatibility issues with the old layout system, the views added with constraints in Auto Layout need to set a special property called `translatesAutoresizingMaskIntoConstraints` to `false`. Every time a view is added to the interface with no constraints, the system tries to guess what the constraints should be according to the value of its frame and creates the necessary constraints to put it in place. Assigning the value `false` to this property avoids the creation of these constraints and ensures that the constraints we define in code won't be in conflict with those defined by the system. We have to declare this property for every view we want to add to the interface.

With the label ready, the next step is to create and add the constraints. We have to create at least two groups of constraints for every view, one for the horizontal axis and another for the vertical axis. The example of Listing 6-2 uses the `constraints` variable to store the array of constraints and the `addConstraints()` method to add them to the view (the constraints have to be added to the container). The first method defines the horizontal constraints between the label and the main view with the string `H:|-[mylabel]`. This is saying that the left side of the label is connected to the left edge of its container by a space determined by the standard value. There are no options for this call (`()`), we didn't use any names for the metrics (`nil`), but we had to provide the dictionary `["mylabel": mylabel]` to define the name we are using to represent the label on the string (the `"mylabel"` key is the name we decided to use inside the string and the `mylabel` value is the reference to the constant where the `UILabel` object is stored).

Once the horizontal constraints are defined, we add them to the view with the `addConstraints()` method and use the variable again to store the array with the vertical constraints. The string we used for these constraints is `V:|-20-[mylabel]`. This is saying that the top of the label is connected to the top edge of the container by a space of 20 points. After these last constraints are added to the view we get the label positioned in the right place with a size determined by its intrinsic content size.



Figure 6-60: *Label with constraints added from code*

The Basics: The addition of new elements to the interface can create some conflicts with the layout and the existent constraints. It is recommended to add a view on the Storyboard that later acts as a container for the elements added from code. Instead of adding the elements to the main view, as we did in the example of Listing 6-2, we add them to their designated container. This generates a new level in the view's hierarchy that isolates the constraints already added in the Storyboard from the new ones added from code.

Defining more constraints from the previous code is easy. For example, to pin the label to the sides (left-right for horizontal constraints or top-bottom for vertical constraints) we just have to provide the string `|-[mylabel]-|`. If we don't want any space in between the views, we just have to omit the hyphens `[[mylabel]]` (this creates two `NSLayoutConstraint` objects to represent Space constraints with a value of 0). To declare a specific size for a Space constraint we have to include the number between hyphens, as demonstrated by the vertical constraint in Listing 6-2 (`|-(20-[mylabel])`). Priorities for these types of constraints are included between parentheses with the value prefixed by the `@` character `|-(20@750)-[mylabel]` (this defines a Leading Space constraint for the label with a value of 20 and a priority of 750). The Width and Height constraints for a view are declared in a similar way, including the values after the name and between parentheses `|-[mylabel(200)]`. We can also specify the relation adding the characters `>=` (Greater Than or Equal), `<=` (Less Than or Equal) and `==` (Equal) before the value (`|-[mylabel(<=200)]`).

Of course, all these combinations of characters may be used to define constraints for multiple views. The constraints and the views are declared in their natural order, as if we were defining the actual layout. For example, the string `|-[label1]-50-[label2]-|` defines the constraints to place two labels pinned to the sides of their container and separated by a space of 50 points.

The simplicity of the visual format language imposes limitations and makes impossible the definition of some types of constraints. Most of the time, the array of constraints created by the `constraintsWithVisualFormat()` method has to be complemented with individual constraints to get the layout we need. To create one `NSLayoutConstraint` object at a time we have to use the `NSLayoutConstraint()` initializer. The class provides an initializer that processes all the necessary values to configure the constraint, including the views involved, the value of the Relation, Multiplier and Constant properties, and the attributes of the views that are going to be controlled by the constraint, such as the side, dimension, etc. The framework includes the `NSLayoutAttribute` enumeration type with the following values to set these attributes:

- **Left** represents the element's left side.
- **Right** represents the element's right side.

- **Top** represents the element's top edge.
- **Bottom** represents the element's bottom edge.
- **Leading** represents the element's leading edge.
- **Trailing** represents the element's trailing edge.
- **Width** represents the element's width.
- **Height** represents the element's height.
- **CenterX** represents the element's horizontal center.
- **CenterY** represents the element's vertical center.
- **Baseline** represents the element's baseline.
- **NotAnAttribute** indicates that no attribute is provided.

The initializer takes one or two views, depending on the type of constraint, the attributes to associate the constraint, and its values. The following example defines the vertical constraints of a button with the visual format language and then adds a single constraint to center the element horizontally in its container.

Listing 6-3: Adding a single constraint

```
import UIKit
```

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let buttonAdd = UIButton(type: .System)
        buttonAdd.setTitle("Add Item", forState: .Normal)
        buttonAdd.translatesAutoresizingMaskIntoConstraints = false
        view.addSubview(buttonAdd)
        let constraints = NSLayoutConstraint.constraintsWithVisualFormat("V:|-50-[button]", options: [], metrics: nil, views:
["button": buttonAdd])
        view.addConstraints(constraints)
        let constraint = NSLayoutConstraint(
            item: buttonAdd,
            attribute: NSLayoutConstraint.CenterX,
            relatedBy: NSLayoutConstraint.Relation.Equal,
            toItem: view,
            attribute: NSLayoutConstraint.CenterX,
            multiplier: 1,
            constant: 0)
    }
}
```

```

    view.addConstraint(constraint)
}
}

```

The first value we need to provide to the `NSLayoutConstraint` initializer is a reference to the first element of the relationship (`buttonAdd`). The second value is the first element's attribute we want to consider (`CenterX`). Next, we have to declare the relation between both sides of the constraint (`Equal`). Right after that comes the second element of the relationship (`view`) and its attribute (`CenterX`). The last values required are the Multiplier and the Constant. From all this information, the initializer generates an `NSLayoutConstraint` object to represent the constraint. In the example of Listing 6-3, the object contains a constraint that goes from the `buttonAdd` element to the `view` element (the main view) and establishes their horizontal centers as equal (`CenterX` of `buttonAdd` is `Equal` to `CenterX` of `view`).

After adding that last constraint, we have all the information we need to draw the button. Its vertical position is defined by the Top Space constraint declared with the visual format language, its horizontal position is defined with the Horizontal Center in Container constraint generated by the `NSLayoutConstraint()` initializer and the `CenterX` attribute, and its size is determined by its intrinsic content size (the button's title).

The Basics: Some methods and initializers require multiple attributes and get too long to express in a single line. In cases like this, you can declare an attribute per line, as we did in Listing 6-3. This is just a visual aid and does not affect the results of the execution of the code.

In this example we used the `Equal` value of the `NSLayoutRelation` enumeration type to set the relation. The rest of the values available are `LessThanOrEqualTo` and `GreaterThanOrEqualTo`.

The visual format language and the `constraintsWithVisualFormat()` method simplify the addition of multiple constraints, but we could organize our interface with only individual constraints if we wanted to. The following example applies a Top Space, a Width, and a Horizontal Center in Container constraint to a label one by one.

Listing 6-4: Laying out elements with individual constraints

```

import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let mylabel = UILabel(frame: CGRectZero)
        mylabel.text = "Hello World"
    }
}

```

```

mylabel.backgroundColor = UIColor.lightGrayColor()
mylabel.translatesAutoresizingMaskIntoConstraints = false
view.addSubview(mylabel)

var constraint = NSLayoutConstraint(item: mylabel, attribute: NSLayoutConstraint.Attribute.Top, relatedBy:
NSLayoutConstraint.Relation.Equal, toItem: view, attribute: NSLayoutConstraint.Attribute.Top, multiplier: 1, constant: 50)
view.addConstraint(constraint)

constraint = NSLayoutConstraint(item: mylabel, attribute: NSLayoutConstraint.Attribute.Width, relatedBy:
NSLayoutConstraint.Relation.Equal, toItem: nil, attribute: NSLayoutConstraint.Attribute.NotAnAttribute, multiplier: 1, constant: 200)
view.addConstraint(constraint)

constraint = NSLayoutConstraint(item: mylabel, attribute: NSLayoutConstraint.Attribute.CenterX, relatedBy:
NSLayoutConstraint.Relation.Equal, toItem: view, attribute: NSLayoutConstraint.Attribute.CenterX, multiplier: 1, constant: 0)
view.addConstraint(constraint)
}
}

```

The Top Space constraint is added by establishing a relation between the label's **Top** attribute and the main view's **Top** attribute. The value for the distance is provided to the Constant. The Width constraint, on the other hand, only affects one element. In situations like this, the values for the second element and its attribute are not necessary (they are declared as null with the values **nil** and **NotAnAttribute**, respectively).

IMPORTANT: The Constant value could be positive or negative, depending on the type of constraint we are defining and also the order in which the elements involved are declared.

When working with single **NSLayoutConstraint** objects, it is easy to add and also remove constraints. The following example takes advantage of the **removeConstraint()** method of the **UIView** class to change the position of a label when a button is pressed.

Listing 6-5: Removing constraints

```

import UIKit

class ViewController: UIViewController {
    var mylabel: UILabel!
    var constraintLeft: NSLayoutConstraint!
    var constraintRight: NSLayoutConstraint!
    var constraintCenter: NSLayoutConstraint!
    override func viewDidLoad() {
        super.viewDidLoad()
    }
}

```

```

mylabel = UILabel(frame: CGRectZero)
mylabel.text = "Center"
mylabel.translatesAutoresizingMaskIntoConstraints = false
view.addSubview(mylabel)

let button = UIButton(type: .System)
button.setTitle("Move Label", forState: .Normal)
button.translatesAutoresizingMaskIntoConstraints = false
button.addTarget(self, action: "moveLabel", forControlEvents: UIControlEvents.TouchUpInside)
view.addSubview(button)

let constraints = NSLayoutConstraint.constraintsWithVisualFormat("V:|-50-[label]-30-[button]", options: [], metrics:
nil, views: ["label": mylabel, "button": button])
view.addConstraints(constraints)

let constraint = NSLayoutConstraint(item: button, attribute: .CenterX, relatedBy: .Equal, toItem: view, attribute:
.CenterX, multiplier: 1, constant: 0)
view.addConstraint(constraint)

constraintLeft = NSLayoutConstraint(item: mylabel, attribute: .Left, relatedBy: .Equal, toItem: view,
attribute: .Left, multiplier: 1, constant: 20)

constraintRight = NSLayoutConstraint(item: mylabel, attribute: .Right, relatedBy: .Equal, toItem: view,
attribute: .Right, multiplier: 1, constant: -20)

constraintCenter = NSLayoutConstraint(item: mylabel, attribute: .CenterX, relatedBy: .Equal, toItem: view,
attribute: .CenterX, multiplier: 1, constant: 0)
view.addConstraint(constraintCenter)
}

func moveLabel() {
    let text = mylabel.text!
    switch text {
    case "Right":
view.removeConstraint(constraintRight)
        view.addConstraint(constraintLeft)
        mylabel.text = "Left"
    case "Left":
view.removeConstraint(constraintLeft)
        view.addConstraint(constraintCenter)
        mylabel.text = "Center"
    case "Center":
view.removeConstraint(constraintCenter)
        view.addConstraint(constraintRight)
        mylabel.text = "Right"
    default:
        break
    }
}
}

```

The code of Listing 6-5 creates a label and a button. The vertical constraints for both elements are added using the visual format language, but the horizontal constraints are added from single objects created by the `NSLayoutConstraint()` initializer. The button will always be at the center of the screen, so we added a permanent Horizontal Center in Container constrain setting the `CenterX` attributes of the button and the main view as `Equal`. The position of the label depends on the current state determined by its text. It could be on the left, at the center, or on the right side of the main view. Its position always changes when the button is pressed. To do this we have defined three properties to store the necessary constraints: `constraintLeft`, `constraintRight`, and `constraintCenter`. After the definition of the constraints for the button, these three constraints are also defined and the `constraintCenter` constraint is added to the label to establish its initial position at the center. Because of this, the initial interface positions both elements at the center of the screen with the label at the top. But this changes when the user presses the button and the `moveLabel()` method is executed. The method checks the current text assigned to the label and compares it to the strings “Right”, “Left”, and “Center” to establish the current position. Then, it removes and adds the constraints to move the label to the next place. For example, if the current value is “Right”, it means that the label is at the right side of the main view and we have to move it to the left. The code removes the `constraintRight` constraint, adds the `constraintLeft` constraint, and changes the label’s text to “Left”. Figure 6-61 illustrates the three different configurations we see after the button is pressed.



Figure 6-61: *Different positions for the label*

Besides adding and deleting constraints, we can also edit their values. For this purpose, the `NSLayoutConstraint` class offers the following properties.

firstItem—This property sets or returns a reference to the first view associated with the constraint. It is of type `AnyObject`.

secondItem—This property sets or returns a reference to the second view associated

with the constraint. It is of type `AnyObject`.

firstAttribute—This property sets or returns the value of the first view’s attribute. It is an `NSLayoutAttribute` enumeration type value.

secondAttribute—This property sets or returns the value of the second view’s attribute. It is an `NSLayoutAttribute` enumeration type value.

priority—This property sets or returns the constraint’s priority value.

relation—This property sets or returns the constraint’s relation value.

multiplier—This property sets or returns the constraint’s multiplier value.

constant—This property sets or returns the constraint’s constant value.

The following example creates a button in the middle of the screen that expands or shrinks when it is pressed, depending on the current size. The first group of constraints set its vertical position and height (50), and then two single constraints set its horizontal position (`CenterX`) and its width. The value of the Width constraint is changed by the `expandButton()` method along with the button’s label every time the button is pressed.

Listing 6-6: Updating constraints

```
import UIKit
```

```
class ViewController: UIViewController {
    var button: UIButton!

    var constraintWidth: NSLayoutConstraint!

    override func viewDidLoad() {
        super.viewDidLoad()
        button = UIButton(type: .System)
        button.setTitle("Expand", forState: .Normal)
        button.backgroundColor = UIColor.lightGrayColor()
        button.translatesAutoresizingMaskIntoConstraints = false
        button.addTarget(self, action: "expandButton", forControlEvents: UIControlEvents.TouchUpInside)
        view.addSubview(button)

        let constraints = NSLayoutConstraint.constraintsWithVisualFormat("V:|-50-[button(50)]", options: [], metrics: nil,
views: ["button": button])
        view.addConstraints(constraints)

        let constraint = NSLayoutConstraint(item: button, attribute: .CenterX, relatedBy: .Equal, toItem: view, attribute:
.CenterX, multiplier: 1, constant: 0)
        view.addConstraint(constraint)

        constraintWidth = NSLayoutConstraint(item: button, attribute: .Width, relatedBy: .Equal, toItem: nil,
attribute: .NotAnAttribute, multiplier: 1, constant: 150)
        view.addConstraint(constraintWidth)
    }
}
```



```

}
func expandButton() {
    if constraintWidth.constant < 280 {
        button.setTitle("Shrink", forState: .Normal)
        constraintWidth.constant = 280
    } else {
        button.setTitle("Expand", forState: .Normal)
        constraintWidth.constant = 150
    }
}
}
}

```

We cannot only modify the constraints added from code but also those created in the Storyboard. All we have to do is control-drag a line from the constraint to the view controller and create an Outlet. This Outlet is similar to any other we have created so far, but instead of referencing a view it references the `NSLayoutConstraint` object that represents the constraint.

Usually the views are updated automatically in every drawing cycle (an automatic cycle that updates the screen several times per second), but sometimes we may need to get the current position or size of an element before this process takes place (e.g., read its `frame` property). To make sure we get the right values after the constraints are modified, we have to ask the system to update the interface. The `UIView` class includes the following methods for this purpose.

setNeedsLayout()—This method informs the system that the current layout is invalid and it should be updated in the next drawing cycle.

layoutIfNeeded()—This method forces the system to update the layout immediately.

The methods are called on the `view` property of the view controller one after another. First we call the `setNeedsLayout()` method to tell the system that the current layout is invalid and then the `layoutIfNeeded()` method to force it to refresh the interface. After this we can read any of the element's properties, such as `frame`, and we will get the values that reflect what the user sees on the screen.

Real-Life Application

The basic calculator developed in Chapter 5 was designed with absolute values. When an app like that is executed, the system automatically generates constraints based on the

position and size of the elements defined in the Storyboard. This allows us to see the interface on the screen but the elements always look the same. They don't adapt to the space available. Taking advantage of Auto Layout we can now design our calculator to adapt to every device and orientation.

We have to start by creating a new Single View Application project for Universal devices (iPhones and iPads) and add a view to the main view with a dark background color, as shown in Figure 6-62.

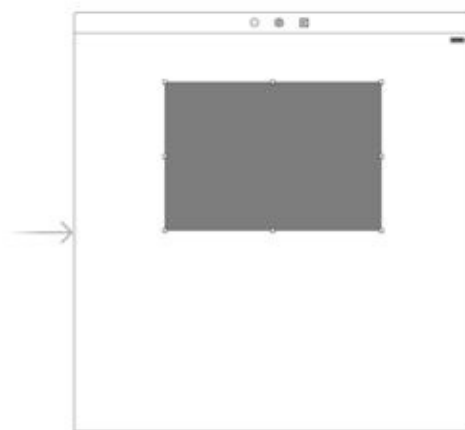


Figure 6-62: *Calculator's case*

This view represents the case of the calculator, so we want it to be the size of the screen. For this, we have to add Space constraints on each side with the value 0 and the Constrain to margins option deactivated, as shown in Figure 6-63.

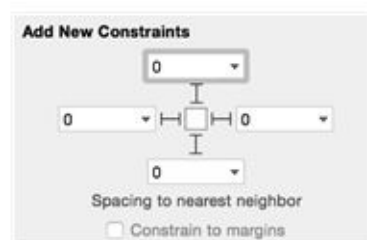


Figure 6-63: *Space constraints for the case*

After updating the frames, the view should look like Figure 6-64 below.

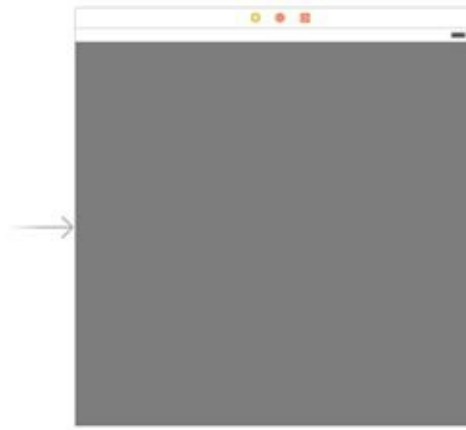


Figure 6-64: Case with space constraints

To create the display and the buttons, we have to add two more views inside the view already in the Storyboard. We have to give the views a shape that resembles a display and a keypad, and a background color that delineates the area, as shown in Figure 6-65.

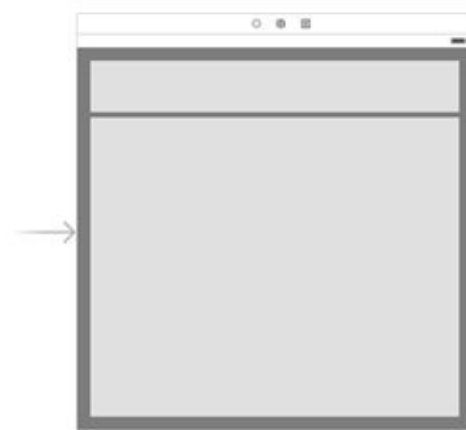


Figure 6-65: Views for the display and buttons

In Figure 6-65, we pinned the views to the sides of their container and between each other with Space constraints and standard values. We also have to add an Aspect Ratio constraint to the view for the display to let it shrink or expand but always preserving the same rectangular shape.

To create the real display we have to add a label to the display view and pin it to the sides of the view with Space constraints and a value of 8 points to give it a little margin. The label's text has to be aligned to the right and have the value “0”, as shown in Figure 6-66.



Figure 6-66: Label for the display

The last step is to create the buttons. We have to add them one by one, as we did in Chapter 5. In our example, the buttons have a gray color for the background, a white color for the title, and a shadow with an offset of 2 points, as shown in Figure 6-67 (all these options are available in the Attribute Inspector panel).

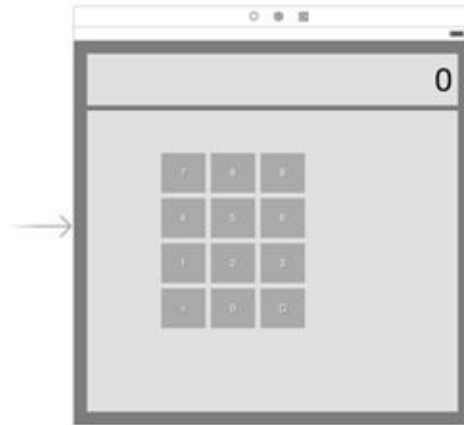


Figure 6-67: Configuring the buttons

There are different ways to organize the buttons and their constraints. For this example we have decided to assign Space constraints to every button and let them shrink or expand all together according to the space available. To add these constraints we have to select all the buttons, open the Pin menu, mark all the Space constraints at the top, select the standard value for each one of them, and activate the Equal Width and Equal Height constraints, as illustrated in Figure 6-68.



Figure 6-68: Constraints for the buttons

This will generate Space constraints for every button between each other and between the buttons and the container. The Equal Width and Equal Height constraints are added to define their sizes. They will shrink or expand but always keep equal sizes, as shown in Figure 6-69.

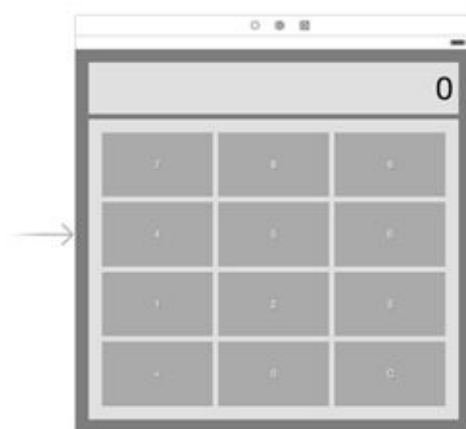


Figure 6-69: *Calculator's final design*

The interface is ready. Now all we have to do is assign the value of the buttons with numbers to their `tag` property from the Attributes Inspector panel, as shown in Figure 5-33, connect the Outlets and Actions as we did in Chapter 5, and copy the code from Listing 5-13 into our view controller. The application works exactly the same way as before, but the interface adapts to the size of the screen and its orientation.

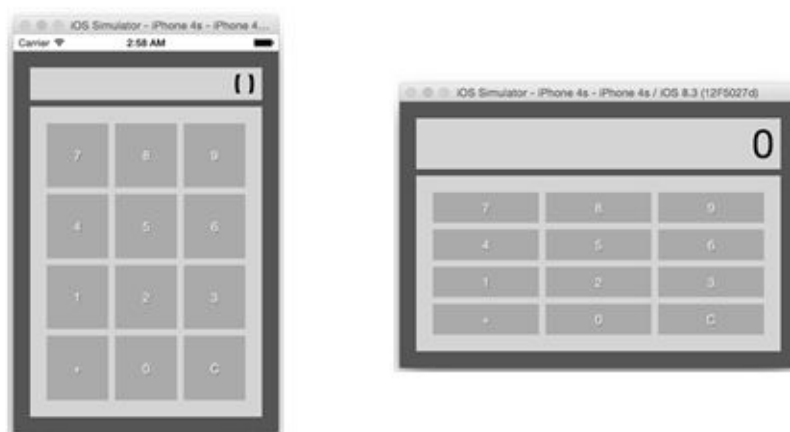


Figure 6-70: *Calculator in portrait and landscape*

Auto Layout can adapt the interface, resizing and adjusting the elements to fit the space available, but it can't modify every property or rearrange the elements to meet new requirements. If we use Auto Layout alone to design our interface, we will soon be affected by its limitations. For example, the label for the display of our calculator looks fine in landscape mode on an iPhone 4, but it gets clipped in portrait mode (Figure 6-70).

On an iPad, the label will actually look too small for a device of that size, but at the same time the buttons on an iPhone in landscape mode look squashed. Solving these issues requires coding complex routines to change not only the elements' properties but probably also restructure every single constraint on the interface or even delete or replace some of the elements. To simplify our work, Xcode includes Size Classes.

6.3 Size Classes

The way Auto Layout organizes the interface is by telling the system how every element should be laid out. It establishes rules that determine the position and size of every element on the screen. In practice this is only useful when the space available preserves its original proportions. For example, the screen of an iPhone 6 in portrait mode is slightly higher than the screen of an older iPhone in the same orientation, but the proportions between width and height are similar. Adapting the interface to these variations only requires simple transformations, such as expanding or shrinking the elements. Things change when we compare devices with very different screens, such as an iPhone and an iPad, or the same device in different orientations. In these disparate conditions, the interface has to adapt in a way Auto Layout can't handle. To get the design we want in every possible configuration, the values of the elements and constraints have to be drastically modified, some of them have to be removed, and others added. But doing this from code demands hours of extra work and it is error prone. The solution is provided by an additional system called Size Classes.

Size Classes is a classification system for the space available (this has nothing to do with the classes defined to create objects). The classification is made based on the magnitude of the horizontal and vertical dimensions of the space. The size is called *Regular* if it is big enough to fit a regular interface or *Compact* otherwise. Because of the rectangular shape of the screen, the interface is defined by two Size Classes, one for the horizontal space and another for the vertical space. For example, the Size Classes defining the space available for iPhones in portrait mode are Compact for their horizontal space and Regular for their vertical space, because in this orientation the screen's width is constrained but its height has enough space to display a normal user interface. This classification is, of course, arbitrary. Apple's developers decided what should be considered Compact and what Regular based on the screen's sizes of the devices currently available in the market.

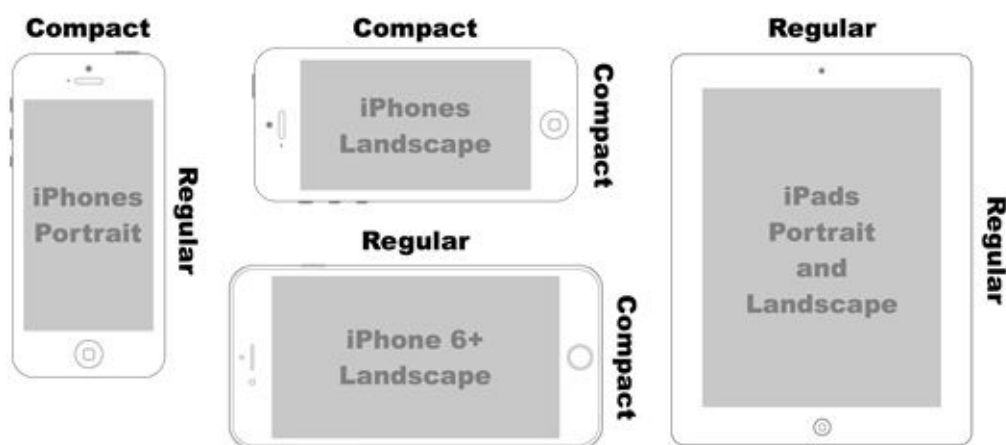


Figure 6-71: Size Classes assigned to mobile devices

The combinations presented in Figure 6-71 are actually the only four possible combinations of Size Classes: Compact width and Regular height, Compact width and Compact height, Regular width and Compact height, and Regular width and Regular height. But the system also contemplates the possibility of adapting the interface for the Size Class corresponding to only one dimension (horizontal or vertical) without caring about the other. For example, we may prepare the interface for a specific width but any height (the vertical space available doesn't matter). This is specified by two additional Size Classes: Any width and Any height. Figure 6-72 shows a grid that represents all the possible combinations.

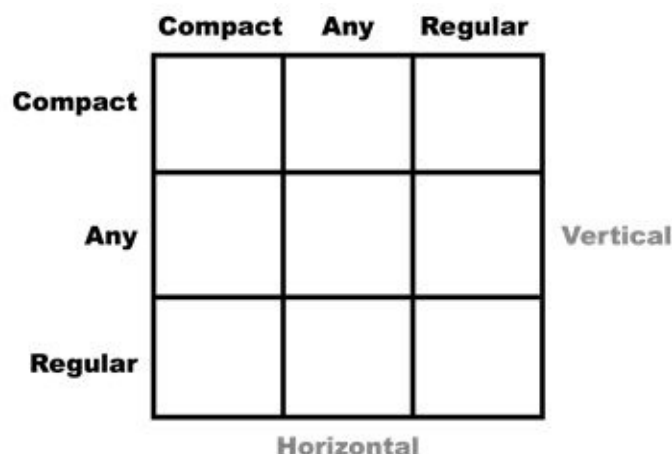


Figure 6-72: *Grid of Size Classes*

By selecting the appropriate Size Classes for the width and height it is possible to adapt our interface to any space available. In theory, we could even design our interface only for a very specific configuration. For example, we could select the Size Classes Compact width and Compact height (top-left cell in the grid of Figure 6-72) and the interface will only show up on iPhones in landscape mode (except for iPhones 6 Plus, for which the horizontal class is Regular). But the system was developed to follow a standard procedure: the default user interface is designed first for Any width and Any height (it shows in every possible configuration) and then it is adapted to other Size Classes. In consequence, every configuration shows the default interface unless something different is specified for a specific one. For example, we may declare the Top Space constraint of an element by default to be 50 points and then change it to 5 when the space's width and height are compact.

	Compact	Any	Regular	
Compact	5			Vertical
Any		50		
Regular				
	Horizontal			

Figure 6-73: Different value for the configuration Compact width and Compact height

In this example, the value 50 is declared for the configuration Any width and Any height, and then an additional value of 5 is declared for the configuration Compact width and Compact height. Every time the space available is compact in its width and height, the system applies the value 5 to the constraint. If the current space doesn't have a compact width and a compact height, then the value for the Top Space constraint is set as 50, because that's the value defined for the rest of the configurations of widths and heights.



Figure 6-74: Different values for a label's constraint

Figure 6-74 shows the simulator of an iPhone running an application that prints a label on the screen with a Top Space constraint of 50 points for Any width and Any height. The label will be at 50 points from the top of its container in every configuration and device except when the configuration is Compact width and Compact height. For this combination of Size Classes, we have defined the value of the constraint as 5, which makes the system draw the label closer to the top of the screen on an iPhone in landscape mode (except iPhones 6 Plus, for which the horizontal class in landscape is Regular).

If we want to modify the value of the constraint when the height is compact but for any width (it doesn't matter if the width is Compact or Regular), we have to declare the value

for the configuration Any width and Compact height, as shown next.

	Compact	Any	Regular	
Compact		5		Vertical
Any		50		
Regular				
	Horizontal			

Figure 6-75: Different value for Any width and Compact height

In this example, the value 5 is applied to the constraint every time the vertical Size Class is Compact, no matter what the horizontal Size Class is. Because of this, the change is now also visible in iPhones 6 Plus (the horizontal Size Class for these devices in landscape is Regular, but the vertical Size Class is still Compact, and that is the only requirement for this particular change).

IMPORTANT: The values declared for any combination of Size Classes that include the Any Size Class are called *Base Values*, and the values declared for a combination of specific Size Classes (Compact or Regular) are called *Final Values*. Final Values have precedence over Base Values and they replace the Base Values when possible. For example, if we declare a value for Any width and Compact height, as we did in the example of Figure 6-75, it will be replaced by a value declared for Compact width and Compact height, because the last one is a more specific configuration.

Grid

The advantage of Size Classes is that differentiating the types of space available we can specify the necessary changes for every screen and orientation in the same Storyboard. We just have to select the configuration we want to work on and always perform the changes in the same interface. For this purpose, Xcode introduces several tools, including a control at the bottom of the Editor Area that allows us to switch between each combination (Figure 6-3, number 2). The control shows the names of the currently selected Size Classes. The initial configuration is Any width and Any height (represented by squared main views in the Storyboard), but we can access other configurations by opening the grid and then clicking on the cell corresponding to the Size Classes we want to see.



Figure 6-76: Grid to select Size Classes

The grid represents all the combinations we have described before and in the same order. It also provides a description of the devices that present each combination at the bottom. For example, if we move the mouse over the top-left cell (Compact width and Compact height), the grid informs that at this moment the configuration corresponds to iPhones in landscape mode (except iPhone 6 Plus).

The selection of Size Classes is simple, we just have to click on the cell and all the main views in the Storyboard adapt to the selected combination. For example, if we click on the top-middle cell of the grid (Any width and Compact height), the main views on the Storyboard adopt new proportions to reflect the relation between the horizontal class and the vertical class (in this case, it resembles an iPhone in landscape mode). Figure 6-77 shows the Editor Area after this selection is made.



Figure 6-77: Main view in Any width and Compact height

The status bar color changes to blue to indicate that we are not editing the default interface anymore, and the names of the selected Size Classes replace the previous ones. To edit the interface for a different configuration all we have to do is reopen the grid and click on the corresponding cell.

Selecting the Size Classes only changes the shape of the main views on the Storyboard, but the default interface developed for Any width and Any height is still shown with the same elements and constraints. To really adapt the interface to the selected Size Classes we have to modify its components, including views, constraints, and fonts.

Adapting Constraints

There are three things we can do with constraints: modify their values, deactivate them, and add new ones. The values are actually not modified; what we do is add a new value for a specific configuration. To illustrate this process, the following example creates a simple interface with a button at the center of the screen. The button has two constraints to establish its position (Horizontal Center in Container and Top Space) and a Width constraint of 100 points to determine its size.

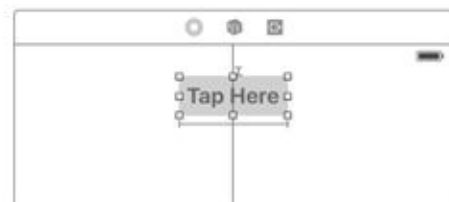


Figure 6-78: Button with constraints for the configuration Any width and Any height

This was all defined for the configuration Any width and Any height and therefore it becomes our app's default interface. If we want the interface to look different in a specific configuration, we have to select the corresponding Size Classes from the grid and perform the changes. For this example, we are going to change the value of the Width constraint to 250 when the space available has a Regular width and Any height (The devices that currently present this configuration are the iPhone 6 Plus in landscape and the iPads in both orientations).

To add a new value for a constraint in a specific configuration we first have to select the configuration from the grid (in our example we use Regular width and Any height) and then select the constraint we want to modify. The current values for the constraint are shown in the Size Inspector panel. The new value is added from the + button at the left side of the value by default (Figure 6-79, number 1).

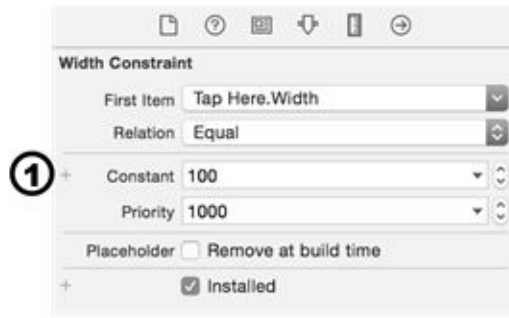


Figure 6-79: *Constraint's properties*

The figure shows the Size Inspector panel after the Width constraint is selected. When we click on the + button to add a new value, Xcode shows a popup window to select the combination of Size Classes we want to associate the value with (Figure 6-80, number 1).

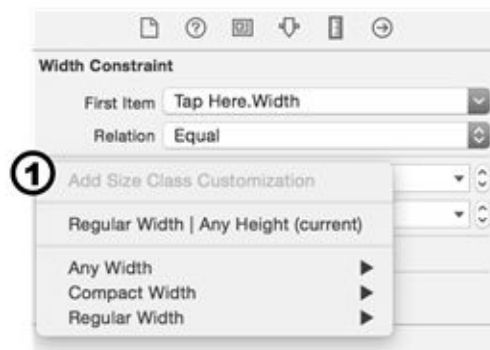


Figure 6-80: *Menu to select the Size Classes for the new value*

Usually, all we want is to add the value for the classes shown in the Storyboard, so the menu offers an option at the top to select the current configuration (in this example, Regular width and Any height), but there are also other options available to assign the value to any combination of Size Classes we want. Once the configuration is selected, the new value is added to the list (Figure 6-81, number 1).

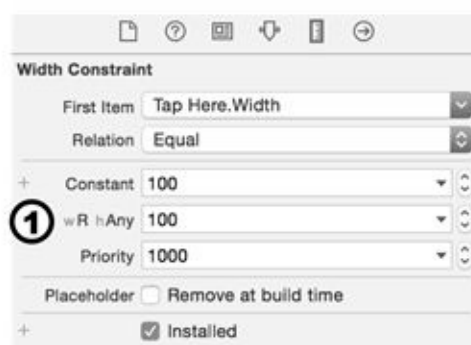


Figure 6-81: *New value for the current Size Classes*

Any value we insert in this field will be applied to the constraint when the interface is shown in a space corresponding to that specific configuration. The change is also reflected in the Storyboard as soon as the value is modified. From now on, both values are shown in the Size Inspector panel, no matter what combination of Size Classes is selected, but each value only affects the constraint in its own configuration. We could change the value of the constraint for a specific configuration and the default interface developed for Any width and Any height won't be affected. In our example we change this value to 250. Figure 6-82 shows how the interface looks on the iPhone 6 Plus simulator.



Figure 6-82: *Interface in Compact width and Regular width*

The iPhone 6 Plus has a Compact width in portrait mode and a Regular width in landscape mode. Therefore, in portrait the system applies the value 100 to the Width constraint, but when the iPhone is rotated, the value assigned for that particular configuration (Regular width) is 250 (the value will be applied also to iPads in portrait and landscape mode, because no matter the rotation the width for iPads is always Regular).

Do It Yourself: Add a button with a visible background to an empty main view. Assign to the button a Top Space constraint, a Horizontal Center in Container constraint, and a Width constraint of 100 points. It should look similar to the button of Figure 6-78. Click on the Size Classes control (Figure 6-3, number 2) and select the Regular width and Any height Size Classes from the grid (middle cell on the right). Once in this configuration, select the Width constraint and add a new value from the Size Inspector panel, as explained in the text. Assign a value of 250 to the constraint for the current configuration. Run the application on the iPhone 6 Plus simulator. You should see some similar to Figure 6-82.

The Basics: The Size Inspector panel shows an **x** button on the left side of every additional value to remove it. When a value is removed, the constraint takes back the value defined for the default interface.

Modifying values of constraints for specific Size Classes is similar to what we have done so far with Auto Layout. To perform more drastic changes we have to remove and add constraints. The constraints are not really removed but rather uninstalled. The process involves selecting the configuration where we want to replace the constraint, uninstall the constraint, and add the new one. The constraints are uninstalled from the Size Inspector panel. The following figure shows this panel after the Horizontal Center in Container constraint assigned to the button in our example is selected from the configuration Compact width and Regular height (bottom-left cell in the grid).

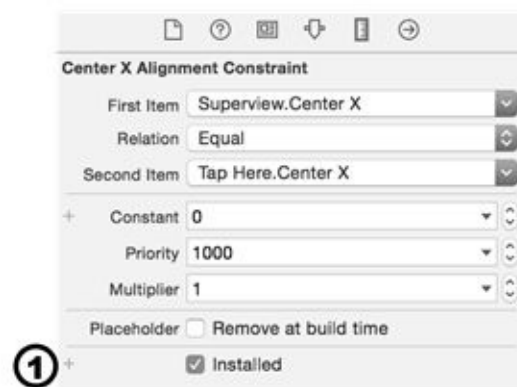


Figure 6-83: *Uninstalling constraints*

The option to uninstall the selected constraint is shown at the bottom of the panel (Figure 6-83, number 1). The option available by default corresponds to Any width and Any height. To uninstall the constraint for a specific configuration we have to add the option clicking the + button on the left. The process is similar to the one we followed before to add a new value. When we click the + button a menu pops up to select the combination of Size Classes (Figure 6-84, number 1).

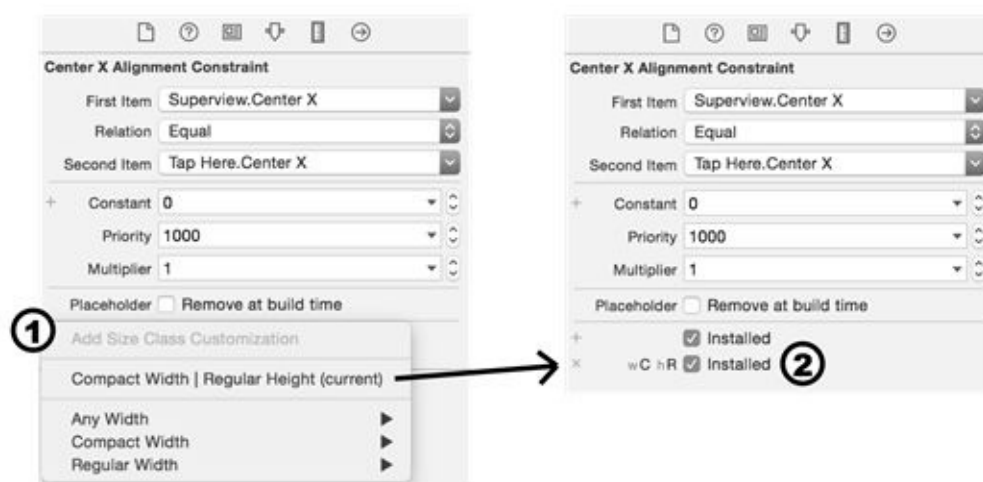


Figure 6-84: *Adding the Installed option for a specific configuration*

Once the new option is added (Figure 6-84, number 2), we have to uninstall the constraint by clicking the checked button (Figure 6-85, number 1).

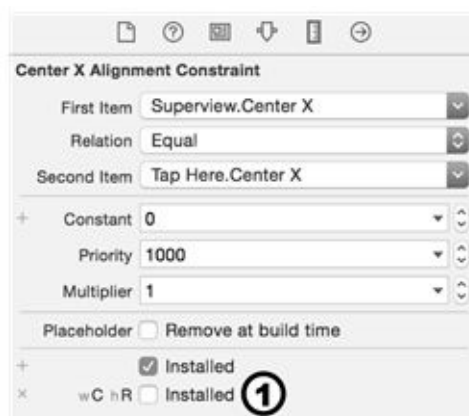


Figure 6-85: *Uninstalling a constraint*

The constraint still exists, but it is not functional in the configuration where it was uninstalled. This means that in those configurations, the element or elements affected by the constraint now need new constraints. The new constraints are added as we did before for our default interface, but they will only be available for the configuration currently selected. To illustrate the process, we have uninstalled the Horizontal Center in Container constraint for the configuration Compact width and Regular height in our example and added a Leading Space constraint to pin the button to the main view's left margin. Now the interface for this configuration looks like Figure 6-86.

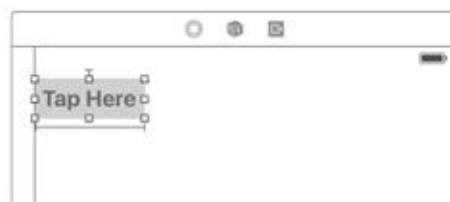


Figure 6-86: *Button with a Leading Space constraint in Compact width and Regular height*

The replacement of the Horizontal Center in Container constraint by the Leading Space constraint was performed in the configuration Compact width and Regular height, which is, at the moment, the combination of Size Classes present in every iPhone in portrait mode. If we execute this application on any of the iPhone simulators, we will see something similar to Figure 6-86 (except on iPhone 6 Plus, where the button will have a width of 250 points in landscape mode because of the additional value we added before).



Figure 6-87: *Different positions for the button in portrait and landscape*

Do It Yourself: Select the configuration Compact width and Regular height from the Size Classes grid (bottom-left cell). Select the button's Horizontal Center in Container constraint (also called Center X), and add the Installed option as explained in the text (Figure 6-84). Deactivate the option to uninstall the constraint (Figure 6-85). Add a Leading Space constraint of value 0 to the button. Run the application on the iPhone 4 or 5 simulators. You should see something similar to Figure 6-87.

The Basics: There is a very useful shortcut that lets us uninstall a constraint without having to add the option on the Size Inspector panel. All we have to do to uninstall a constraint in a specific configuration is to select the constraint and press the keys Command-Delete.

The procedure we have described above is useful to replace one constraint at a time, but becomes cumbersome when working with multiple constraints. An easy way to uninstall several constraints at once is to select the elements and then click on the Clear Constraints option from the Resolve Auto Layout Issues menu (the option that affects the selected views). The option automatically uninstalls all the constraints for the selected elements in the current configuration.

Adapting Elements

The process to uninstall an element is the same we used for constraints: we have to select the configuration we want to remove the element from, select the element and add the Installed option, but now the option is not available from the Size Inspector panel but from the Attribute Inspector panel (Figure 6-88, number 1).

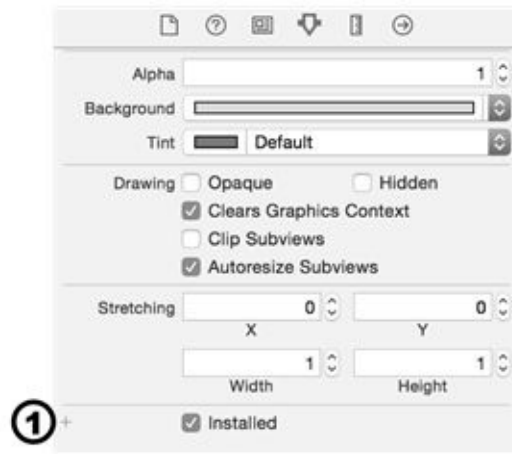


Figure 6-88: *Installed option for views*

As we did for constraints, every time we want to uninstall an element we have to select the configuration first, then select the element and click on the + button to add another option for this specific configuration (Figure 6-88, number 1). As an example, we have created a single button in the default interface that we are going to replace two buttons when the application is running on iPhones in landscape mode. The first step is to add the button with the proper constraint to the Any width and Any height configuration.

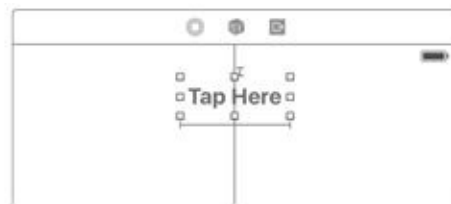


Figure 6-89: *Default interface with one button*

After this, we have to select the configuration where we want to uninstall the element (in this case, Any width and Compact height), select the element, open the Attributes Inspector panel, and click on the + button (Figure 6-88, number 1). The quick menu to select the combination of Size Classes is exactly the same shown for constraints. Once we select the configuration (Any width and Compact height), the option is added to the panel, as shown in Figure 6-90.



Figure 6-90: *New option added for the current configuration*

Now all we have to do is deactivate the option to uninstall the element.



Figure 6-91: *The element is uninstalled in the current configuration*

When an element is uninstalled from a specific configuration, not only the element but also its constraints are uninstalled. It is like the element was never defined for that configuration. The elements disappear from the interface and we are now able to modify the rest of the elements or add new ones if we want to. For our example, we added two new buttons on the sides of the main view.



Figure 6-92: *Two buttons for iPhones in landscape*

The new buttons are only added to the configuration currently selected. If we select the configuration Any width and Any height again, Xcode shows back the interface depicted in Figure 6-89. The original Tap Here button was only uninstalled for the Any width and Compact height configuration (iPhones in landscape), but it will be shown in the rest of the possible configurations.

The Basics: As well as for constraints, there is also a very useful shortcut to uninstall element without having to add the option on the Attributes Inspector panel. All you have to do to uninstall an element in a specific configuration is to select the element and press the keys Command-Delete.

These tools not only let us adapt the interface and make our app look better in every configuration, but they also allow us to add new functionality according to the circumstances. For example, we can take advantage of the possibility to uninstall and add new elements to expand the functions of our calculator when more horizontal space becomes available.

In the following example we took the calculator from Figure 6-70 and added the = button when the interface is shown on an iPhone in landscape mode. The buttons were repositioned and the C button expanded to occupy the whole column.



Figure 6-93: *New buttons in Any width and Compact height*

Figure 6-93 shows the interface in Any width and Compact height. To add the new button we selected the buttons and then click on the Clear Constraints option from the Resolve Auto Layout Issues menu. This erases the current constraints and lets us move the buttons around. We created a new button with the = character as a title, expanded the C button, and then add the new constraints for this configuration. The constraints are a little bit different this time, we have to select all the buttons and add the Top, Left, Right and Bottom Space constraints as we did before, but the constraints to determine their sizes are not the same. They all have the same width, but the height of the C button is different. Therefore, the Equal Width constraint is added to all of them, but the Equal Height constraint is added to all except the C button, which height is determined by the Space constraints defined between the button and its container.

The interface is ready. Every time the user executes the app on an iPhone in landscape mode, the system reorders the elements and adds the = button. All we have left is to add the functionality in the view controller class for this button. We have to create a new Action that calculates the total when the button is pressed. The action is reported to the rest of the code through a new property called **result**. Listing 6-7 shows the new code for the view controller.

Listing 6-7: *Calculating the result*

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var display: UILabel!
    var current: Double = 0
    var total: Double = 0
    var result: Double = 0
    let digits = 12
    var format = NSNumberFormatter()

    @IBAction func addNumber(sender: UIButton) {
        let number = Double(sender.tag)
```

```

    let text = display.text!
    result = 0
    if current == 0 {
        current = number
    } else if text.characters.count < digits {
        current = (current * 10) + number
    }
    display.text = format.stringFromNumber(current)
}

@IBAction func calculate(sender: UIButton) {
    if result != 0 {
        current = result
        result = 0
    }
    total += current
    display.text = format.stringFromNumber(total)
    current = 0
}

@IBAction func clean(sender: UIButton) {
    current = 0
    total = 0
    result = 0
    display.text = "0"
}

@IBAction func showTotal(sender: UIButton) {
    if result == 0 && current != 0 {
        result = total + current
        display.text = format.stringFromNumber(result)
        total = 0
        current = 0
    }
}
}

```

The **ViewController** class now includes the definition of the **result** property and a new Action for the = button called **showTotal()**. The code uses the **result** property instead of the **total** property to store the result. This is a way to know if the last button pressed was = and perform the action according to its value. If the value of **result** is 0 and the value of **current** is different than 0 it means that the = button was not pressed before and there is a number to add to the total. If this condition is true, the method adds the current value to **total**, assigns the result to the **result** property, and prints its value on the display. At the end of the **showTotal()** method, **total** and **current** are clean to let the user start adding numbers again.

After this, there are two possibilities: the user could insert an initial value or add a number to the last result. If the user inserts a new number, the value of `result` is cleared (see the `addNumber()` method), but if the user presses the `+` button instead, the value of `result` is considered to be the current value and used as the initial value for the addition (see the `calculate()` method).

Adapting Fonts

The tools to modify constraints and elements affect views but not their content. Some standard controls defined in the UIKit framework adapt their content to the size of their views automatically, but when the content is text they always present it with the values defined by default. To solve this issue, Xcode offers the possibility to modify the fonts for a specific configuration. The process is similar to those studied before. The Attribute Inspector panel includes a `+` button on the left side of the Font field to add a new definition for a specific configuration (Figure 6-94, number 1).

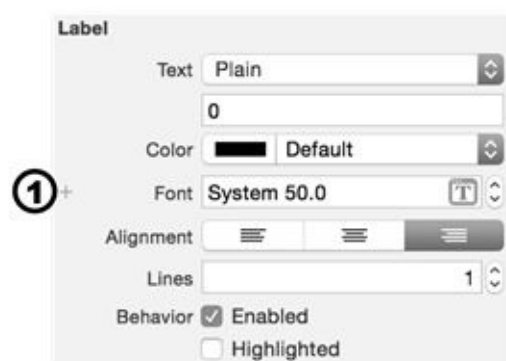


Figure 6-94: Adding fonts from the Attributes Inspector panel

After pressing this button we will see the familiar menu to select the combination of Size Classes and then we can define the font for that specific configuration.

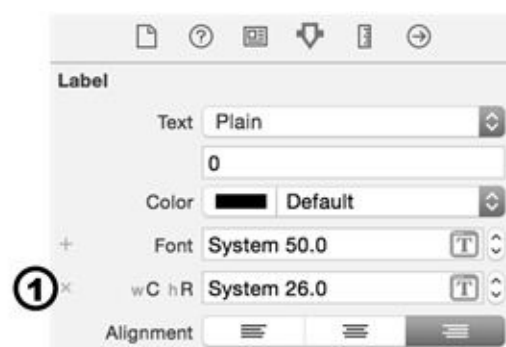


Figure 6-95: New font size for the display in Compact width and Regular height

In the example of Figure 6-95, we have defined the font size for the label that represents the display as 26 points when the interface is presented on iPhones in portrait mode (Compact width and Regular height). Now the display looks right in any orientation.

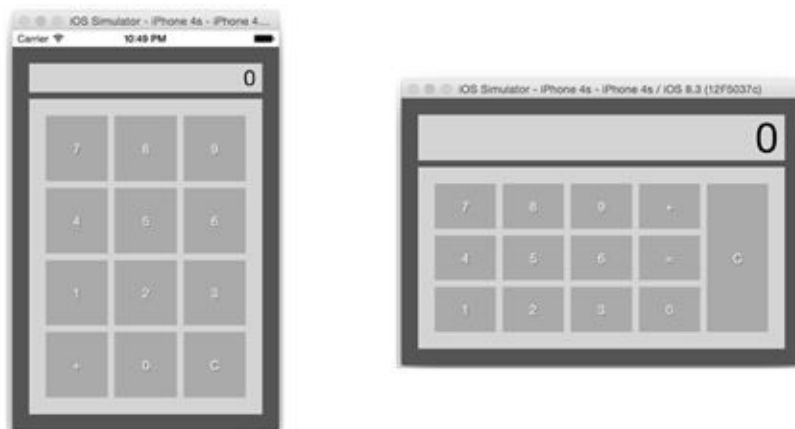


Figure 6-96: *The final design for our calculator*

Do It Yourself: We have explained how to define a new font size for the display, but you can do the same for the buttons. Select each button and add a new font with the size according to the configuration. For example, to make the interface look better on iPads you can select the combination of Size Classes Regular width and Regular height and assign a new font size of 60 points.

UITraitCollection

Every time the interface is displayed, each part of it receives information about its own space on the screen, including the horizontal and vertical Size Classes, the identification of the device and the scale of the display. This information is called *Trait Collection* and it is stored in an object of the `UITraitCollection` class. This is a simple class that includes only the four properties necessary to store the values.

horizontalSizeClass—This property returns the value of the horizontal Size Class represented by the `UITraitCollection` object. It is an enumeration of type `UIUserInterfaceSizeClass` with the values `Unspecified`, `Compact`, and `Regular`.

verticalSizeClass—This property returns the value of the vertical Size Class represented by the `UITraitCollection` object. It is an enumeration of type `UIUserInterfaceSizeClass` with the values `Unspecified`, `Compact`, and `Regular`.

userInterfaceIdiom—This property returns a string that represents the device where the app is running. It is an enumeration of type `UIUserInterfaceIdiom` with the values `Unspecified`, `Phone`, and `Pad`.

displayScale—This property returns a `CGFloat` value that represents the scale of the screen.

Views and view controllers adhere to a protocol called `UITraitEnvironment` that defines a property to store the `UITraitCollection` object and a method to report changes.

traitCollection—This property returns a `UITraitCollection` object with information of the current trait collection.

traitCollectionDidChange(previousTraitCollection: collection)—This method is called by views and view controllers that adopt the `UITraitEnvironment` protocol every time their trait collection changes. The **collection** attribute is a `UITraitCollection` object that represents the previous trait collection.

We can access these properties and methods from every view controller and also our own subclasses of `UIView` (we will learn how to create `UIView` subclasses in Chapter 24). The following example overwrites the `traitCollectionDidChange()` method to print on the console the values of the current trait collection.

***Listing 6-8:** Detecting changes in trait collection*

```
import UIKit

class ViewController: UIViewController {
    override func traitCollectionDidChange(previousTraitCollection: UITraitCollection?) {
        super.traitCollectionDidChange(previousTraitCollection)
        var horizontal: String!
        var vertical: String!

        switch traitCollection.horizontalSizeClass {
        case .Regular:
            horizontal = "Regular"
        case .Compact:
            horizontal = "Compact"
        default:
            break
        }

        switch traitCollection.verticalSizeClass {
```



```

case .Regular:
vertical = "Regular"
case .Compact:
vertical = "Compact"
default:
break
}
if horizontal != nil && vertical != nil {
print("The configuration is \(horizontal) width and \(vertical) height")
}
}
}

```

Every time the space occupied by the view controller’s view changes, the system calls the `traitCollectionDidChange()` method on the view controller. From this method we can read the information of the current trait collection in the `traitCollection` property, as we did in Listing 6-8, or get information about the previous trait collection from the `previousTraitCollection` parameter (to know about the previous conditions of the space occupied by the view).

The example of Listing 6-8 reads the `horizontalSizeClass` and `verticalSizeClass` properties of the `UITraitCollection` object and according to their values assigns the strings “Regular” or “Compact” to the corresponding variables (`horizontal` or `vertical`). At the end, we use the values of these variables to print the current configuration on the console. Sometimes the values of these properties are not defined (their value is `.Unspecified`). To take into account this possibility, we declared the `horizontal` and `vertical` variables as optionals and compare them against `nil` before to print their values.

Do It Yourself: Create a new Single View Application project. Complete the `ViewController` class with the code of Listing 6-8 and run the application. As soon as the app is executed and every time the simulator is rotated, the code prints a message on the Xcode’s console with the information of the configuration.

View controllers also adhere to the `UINavigationController` protocol that defines the following two methods to report when the trait collection is going to change (e.g., the device is being rotated).

viewWillTransitionToSize(size, withTransitionCoordinator: coordinator)
—This method is called to inform the view controller that its view is going to change size. The **size** attribute is a `CGSize` value with the new dimensions of the view, and the **coordinator** attribute is an object that adheres to the `UIViewControllerTransitionCoordinator`

protocol, which defines a few methods to let us modify the transition process.

willTransitionToTraitCollection(collection, withTransitionCoordinator: coordinator)—This method is called to inform the view controller that the trait collection is going to change. The **collection** attribute is a **UITraitCollection** object with the information of the new trait collection, and the **coordinator** attribute is an object that adheres to the **UIViewControllerTransitionCoordinator** protocol, which defines a few methods to let us modify the transition process.

The **viewWillTransitionToSize()** method is called every time the view is going to change to a different size, but the **willTransitionToTraitCollection()** method is only called when the trait collection changes. For example, when an iPhone rotates from portrait to landscape, both methods are called because the size of the view and the Size Classes change from one orientation to another. But iPads present a different situation. The size of the main view differs from portrait to landscape, but the Size Classes are both always Regular (Regular width and Regular height), so the **willTransitionToTraitCollection()** method is never called. For this reason, it is better to implement the **viewWillTransitionToSize()** method every time we need to detect the change in the space available and only use the **willTransitionToTraitCollection()** method when we have to perform a task that is related to the transition between trait collections. The following example implements the **viewWillTransitionToSize()** method to detect rotation in any device.

Listing 6-9: Detecting new orientation

```
import UIKit

class ViewController: UIViewController {
    override func viewWillTransitionToSize(size: CGSize, withTransitionCoordinator coordinator:
    UIViewControllerTransitionCoordinator) {
        super.viewWillTransitionToSize(size, withTransitionCoordinator: coordinator)
        let width = size.width
        let height = size.height
        if width > height {
            print("We are going Landscape")
        } else {
            print("We are going Portrait")
        }
    }
}
```

These methods may be used to perform changes on elements and constraints that are not possible to anticipate on Interface Builder, or just simply to adapt the content of some of the views, such as the text of a label. In Listing 6-9, we use the **viewWillTransitionToSize()**

method to report the new orientation. The method includes the `size` parameter with the dimensions the main view is going to take. Reading this parameter we can get the values of the width and height and compare them to determine the orientation. We are going to landscape mode if the width is greater than the height or to portrait mode otherwise.

The value received by the `coordinator` parameter in both methods is an object that adopts the `UIViewControllerTransitionCoordinator` protocol. This protocol defines a few methods to let us participate in the transition process. The following is the most relevant:

`animateAlongsideTransition(animation, completion: block)`—This method executes a closure during the transition process and another after the transition is completed. The **`animation`** and **`block`** attributes are the closures we want to execute. If we don't want to provide a closure for a parameter we can declare the value as `nil`.

This method sends an object to the closure with information about the transition that is taking place. This object adopts the `UIViewControllerTransitionCoordinatorContext` protocol, which defines several methods to get the information. The following are the most relevant.

`viewControllerForKey(key)`—This method returns a reference to the view controller participating in the transition. The **`key`** attribute is a constant with a `String` value that specifies which view controller we want to be returned when two are participating in the transition (one view controller is replacing another). The constants available are `UITransitionContextFromViewControllerKey` (references the view controller that is visible at the beginning of the transition), and `UITransitionContextToViewControllerKey` (references the view controller that is visible at the end of the transition).

`viewForKey(key)`—This method returns a reference to the view participating in the transition. The **`key`** attribute is a constant with a `String` value that specifies the view we want to be returned when two are participating in the transition (one view is replacing another). The constants available are `UITransitionContextFromViewKey` (references the view that is visible at the beginning of the transition), and `UITransitionContextToViewKey` (references the view that is visible at the end of the transition).

The `animateAlongsideTransition()` method must be called on the coordinator object inside the transition methods studied before every time we want to participate in the transition process. In the following example we implement this method inside `viewWillTransitionToSize()` method to adapt the size of the label for the display of our calculator.

Listing 6-10: Calculating the font size for the display

```
override func viewWillTransitionToSize(size: CGSize, withTransitionCoordinator coordinator:
UIViewControllerTransitionCoordinator) {
```

```
super.viewWillTransitionToSize(size, withTransitionCoordinator: coordinator)
```

```
coordinator.animateAlongsideTransition(nil, completion: {(context:
UIViewControllerTransitionCoordinatorContext!) in
    let controller = context.viewControllerForKey(UITransitionContextFromViewControllerKey) as!
    ViewController
        let display = controller.display
        let height = display.frame.size.height
        display.font = UIFont.systemFontOfSize(height)
    })
}
```

The `viewWillTransitionToSize()` method of Listing 6-10 was programmed to work along with the code in the `ViewController` class defined in Listing 6-7. Inside, we call the `animateAlongsideTransition()` method to perform custom tasks every time the interface is transitioning to a new state. There are two things we can do with this method: perform a task during the animation and after completion. The tasks are declared inside closures that include a parameter of type `UIViewControllerTransitionCoordinatorContext` (called `context` by default) that we can use to access the objects participating in the transition. Because our interest is to modify the font size for the display's label to adapt it to the space available and we are only able to know the dimensions of that space after the transition is over, we ignored the first parameter and only provide the closure for **completion**. Inside this closure we get a reference to our view controller from the `viewControllerForKey()` method. From this reference we also get a reference to the `display` property (the Outlet of the label) and assign it to the `display` constant. This allows us to read the value of the `height` property of the label's frame and assign it as the new font size.

Every time a transition occurs, the `viewWillTransitionToSize()` method is called in the `ViewController` object, and then the `animateAlongsideTransition()` method is executed. When the transition is over, the completion closure is executed and the label's font size is changed. Now we don't need to declare a font size for every possible configuration from Interface Builder, the label adapts automatically to the space available.

Do It Yourself: Copy the method of Listing 6-10 inside the `ViewController` class of Listing 6-7 (the view controller used for our calculator). Run the application in the simulator and perform a rotation to see how the label's text adapts to the space available for the display.

IMPORTANT: In the closure declared in Listing 6-10, we could have read the `display` property directly from the `ViewController` object using `self` (e.g., `self.display.frame.size.height`), but accessing variables and properties outside the closure from its statements can create strong references and lead to the accumulation of garbage in memory (see Memory Management, Chapter 3). That's the reason why we accessed the

`ViewController` object and its properties from the reference returned by the `viewControllerForKey()` method.

The Basics: Modifying the interface after the transition is over has its consequences. The most important is that the change is visible to the user. You should always try to determine the right interface for every possible configuration on Interface Builder and apply these methods only when there is no other option.

Orientation

Auto Layout and Size classes tell the application how the interface is going to be presented on the screen according to the space available and its characteristics (compact or regular), but the app is in charge of determining the orientations that the interface can adapt to.

The first level of control is provided by the app's settings. Xcode includes an entry at the top of the Navigator Area to access an editor where we can configure our app (Figure 5-3, number 5). When we click this entry, a project editor is shown in the Editor Area. The editor is divided in panes, with each pane allowing the configuration of different aspects of the application, including basic things like setting the possible orientations to instructions on how the code should be compiled. Figure 6-97 shows all the panes available.



Figure 6-97: Configuration panes

The pane we will probably use the most is the General pane (Figure 6-97, number 1). Here, we can set the app's version, declare the oldest iOS version our app supports, and also control the orientations available, among other things. Figure 6-98 illustrates the section of this pane where we can activate and deactivate the possible orientations.



Figure 6-98: *Orientations allowed*

The pane shows the four options available (Portrait, Portrait Upside Down, Landscape Left, and Landscape Right). If any of these options is deactivated, the interface of our app doesn't change when the device rotates to that orientation. The options are pre set by Xcode according to the device selected when the project was created. At this moment, all the orientations are activated for iPad and Universal, but the Upside Down orientation is deactivated if we have created the project for iPhones. This is to avoid the user making the mistake of answering a call with the phone upside down. But no matter what the original options are, we can change them from this pane as we wish.

Do It Yourself: Open any of the projects you have created for previous examples. Click on the entry at the top of the Navigator Area to open the project's editor (Figure 5-3, number 5). Click on each pane at the top of the editor to see the type of information you can specify for your project. Open the General pane and deactivate all the orientations except for Portrait. Run the application on the simulator. If you try to rotate the screen (holding the Command key and pressing the left and right arrows on the keyboard), you will see that the interface never rotates and only adapts to the Portrait mode.

These options affect the entire app. The rotation of the initial view controller and every view controller we add later to the Storyboard will be limited to the options activated in this pane. If we want to define the orientation for each view controller, we have to specify it in the definition of our subclass. The `UIViewController` class includes several methods we can overwrite to control the orientation. The most frequently used are the following:

`shouldAutorotate()`—This method is called by the system when a rotation took place and it needs to know if the view of the view controller should be rotated or not. It returns a Boolean value to indicate our preference.

`supportedInterfaceOrientations()`—This method is called by the system when a

rotation took place and it needs to know what orientations are supported by the view controller. It returns an enumeration of type `UIInterfaceOrientationMask` with the values `Portrait`, `LandscapeLeft`, `LandscapeRight`, `PortraitUpsideDown`, `Landscape`, `All`, and `AllButUpsideDown`.

The `shouldAutorotate()` method indicates whether or not the main view should respond to the device's rotation. When we return `true` from this method, the main view is rotated in any of the orientations available (those activated in the app's settings), but when `false` is returned, as we do in the following example, the main view is presented in the current orientation and then never rotated.

Listing 6-11: Deactivating rotation

```
import UIKit

class ViewController: UIViewController {
    override func shouldAutorotate() -> Bool {
        return false
    }
}
```

Another way to tell the system if the main view should rotate or not is from the `supportedInterfaceOrientations()` method. With this method we can select exactly the orientations available for each view controller by providing a value or a set of values of type `UIInterfaceOrientationMask`.

Listing 6-12: Allowing only landscape

```
import UIKit

class ViewController: UIViewController {

    override func supportedInterfaceOrientations() -> UIInterfaceOrientationMask {
        let orientation = UIInterfaceOrientationMask.Landscape
        return orientation
    }
}
```

In this example we return the value `Landscape` to let the interface adapt to any of the landscape orientations (left or right). The main view of this view controller will only adapt to the landscape mode. If we want to specify two or more possible orientations, we have to declare them as elements of a set. The following example allows the view controller to rotate when the device is in landscape left and portrait.

Listing 6-13: *Allowing two orientations*

```
import UIKit

class ViewController: UIViewController {
    override func supportedInterfaceOrientations() -> UIInterfaceOrientationMask {
        let orientation: UIInterfaceOrientationMask = [.LandscapeLeft, .Portrait]
        return orientation
    }
}
```

IMPORTANT: The settings defined on the General pane have precedence over the view controllers. If we deactivate the landscape modes from the pane, for example, those orientations won't be available in any view controller, no matter the values returned by their methods.

Previews

The main views in the Storyboard adapt to every combination of Size Classes, but they don't reflect the actual space available on a real device. Because of this, working with Auto Layout and Size Classes demands the execution of the application every now and then to check whether the changes we have introduced produce the expected results. Although the simulator is very fast and it usually takes no more than a few seconds to open, it presents some disadvantages. To accelerate the development process, Xcode offers a feature called *Previews* that shows representations of different devices on any orientation inside the Assistant Editor. The previews are opened from a menu at the top of the Assistant Editor (Figure 6-99, number 1).



Figure 6-99: *Option to select the content of the Assistant Editor*

The button is represented by the name of the last option selected (usually *Automatic* or *Manual*). Clicking this button opens the menu, as shown in Figure 6-100.

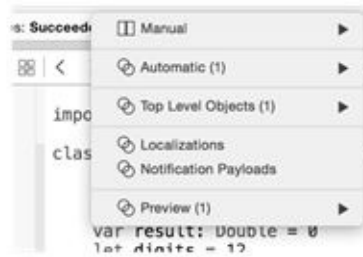


Figure 6-100: Assistant Editor's menu

The menu presents several options, including Preview. Once we click on the Preview option, the Assistant Editor shows a device displaying our interface.

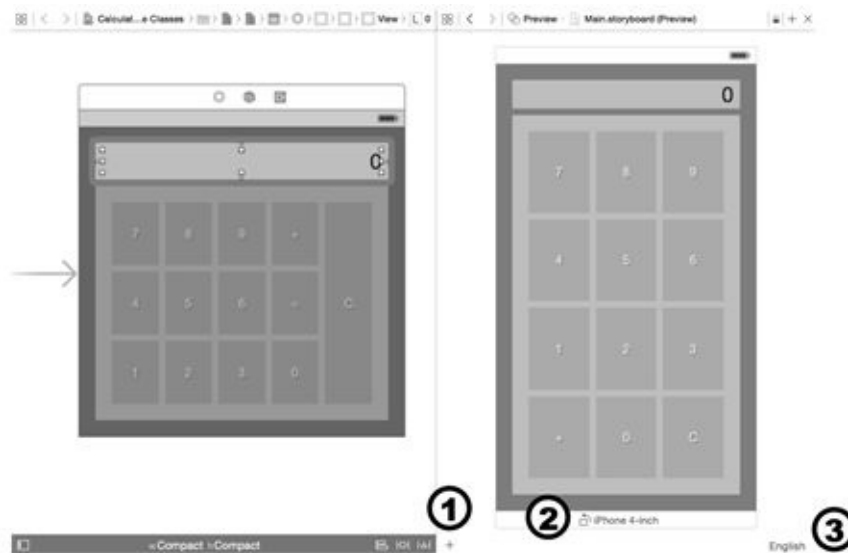


Figure 6-101: Previews in the Assistant Editor

The initial preview represents only one device, but we can add all the representations we want from a menu opened by the + button at the bottom of the Assistant Editor (Figure 6-101, number 1). There is also a button to change the orientation of the preview (Figure 6-101, number 2), and another button to duplicate the text to see how the interface will respond and adapt to other languages (Figure 6-101, number 3).

This feature doesn't replace the simulator or a real device, but it is very useful during development because it shows the changes right after we introduce them, letting us experiment and play around with our interface.

The Basics: There are other useful options in the Assistant Editor's menu, including Automatic and Manual. The Automatic option automatically selects the content to present in the Assistant Editor according to what is selected in the Editor Area. For example, when we select a main view, the Assistant Editor shows its view controller.

On the other hand, the Manual option lets us select any file in the project.

Chapter 7

Text

7.1 Input and Output

Up to this point we have been working with the `UILabel` class to print text on the screen. Although this class can manage any type of text we want, and even create labels with multiple lines of text, it can't take input from the user and it is not efficient when it comes to present extensive content. Considering these limitations, the UIKit framework includes two more classes to complement it: `UITextField` and `UITextView`.

UITextField

The `UITextField` class provides properties and methods to create a field where the user can type text. This is a rectangular box on the screen that activates the keyboard to let the user type characters or paste text in it. The class includes the `UIView` initializer to create the object.

`UITextField(frame: frame)`—This initializer creates an object of the `UITextField` class with a position and size determined by the **`frame`** attribute.

Of course, if we are creating our interface in Interface Builder using Auto Layout, it is more convenient to add the Text Field from the Object Library. Figure 7-1 shows what the option looks like.

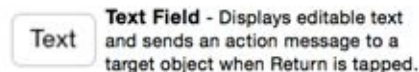


Figure 7-1: Text Field option in the Object Library

The following are the most frequently used properties included in the class.

`text`—This property sets or returns the text inside the field. It is an implicitly unwrapped optional of type `String`.

`placeholder`—This property sets or returns the message that is displayed while the field is empty. It is an optional of type `String`.

`font`—This property sets or returns the font used by the Text Field to show the text. It is an implicitly unwrapped optional of type `UIFont`.

textColor—This property sets or returns the color of the text. It is an implicitly unwrapped optional of type `UIColor`.

textAlignment—This property sets or returns the text’s alignment. It is an enumeration of type `NSTextAlignment` with the values `Left`, `Center`, `Right`, `Justified`, and `Natural`.

adjustFontSizeToFitWidth—This property sets or returns a Boolean value to indicate whether or not the size of the font should be reduced to fit the size of the field. Its value is `false` by default. If we set it to `true`, we also have to set a minimum size with the `minimumFontSize` property.

minimumFontSize—This property sets or returns the minimum size allowed for the font. It is of type `CGFloat` with a value of 0.0 by default.

clearsOnBeginEditing—This property sets or returns a Boolean value to indicate whether or not the text inside the field should be erased when the user begins editing.

borderStyle—This property sets or returns the value of the style for the field’s border. It is an enumeration of type `UITextBorderStyle` with the values `None` (no border), `Line` (single line around the field), `Bezel` (single line with shadows), and `RoundedRect` (lines with round corners).

background—This property sets or returns the image for the field’s background. It is an optional value of type `UIImage`. The image replaces the border defined by the `borderStyle` property.

disabledBackground—This property sets or returns the image for the field’s background when it is disabled. It is an optional of type `UIImage`.

clearButtonMode—This property sets or returns a value that determines when the button to clear the text should be displayed. It is an enumeration of type `UITextFieldViewMode` with the values `Never`, `WhileEditing`, `UnlessEditing`, and `Always`.

The `UITextField` class also adopts a protocol called `UITextInputTraits` that defines a set of properties to configure the input produced by the keyboard and the keyboard itself.

autocapitalizationType—This property sets or returns a value that determines when the Shift key will be automatically pressed to insert uppercase letters. It is an enumeration of type `UITextAutocapitalizationType` with the values `None` (Shift is never activated), `Words` (Shift is activated to capitalize each word), `Sentences` (Shift is activated to capitalize sentences), and `AllCharacters` (Shift is always activated).

autocorrectionType—This property sets or returns a value that determines whether or not auto-correction is enabled while typing. It is an enumeration of type `UITextAutocorrectionType` with the values `Default`, `No`, and `Yes`.

spellCheckingType—This property sets or returns a value that determines whether or

not spell checking is enabled while typing. It is an enumeration of type `UITextSpellCheckingType` with the values **Default**, **No**, and **Yes**.

keyboardType—This property sets or returns a value that determines what type of keyboard should be shown when the Text Field is tapped. It is an enumeration of type `UIKeyboardType` with the values **Default**, **ASCIICapable**, **NumbersAndPunctuation**, **URL**, **NumberPad**, **PhonePad**, **NamePhonePad**, **EmailAddress**, **DecimalPad**, **Twitter**, and **WebSearch**.

keyboardAppearance—This property sets or returns a value that determines the appearance of the keyboard. It is an enumeration of type `UIKeyboardAppearance` with the values **Default**, **Dark**, and **Light**.

returnKeyType—This property sets or returns a value that determines the title of the Return key. It is an enumeration of type `UIReturnKeyType` with the values **Default**, **Go**, **Google**, **Join**, **Next**, **Route**, **Search**, **Send**, **Yahoo**, **Done**, and **EmergencyCall**.

enablesReturnKeyAutomatically—This property sets or returns a Boolean value that determines whether or not the Return key is enabled when the user begins typing. The value by default is **false**, which indicates that the key is always enabled.

secureTextEntry—This property sets or returns a Boolean value that determines whether or not the text should be hidden (usually applied to hide passwords). The value by default is **false**.

All these properties are also available from the Attributes Inspector panel. As with any other element, we can setup the Text Field, assign the initial values for its properties from Interface Builder and later change some aspects, read its values, and control the field from code. The following example includes a Text Field in the Storyboard with the placeholder “Insert title here” and the rest of the values by default. The purpose of this app is to replace the text on the label with the one provided by the user when the Save button is pressed.

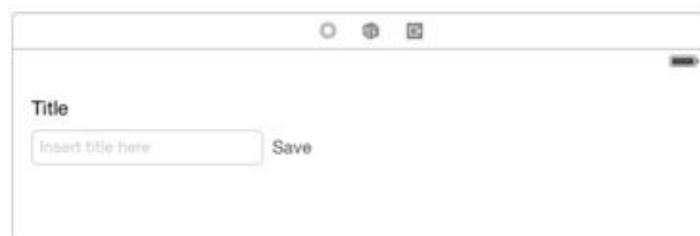


Figure 7-2: Interface to receive user’s input

To access the text inserted by the user and modify the characteristics of the Text Field from our code, we need to create an Outlet. For the example of Figure 7-2, we have to create two Outlets to connect the label and the Text Field with our view controller and an Action for the Save button.

Listing 7-1: Processing user’s input

import UIKit

```
class ViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    @IBAction func changeTitle(sender: UIButton) {
        if titleInput.text != "" {
            titleLabel.text = titleInput.text
            titleInput.text = ""
        }
    }
}
```

The code of Listing 7-1 doesn’t introduce anything new, but thanks to the Text Field we are now able to process user’s input. When the user taps on the Save button, the `changeTitle()` method is executed and the `titleLabel` and `titleInput` properties are used to replace the label’s text with the text inserted by the user. The first thing we do is to confirm that the Text Field is not empty comparing its `text` property with an empty string (`titleInput.text != ""`). When there is a text to process, the value of this property is assigned to the label’s `text` property (`titleLabel.text = titleInput.text`), effectively replacing the text on the screen with the text inserted by the user. Finally, the Text Field is cleared for a new entry (`titleInput.text = ""`).

Do It Yourself: Create a new Single View Application project. Add a label with the word “Title”, a Text Field with the placeholder “Insert title here”, and a button called “Save” to the main view, as shown in Figure 7-2. Add the necessary constraints to pin the elements to the left side of the view. Create Outlets for the label and the Text Field called `titleLabel` and `titleInput`, respectively. Create an Action for the Save button called `changeTitle()`. Complete your code with the code of Listing 7-1. Test the application in the simulator.

In the example of Listing 7-1, we compare the value inserted by the user with an empty string. If the string is not empty, we process the full string, with all the characters inserted by the user. But sometimes users unintentionally add space characters at the beginning or the end of the string. To make sure that only the relevant characters are processed, we have to trim the string, eliminating any unwanted character at the beginning or the end of the string. The `String` structure includes a convenient method defined specifically for this purpose.

stringByTrimmingCharactersInSet(characters)—This method erases the characters indicated by the **characters** attribute at the beginning and end of the string and returns a new string with the result. The **characters** attribute is an **NSCharacterSet** object that contains the set of characters we want to erase.

The **stringByTrimmingCharactersInSet()** method takes an object of the **NSCharacterSet** class to know what to remove. This class creates sets of characters for searching operations. The class includes a few type methods to create objects with standard sets. The most important are **whitespaceCharacterSet()** and **whitespaceAndNewLineCharacterSet()**, which create sets that contain invisible characters, like spaces and characters that indicate new lines. The following example removes the space characters at the beginning and ending of the string inserted by the user.

Listing 7-2: Trimming text

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    @IBAction func changeTitle(sender: UIButton) {
        var text = titleInput.text!
        let set = NSCharacterSet.whitespaceCharacterSet()
        text = text.stringByTrimmingCharactersInSet(set)
        if text != "" {
            titleLabel.text = text
            titleInput.text = ""
        }
    }
}
```

The **UITextField** class, as well as other classes that create visual controls on the screen, is a subclass of the **UIControl** class and the **UIControl** class is a subclass of the **UIView** class, therefore, **UITextField** objects have access to all their properties and methods, such as the **backgroundColor** property to change the color of their background or the **enabled** property to activate or deactivate the field (see Chapter 4, Figure 4-7). The following example modifies the **enabled** property to deactivate the Text Field after the first input.

Listing 7-3: Disabling the Text Field

```
import UIKit
```

```

class ViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    @IBAction func changeTitle(sender: UIButton) {
        if titleInput.text != "" {
            titleLabel.text = titleInput.text
            titleInput.text = ""
            titleInput.placeholder = "Text Field Deactivated"
            titleInput.enabled = false
        }
    }
}

```

The `changeTitle()` function of Listing 7-3 assigns the new text to the label as before, but now it also modifies the `enabled` property to deactivate the field and shows a new placeholder text to report the situation to the user. Once the Text Field is deactivated, the user can't type anymore.

Properties provide some level of customization, but they are not enough to control the user's input. Sometimes we need to determine what the user is allowed to insert or how the Text Field should respond. For this purpose, the `UITextField` class includes the `delegate` property to designate a delegate to the Text Field. The `UITextField` object calls methods on this delegate object to report the state of the process. The object assigned as the Text Field's delegate must conform to the `UITextFieldDelegate` protocol, which includes the following methods:

textFieldShouldBeginEditing(textField)—This method is called by the Text Field on the delegate object to know if edition should be allowed. The method includes the **textField** attribute to receive a reference to the `UITextField` object that performed the call and returns a Boolean value to indicate if edition is allowed or not.

textFieldDidBeginEditing(textField)—This method is called by the Text Field on the delegate object when the user begins editing its content. The **textField** attribute is a reference to the `UITextField` object that performed the call.

textFieldShouldEndEditing(textField)—This method is called by the Text Field on the delegate object when the user tries to switch focus to another element. The method includes the **textField** attribute to receive a reference to the `UITextField` object that performed the call and returns a Boolean value to indicate if edition should stop or not.

textFieldDidEndEditing(textField)—This method is called by the Text Field on the delegate object after the element loses focus. The **textField** attribute is a reference to the `UITextField` object that performed the call.

textField(textField, shouldChangeCharactersInRange: range, replacementString: string)—This method is called by the Text Field on the delegate object every time the user inserts or deletes a character or string. The **textField** attribute is a reference to the **UITextField** object that performed the call, the **range** attribute is an **NSRange** value that determines the range of characters on the field affected by the operation, and the **string** attribute is the new character or string inserted by the user. The method returns a Boolean value to indicate whether the text should be replaced or not.

textFieldShouldClear(textField)—This method is called by the Text Field on the delegate object to know if the Text Field should be cleared when the Clear button is pressed. The method includes the **textField** attribute to receive a reference to the **UITextField** object that performed the call and returns a Boolean value to indicate whether the Text Field should be clear or not.

textFieldShouldReturn(textField)—This method is called by the Text Field on the delegate object when the user taps the Return key on the keyboard. The method includes the **textField** attribute to receive a reference to the **UITextField** object that performed the call and returns a Boolean value to indicate if the action should be considered or not.

Usually the object assigned as the Text Field’s delegate is the view controller that controls the main view in which the Text Field was inserted. In the following example, the **ViewController** object is assigned to the **delegate** property of the Text Field and then the **textFieldShouldBeginEditing()** method is implemented. This method is another way to enable or disable a Text Field. The **UITextField** object calls it as soon as the user taps on the Text Field. If the value returned by the method is **false**, the user is not allowed to insert any text.

Listing 7-4: Assigning a delegate to the Text Field and declaring its methods

```
import UIKit

class ViewController: UIViewController, UITextFieldDelegate {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    override func viewDidLoad() {
        super.viewDidLoad()
        titleInput.delegate = self
    }
    @IBAction func changeTitle(sender: UIButton) {
        if titleInput.text != "" {
            titleLabel.text = titleInput.text
            titleInput.text = ""
        }
    }
}
```

```

}
func textFieldShouldBeginEditing(textField: UITextField) -> Bool {
    return false
}
}

```

The first thing we need to do in our view controller is to declare that the class conforms to the `UITextFieldDelegate` protocol, so the `UITextField` object knows that this object implements the `UITextFieldDelegate` protocol's methods. This is done by adding the name of the protocol after the name of the class, separated by a comma (see Protocols and Delegates in Chapter 3). Next, we have to declare the view controller object as the delegate of the Text Field, so the `UITextField` object knows what the object is that implements the protocol's methods. This is part of the Text Field's configuration, so it has to be done before anything else, and that is why we assign the `self` keyword to the `delegate` property inside the `viewDidLoad()` method (remember that `self` is a reference to the object that the code belongs to, and the `viewDidLoad()` method is a `UIViewController` method that is executed as soon as the views are loaded in memory). After the `viewDidLoad()` method is executed and `self` is assigned to the `delegate` property, the `UITextField` object can use this property to access our view controller object and report changes in the state of the Text Field by calling the corresponding methods. For example, when the user taps on the Text Field to start writing on it, the `UITextField` object calls the `textFieldShouldBeginEditing()` method on our view controller object to know if the user should be allowed to type on it (in our example, this method returns the value `false`, indicating that the user is not allowed).

The methods of the `UITextFieldDelegate` protocol are optional, which means we only have to declare the ones we need. If the method doesn't exist inside the delegate, the Text Field uses the values by default. For example, if we don't declare the `textFieldShouldBeginEditing()` method, as we did in Listing 7-4, the Text Field considers the value to be `true` and allows the user to edit the field.

The `textFieldShouldBeginEditing()` method is a very simple method, but we can declare others to control more aspects of the interaction between the user and the Text Field. For example, if we want to control what the user is allowed to insert, we can implement the `textField(textField, shouldChangeCharactersInRange: range, replacementString: string)` method. Every time the user types a character or pastes a text inside the Text Field, before accepting the entry the `UITextField` object calls this method on its delegate to know whether it should include the entry or not. Implementing this method we can process the input as it is entered.

Listing 7-5: Determining the amount of characters allowed

```

import UIKit

class ViewController: UIViewController, UITextFieldDelegate {
    @IBOutlet weak var titleLabel: UILabel!

```

```

@IBOutlet weak var titleInput: UITextField!
override func viewDidLoad() {
    super.viewDidLoad()
    titleInput.delegate = self
}
@IBAction func changeTitle(sender: UIButton) {
    if titleInput.text != "" {
        titleLabel.text = titleInput.text
        titleInput.text = ""
    }
}

func textField(textField: UITextField, shouldChangeCharactersInRange range: NSRange, replacementString
string: String) -> Bool {
    if let text = textField.text {
        let total = text.characters.count + string.characters.count
        if total > 10 {
            return false
        }
    }
    return true
}
}

```

The `textField(textField, shouldChangeCharactersInRange: range, replacementString: string)` method receives three values from the Text Field: a reference to the `UITextField` object (`textField`), an `NSRange` value that represents the range of characters in the Text Field that are going to be replaced by the new entry (`range`), and a `String` value with the new entry (`string`). Based on these values, we have to determine if the entry should be included in the field or not and return a Boolean value to indicate our decision. For example, a common procedure is to limit the number of characters allowed in the field. We add the amount of characters currently stored in the `text` property to the number of characters the user wants to introduce and return `false` if the value exceeds the maximum allowed. This procedure is followed in the example of Listing 7-5. We count the characters in the `text` property and the `string` parameter, add both values, and store the total in the `total` constant. The value of `total` is then compared against 10 (the maximum number of characters we want to allow in the Text Field for this example) and the value `false` is returned. If `total` is greater than this number, rejecting the new entry.

IMPORTANT: Every method of the `UITextFieldDelegate` protocol includes the `textField` parameter. This parameter contains a reference to the `UITextField` object that performed the call. This is useful when we don't have an Outlet to reference the Text Field or when we have multiple Text Fields in our interface, as we will see later.

The possibilities of control are limitless. We can filter the input in any way we want. For instance, we could let the user insert only numbers.

Listing 7-6: Allowing only numbers

```
func textField(textField: UITextField, shouldChangeCharactersInRange range: NSRange, replacementString string: String) -> Bool {  
    if (Int(string) != nil && textField.text != "0") || string == "" {  
        return true  
    }  
    return false  
}
```

To determine if the user inserted a number we use the `Int()` initializer. If the value returned is not `nil`, it means that the `string` parameter contains a number. But there are a few other things we have to contemplate to create this filter. The first is that, although the number 0 is a valid integer number, an integer number of two digits or more can't start with 0. Therefore we have to validate the entry only if the current value in the Text Field is not equal to 0. These two values, the value created by the initializer and the current value of the `text` property, determine two of the conditions we need to check to validate the entry, but there is one more. When the user presses a key that doesn't produce any character, such as the Delete key, the value assigned to the `string` parameter is an empty string. We also need to check this value to determine if the entry is valid or not. Inside the method of Listing 7-6 we create a logical sequence to check all these conditions. If `string` contains a number and the current value of the `text` property is different than 0, or the value of `string` is an empty string, then the whole logical sequence is true and the entry is validated.

Do It Yourself: Replace the `textField(textField, shouldChangeCharactersInRange: range, replacementString: string)` method of Listing 7-5 by the code of Listing 7-6. Run the application. You should only be allowed to type numbers in the field. If you type the number 0 first, the system won't let you insert any additional number.

Another useful method of the `UITextFieldDelegate` protocol is `textFieldShouldReturn()`. This method is called every time the user presses the Return key on the keyboard. As an example, we can use it to perform the same action as the Save button.

Listing 7-7: Responding to the Return key

```
import UIKit
```

```

class ViewController: UIViewController, UITextFieldDelegate {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    override func viewDidLoad() {
        super.viewDidLoad()
        titleInput.delegate = self
    }
    @IBAction func changeTitle(sender: UIButton) {
        assignTitle()
    }
    func textFieldShouldReturn(textField: UITextField) -> Bool {
        assignTitle()
        return true
    }
    func assignTitle() {
        if titleInput.text != "" {
            titleLabel.text = titleInput.text
            titleInput.text = ""
        }
    }
}

```

Since we are performing the same action for both, the Save button and the Return key, we move the statements to a new method called **assignTitle()** and then call this method every time the button or the key is pressed. When the user presses the Return key, the **textFieldShouldReturn()** method is called. This method executes the **assignTitle()** method to replace the label's text and then returns the value **true**. The value returned by the method tells the system whether it should implement the default behaviour after the Return key is pressed or not (usually declared as **true**), but the statements inside the method are executed anyway, no matter what is the value returned.

Other methods of the **UITextFieldDelegate** protocol are called when the edition of the Text Field begins or ends. For example, we can implement the **textFieldDidBeginEditing()** method to highlight the active Text Field. The following example changes the Text Field's background color to a light gray when the user taps on it to start typing.

Listing 7-8: Highlighting the Text Field

```

func textFieldDidBeginEditing(textField: UITextField) {
    textField.backgroundColor = UIColor.lightGrayColor()
}

```

Do It Yourself: Add the method of Listing 7-8 to the `ViewController` class of Listing 7-7 and run the application. The Text Field's background color should change as soon as you tap on it.

If our interface includes two or more Text Fields, we can take advantage of other methods like `textFieldDidEndEditing()`. Instead of performing a task when the Text Field is tapped, with this method we can do it when the Text Field loses focus (the user taps somewhere else). For example, if we expand our interface to include another Text Field, we can highlight the one currently selected. Figure 7-3 shows what an interface like this would look like.

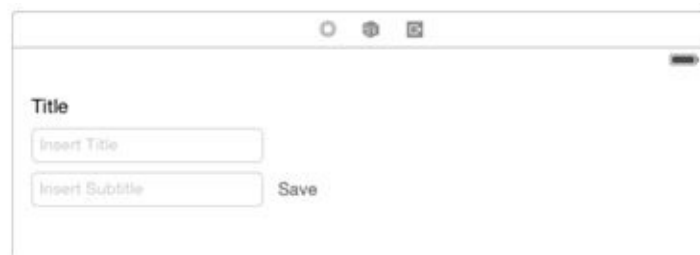


Figure 7-3: *Interface with two Text Fields*

In this example, the original Text Field is used to introduce the title and the new Text Field is used for the subtitle. The Save button still performs the same task, but the view controller has to be updated with another Outlet for the new Text Field and an improved Action for the button that includes the values of both fields.

Listing 7-9: *Working with two Text Fields*

```
import UIKit

class ViewController: UIViewController, UITextFieldDelegate {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    @IBOutlet weak var subtitleInput: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
        titleInput.delegate = self
        subtitleInput.delegate = self
    }

    @IBAction func changeTitle(sender: UIButton) {
        if titleInput.text != "" && subtitleInput.text != "" {
            titleLabel.text = titleInput.text! + " - " + subtitleInput.text!
            titleInput.text = ""
        }
    }
}
```



```

    subtitleInput.text = ""
}
}
func textFieldDidBeginEditing(textField: UITextField) {
    textField.backgroundColor = UIColor.lightGrayColor()
}
func textFieldDidEndEditing(textField: UITextField) {
    textField.backgroundColor = UIColor.whiteColor()
}
}

```

In the view controller of Listing 7-9 we call the new Outlet `subtitleInput`. This new property is later used in the `viewDidLoad()` method to assign the `ViewController` object as the delegate of the second Text Field. Now every time something happens in any of the Text Fields in our interface, both `UITextField` objects call the protocol's methods in our view controller. In this example, we have implemented the `textFieldDidBeginEditing()` method and the `textFieldDidEndEditing()` method to change the background colors. When the user taps on one of the Text Fields to start typing, its background becomes gray, as it did in the previous example, but now when the user taps on the other Text Field, the first one loses focus and it calls the `textFieldDidEndEditing()` method to report what happened to the delegate. From this method we change the Text Field's background color back to white to reflect the new state on the screen.

Do It Yourself: Add a new Text Field to the interface, as shown in Figure 7-3. Connect this element to the view controller with an Outlet called `subtitleInput`. Update the `ViewController` class with the code of Listing 7-9. Run the application and tap on one Text Field and another to see how their background colors change.

Working with two or more Text Fields in the same view and connected to the same delegate presents a challenge. The methods called on the delegate are always the same, no matter which Text Field performed the call. This is the reason why all the methods defined in the `UITextFieldDelegate` protocol include the `textField` parameter with a reference to the `UITextField` object. We can use this reference to modify the object, as we did before, but also to identify the Text Field we are working with. One simple way to do this is to assign a value to the object's `tag` property. We used this property before for the buttons of the calculator developed in previous chapters. A different value is assigned to the `tag` property of every Text Field from the Attributes Inspector panel and then the property is read from our code to know which Text Field called the method. For example, the following view controller implements the `textField()` method to establish different limits to the number of characters the user can type on each field. To know which limit to apply, the method identifies the field reading its `tag` property.

Listing 7-10: Identifying Text Fields

import UIKit

```
class ViewController: UIViewController, UITextFieldDelegate {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    @IBOutlet weak var subtitleInput: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
        titleInput.delegate = self
        subtitleInput.delegate = self
    }

    @IBAction func changeTitle(sender: UIButton) {
        if titleInput.text != "" && subtitleInput.text != "" {
            titleLabel.text = titleInput.text! + " - " + subtitleInput.text!
            titleInput.text = ""
            subtitleInput.text = ""
        }
    }

    func textField(textField: UITextField, shouldChangeCharactersInRange range: NSRange, replacementString string: String) -> Bool {
        var maximum = 0
        if textField.tag == 1 {
            maximum = 10
        } else {
            maximum = 15
        }
        if let text = textField.text {
            let total = text.characters.count + string.characters.count
            if total > maximum {
                return false
            }
        }
        return true
    }
}
```

This example assumes that the values of the Text Fields' **tag** properties were defined as 1 and 2, respectively (the value for the second Text Field could be 2 or any other value different than 1). The **textField()** method checks this value to know which field is making the

call (`textField.tag == 1`), sets the value for the `maximum` variable, and then compares the result against the total amount of characters, as we did before, to validate the entry (`total > maximum`).

Do It Yourself: Select each Text Field in the Storyboard and assign the corresponding values to their `tag` properties from the Attributes Inspector panel, as shown in Figure 5-33 (1 for the title and 2 for the subtitle). Update the `ViewController` class with the code of Listing 7-10. Run the application and insert text on the fields. The title should be limited to 10 characters and the subtitle to 15.

UITextView

The `UITextView` class provides properties and methods to create a scrollable view that presents multiline text. The element can display text on the screen and also receive input from the user. If the size of the text surpasses the size of the view, the view allows the user to scroll its content. The class includes the `UIView` initializer to create the object.

UITextView(frame: frame)—This initializer creates an object of the `UITextView` class with a position and size determined by the **frame** attribute.

Again, if we are creating our interface in Interface Builder, using Auto Layout is more convenient to add the Text View from the Object Library.

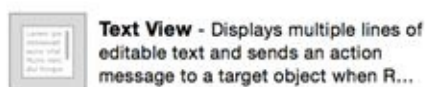


Figure 7-4: Text View option in the Object Library

The following are the most frequently used properties included in this class.

text—This property sets or returns the text in the view. It is an implicitly unwrapped optional of type `String`.

font—This property sets or returns the font used by the Text View to display the text. It is an implicitly unwrapped optional of type `UIFont`.

textColor—This property sets or returns the color of the text. It is an implicitly

unwrapped optional of type `UIColor`.

textAlignment—This property sets or returns the text’s alignment. It is an enumeration of type `NSTextAlignment` with the values `Left`, `Center`, `Right`, `Justified`, and `Natural`.

textStorage—This property returns an object that represents the attributed text in the Text View. The object is of a subclass of `NSMutableAttributedString` called `NSTextStorage`. The code provided by the `NSMutableAttributedString` class allows us to edit the text’s attributes and the code provided by the `NSTextStorage` class notifies the changes to the Text View as soon as they are made.

editable—This property sets or returns a Boolean value that indicates whether the user is allowed to edit the text inside the Text View or not.

selectable—This property sets or returns a Boolean value that indicates whether the user is allowed to select text inside the Text View or not.

selectedRange—This property sets or returns the range of the text selected inside the Text View. It is of type `NSRange`.

scrollRangeToVisible(range)—This method scrolls the view to show on the screen the text that corresponds to the range specified by the range attribute. It takes a value of type `NSRange`.

The `UITextView` class can also designate a delegate to report the state of the process. The object assigned as the Text View’s delegate must conform to the `UITextViewDelegate` protocol, which includes the following methods.

textViewShouldBeginEditing(textView)—This method is called by the Text View on the delegate object to know if edition should be allowed. The method includes the **textView** attribute to receive a reference to the `UITextView` object that performed the call and returns a Boolean value to indicate if edition is allowed or not.

textViewDidBeginEditing(textView)—This method is called by the Text View on the delegate object when the user begins editing its content. The **textView** attribute is a reference to the `UITextView` object that performed the call.

textViewShouldEndEditing(textView)—This method is called by the Text View on the delegate object when the user tries to switch focus to another element. The method includes the **textView** attribute to receive a reference to the `UITextView` object that performed the call and returns a Boolean value to indicate if edition should stop or not.

textViewDidEndEditing(textView)—This method is called by the Text View on the delegate object when the element loses focus. The **textView** attribute is a reference

to the `UITextView` object that performed the call.

`textView(textView, shouldChangeTextInRange: range, replacementText: text)`—This method is called by the Text View on the delegate object every time the user inserts or deletes a character or string. The **`textView`** attribute is a reference to the `UITextView` object that performed the call, the **`range`** attribute is an `NSRange` value that determines the range of characters on the view affected by the operation, and the **`text`** attribute is the new character or string inserted by the user. The method returns a Boolean value to indicate whether the text should be replaced or not.

`textViewDidChange(textView)`—This method is called by the Text View on the delegate object when the user changes the text or any of its attributes. The method includes the **`textView`** attribute to receive a reference to the `UITextView` object that performed the call.

`textViewDidChangeSelection(textView)`—This method is called by the Text View on the delegate object when the user selects text. The method includes the **`textView`** attribute to receive a reference to the `UITextView` object that performed the call.

As always, we can provide the values for all the properties from the Attributes Inspector panel, including the text to show inside the view. When a Text View is selected, the panel includes an option at the top to indicate the type of text we want to assign to the view: Plain or Attributed. If we select the Attributed option, additional buttons are incorporated to the panel to assign the attributes.

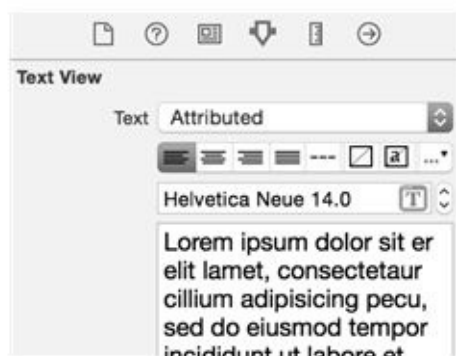


Figure 7-5: *Attributed text for the Text View*

From the Attributes Inspector panel we can change every aspect of the whole text or part of it, but this only sets the Text View’s initial value. If we want to modify the view’s content or replace it when the app is running, we have to do it from code. The `UITextView` class offers two properties to access the object’s content: **`text`** and **`attributedText`**. With the **`text`** property we can set plain text with general attributes and with the **`attributedText`** property we can assign attributed text to the view. These properties store different types of texts but are connected with each other. If a value is assigned to one property, the text on the other property is automatically modified. This allows us to use the Text View to display any

type of text we want at any moment. If we want to display plain text, we assign a `String` value to the `text` property, but if we want to display text that presents characters or ranges of characters with different attributes we assign an `NSAttributedString` object to the `attributedText` property.

The problem with this approach is that the `NSAttributedString` class creates immutable objects, which means that every time we want to change any of the attributes in the attributed text we have to create and assign a new object. This issue was solved with the inclusion of an additional property called `textStorage`. This property stores an object of the `NSMutableAttributedString` class that we can modify as we do with any other `NSMutableAttributedString` object (see Chapter 4). The advantage of using this property is that we can modify the attributes on the same object and the changes performed are reflected automatically in the Text View.

The Basics: The `textStorage` property is actually of type `NSTextStorage`, a subclass of `NSMutableAttributedString` that was defined to make easy to handle and update attributed text. For more information, visit our website and follow the links for this chapter.

The following example introduces a new interface with a button and a Text View. The purpose of the button is to assign new attributes to the characters selected by the user.



Figure 7-6: Text View with plain text

The view controller for an application like this is very simple. We just need an Outlet to reference the Text View and an Action for the button. Because we want to modify the attributes of the text (change the color of the characters selected by the user) we have to access the `textStorage` property and call its `addAttribute()` method.

Listing 7-11: Adding attributes to the attributed text of a Text View

```
import UIKit
```

```
class ViewController: UIViewController {  
    @IBOutlet weak var message: UITextView!
```

```

@IBAction func selection(sender: UIButton) {
    let attributedText = message.textStorage
    attributedText.addAttribute(NSForegroundColorAttributeName, value: UIColor.redColor(), range:
message.selectedRange)
}
}

```

The `addAttribute()` method is one of the methods offered by the `NSMutableAttributedString` class (see Chapter 4). The method takes three values: the keyword that represents the attribute we want to add, the value for the attribute, and the range of characters that are going to receive the attribute. Since we want to modify the color of the text selected by the user, we declare the attribute `NSForegroundColorAttributeName` and a `UIColor` object with the values for the color we want to assign. The range of characters to modify in this example is the range of characters currently selected. This value is provided by the Text View's `selectedRange` property. Every time the user selects text in the Text View, the value of this property changes to reflect the location of the selected characters. The `selection()` method of Listing 7-11 calls the `addAttribute()` method with all these values to change the color of the text. Because of this, every time the Change Color button is pressed, the color of the selected text changes to red.

Do It Yourself: Create a new Single View Application project. Add a button called Change Color at the top of the view, and a Text View and all the necessary constraints to get an interface similar to Figure 7-6. Connect the Text View to your view controller with an Outlet called `message` and the button with an Action called `selection()`. Complete your code with the code of Listing 7-11. Run the application in the simulator, double click a word to select it, and press the Change Color button. The color of the selected text should change to red.

Of course, we don't have to always affect the characters selected; we can specify any range of character we want. For example, we could search for specific words in the text and change the attributes for only those characters. In the following example we look for the first occurrence of the text "John" and change the color of its characters to blue.

Listing 7-12: Adding attributes to a range of characters

```

import UIKit

class ViewController: UIViewController, UITextViewDelegate {
    @IBOutlet weak var message: UITextView!

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}

```

```

    message.delegate = self
}

@IBAction func selection(sender: UIButton) {
    let attributedText = message.textStorage
    attributedText.addAttribute(NSForegroundColorAttributeName, value: UIColor.redColor(), range:
message.selectedRange)
}

func textViewDidChange(textView: UITextView) {
    let text = textView.text as NSString
    let range = text.rangeOfString("John", options: .CaseInsensitiveSearch)

    let attributedText = textView.textStorage
    attributedText.addAttribute(NSForegroundColorAttributeName, value: UIColor.blueColor(), range: range)
}
}

```

The code of Listing 7-12 uses the same interface, but this time we assign the view controller as the Text View’s delegate to implement the `textViewDidChange()` method. The method is called by the `UITextView` object every time the content of the Text View changes. This way, when the user types or erases a character, our code is aware of it and can process the new value. In this example, we search for the word “John” and change the color of its characters to blue if we find it. The process is similar to the one we used before for the button’s Action, but since the `addAttribute()` method takes an `NSRange` value we had to convert the string to an `NSString` object before to call the `rangeOfString()` method to get the range (see `NSRange` in Chapter 4).

The Basics: The rest of the `UITextViewDelegate` protocol’s methods are similar to the methods studied before for the `UITextField` class, except for how the Return key is handled. The `UITextViewDelegate` protocol doesn’t have a method to process the Return key because the key is used to generate new lines of text.

Keyboard

iOS has a particular way to control the keyboard. It doesn’t consider the keyboard as a tool, but rather as a way to detect events produced by the user, and therefore it keeps it on screen as long as the element that made it appear is still capable of processing the events coming from it. To dismiss the keyboard, we have to tell the system that the element is no longer in charge of processing its events, and this is done by altering the chain of response.

When an event occurs, such as a tap on the screen, the system has to determine which is

the element that is going to process the event. The position of the graphics on the screen does not always reflect this responsibility. For example, we could have two views that overlapped but only the view on the back is capable of processing touch events. If we let the view on the front get the events, the view on the back will never know that the user tapped the screen. To solve this problem, the system considers the elements hierarchy and creates a virtual chain in which some elements in certain positions, called *Responders*, receive the event and then decide whether to process it or deliver it to the next element in the chain. Some events, like key events, are sent to specific elements called *First Responders*.

First Responders are designated by the system or our code. This is actually what the system does when the user taps on a Text Field or a Text View: it declares the element as the First Responder, so all the events are first sent to it. And this is also how the system manages the keyboard. The keyboard is opened when an element that is capable of handling key events becomes the First Responder, and it is closed when the element ceases to be the First Responder (it resigns as First Responder).

The UIKit framework defines the **UIResponder** class to manage First Responders and respond to events. The following are its most frequently used methods.

becomeFirstResponder()—This method designates the element as the First Responder.

resignFirstResponder()—This method notifies the element that it has to resign its condition of First Responder.

isFirstResponder()—This method returns a Boolean value that indicates whether the element is the current designated First Responder or not.

touchesBegan(touches, withEvent: event)—This method is called by the system in Responder objects (views and view controllers) when the user starts touching the screen.

touchesMoved(touches, withEvent: event)—This method is called by the system in Responder objects (views and view controllers) when the fingers touching the screen move.

touchesEnded(touches, withEvent: event)—This method is called by the system in Responder objects (views and view controllers) when the user stops touching the screen.

touchesCancelled(touches, withEvent: event)—This method is called by the system in Responder objects (views and view controllers) when a touch event is cancelled by the system.

The **UIResponder** class is a basic class that most of the classes inherit from. Because of this, all the elements created from UIKit classes have access to its properties and methods and are capable of handling events. If we need to open the keyboard, we just have to call the **becomeFirstResponder()** method on an element capable of handling key events (which produces the same effect as the user tapping on the element). Closing the keyboard is as simple as calling the **resignFirstResponder()** on the element currently assigned as First Responder. The following example shows how to implement these two methods. This view controller assumes that we have an interface with a Text Field, a label and a button, as the one presented in Figure 7-2.

Listing 7-13: *Opening and closing the keyboard from code*

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    @IBAction func changeTitle(sender: UIButton) {
        if titleInput.text != "" {
            titleLabel.text = titleInput.text
            titleInput.text = ""
            titleInput.resignFirstResponder()
        } else {
            titleInput.becomeFirstResponder()
        }
    }
}
```

This example is similar to the one presented before in Listing 7-1. We have an Outlet for the label and another for the Text Field. When the user presses the Save button, the **changeTitle()** method checks the content of the Text Field as before, but this time it performs an action either if the text is empty or not. If something was inserted into the field, the method assigns this value to the label and then executes the **resignFirstResponder()** method to force the Text Field to resign its condition as First Responder. When the system detects that there is no First Responder that can handle key events, it closes the keyboard. On the other hand, if the **changeTitle()** method can't find any text inside the Text Field, it executes the **becomeFirstResponder()** method, turning the Text Field into the First Responder and compelling the system to open the keyboard.

Do It Yourself: Create an interface similar to the one presented in Figure 7-2. Connect the label, the Text Field and the button with the corresponding Outlets and Action, as we did for the example of Listing 7-1. Complete your view controller with

the code of Listing 7-13. Run the application in the simulator and click the Save button. The keyword will pop up and the Text Field will show the cursor. Write something in the Text Field and press the Save button again. The keyboard should be dismissed.

The Basics: The simulator includes the same keyboard of a real device, but it can also work with the computer's keyboard. To select one keyboard or another, open the Hardware menu at the top of the screen, go to Keyboard and select the option of your preference.

The `becomeFirstResponder()` method is not always necessary. Usually the user taps on the element when it needs to type something on it and this action automatically makes the element the First Responder, compelling the system to open the keyboard. But the execution of the `resignFirstResponder()` method is required every time we want to dismiss the keyboard because there is no automatic action to do it. In our previous example, we called this method when the Save button was pressed, but a more intuitive interface will also close the keyboard when the user taps somewhere else on the screen. These types of events are detected by the event-handling methods included in the `UIResponder` class.

***Listing 7-14:** Dismissing the keyboard when the user taps the screen*

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    @IBAction func changeTitle(sender: UIButton) {
        if titleInput.text != "" {
            titleLabel.text = titleInput.text
            titleInput.text = ""
        }
    }
    override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {
        super.touchesBegan(touches, withEvent: event)
        titleInput.resignFirstResponder()
    }
}
```

The `touchesBegan()` method implemented in Listing 7-14 is called on the view controller when the user touches the screen. We overwrite this method in our view controller to perform custom tasks. In this case, we use it to call the `resignFirstResponder()` method on the

Text Field to dismiss the keyboard. As a result, the keyboard is opened when the user taps on the Text Field and is closed when the user taps somewhere else (except on the Save button, because the button provides its own response to the event).

IMPORTANT: The UIKit framework provides the `UIGestureRecognizer` class and several subclasses to control typical events and recognize complicated gestures, such as a swipe, a rotation of the finger, a pinch, etc. Except for specific cases like those studied in this chapter, you will barely use the `UIResponder` event-handling methods. We will study `UIGestureRecognizer` in Chapter 12.

Calling the `resignFirstResponder()` method on an element only removes the status of First Responder for that element. If we have several elements that can become First Responders, such as the multiple Text Fields we had in the example of Figure 7-3, we have to call the method on each one of them. This can get messy and be error prone sometimes. For this reason, the `UIView` class includes a special method that looks at the view and its subviews to find the current First Responder.

endEditing(value)—This method looks for the First Responder in a view and its subviews and asks it to resign its condition. The **value** attribute is a Boolean value to indicate if the element should be forced to resign or not.

Listing 7-15: Finding the First Responder

import UIKit

```
class ViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var titleInput: UITextField!
    @IBAction func changeTitle(sender: UIButton) {
        if titleInput.text != "" {
            titleLabel.text = titleInput.text
            titleInput.text = ""
        }
    }
    override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {
        super.touchesBegan(touches, withEvent: event)
        view.endEditing(true)
    }
}
```

This example uses the same view controller of Listing 7-14 but instead of calling the `resignFirstResponder()` method on each Text Field we call the `endEditing()` method on the main view (the `view` property) and let it find the current First Responder inside the view and force it to resign its condition. It is the same process, but now we don't have to call the `resignFirstResponder()` method for every Text Field on the view.

IMPORTANT: In the examples of this chapter we positioned the elements at the top of the screen. This guarantees that the elements are going to be visible when the keyboard comes up, but that is not always possible. Sometimes the keyboard hides the field the user wants to write in and we have to move the element to a visible area of the screen. The most common and easy way to do this is to embed the interface in a scrollable view. We will study how to create scrollable views from the `UIScrollView` class in Chapter 11.

Chapter 8

Images and Icons

8.1 Images

Images are used for everything in mobile applications, from the addition of backgrounds and patterns to the creation of customized controls. Adding images to our project is very simple: the files have to be dragged from Finder to the Xcode's interface and then included in the Storyboard or loaded from code, but there is one important issue we have to consider first: images are stored in files with a resolution in pixels, while our interface is defined in points.

As we have seen in Chapter 4, the screens of Apple devices have different resolutions and scales. In some devices, one pixel represents one point and in others more. At this moment, three scales have been defined: 1x, 2x, and 3x. The 1x scale defines one point as one pixel, the 2x scale defines 1 point as a square of 2 pixels, and the 3x scale defines one point as a square of three pixels (see the full list of devices, resolutions and scales in Chapter 4). Because of this, every time we want to introduce images in our interface we have to consider the conversion between pixels and points. For example, if we have an image of 300 pixels wide and 400 pixels tall, in a device with a scale of 1x the image will almost occupy the whole screen, but in a device with a scale of 2x the image will look half its size. The image is actually occupying the same space on the screen, 300 by 400 pixels, but because of the higher resolution this amount of pixels represents a smaller area on the screen in devices with scales of 2x or 3x, as shown in Figure 8-1.

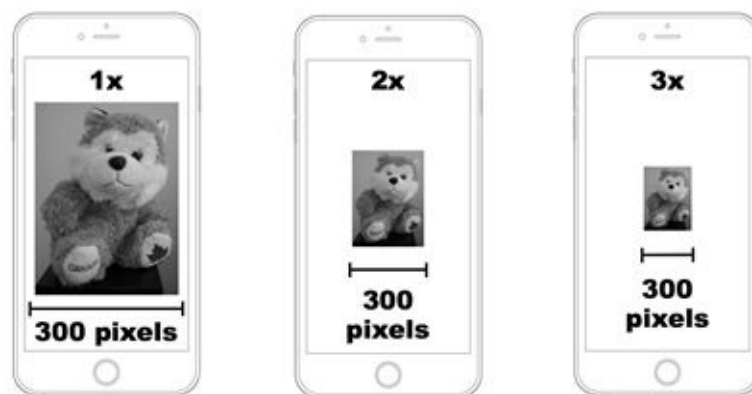


Figure 8-1: *The same image in devices with different scale*

There are two solutions to this problem. The most simple is to scale up a small image in devices with higher resolution or scale down a big image in devices with lower resolution. For example, we can expand an image of 300 x 400 pixels to 600 x 800 pixels and make it look like the same size in a screen with a scale of 2x (a space of 300 x 400 points represents 600 x 800 pixels at this scale), or we could start with an image of 600 x 800 pixels and reduce it to 300 x 400 pixels for devices with half the scale. One way or another, we have a problem. If we expand a small image to fit a bigger screen, it loses quality, and if we reduce it, it occupies unnecessary space in memory because the image is

never shown in its original size. The second solution is the more efficient. It requires us to include in our project three versions of the same image, one for every scale. Considering the image of our example, we will need one picture of the husky in a size of 300 x 400 pixels for devices with a scale of 1x, another of 600 x 800 pixels for devices with a scale of 2x, and a third one of 900 x 1200 for devices with a scale of 3x. When we take this approach, the images are shown in the same size and with the same quality no matter the device, as illustrated by Figure 8-2.

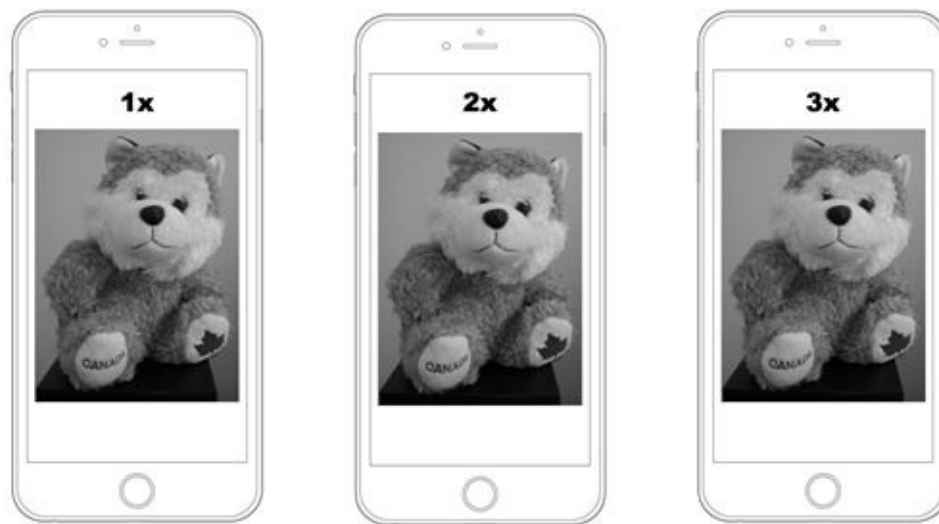


Figure 8-2: Different images for specific scales

Providing the same image in different resolutions solves the problem but introduces some complications. We have to create three versions of the same image and then select which one is going to be shown depending on the scale of the device. Selecting the right image to show each time requires us to follow a naming protocol. The iOS system was designed to detect the scale that corresponds to the image by reading a suffix on the file's name. What we have to do is to provide three files with the same name but with suffixes that determine the scale they correspond to. The file containing the image for the 1x scale (300 x 400 pixels in our example) only requires the name and the extension (e.g., **husky.png**), the name of the file with the image for the 2x scale (600 x 800) has to contain the suffix **@2x** (e.g., **husky@2x.png**), and the name of the file with the image for the 3x scale (900 x 1200) has to contain the suffix **@3x** (e.g., **husky@3x.png**). Every time the interface requires an image, the system reads the suffixes and loads the file corresponding to the scale of the screen.

The Basics: There is a very useful app available on the App Store for Mac computers called Prepo that can take an image of one size and expand it or reduce it to generate the images necessary for the rest of the scales. It can also help you generate the icons for your app. To edit the images you can use any image editor available on the market. There is a free online editor called Sumo Paint, available at

To incorporate the images to the project, we have to drag the files from Finder to the project's folder. Figure 8-3 shows how this action looks like.



Figure 8-3: Dragging files from Finder to our Xcode's project

When we drop the files into our Xcode's project, a window asks for information about the destination and the target. If we want the files to be copied to the project's folder (recommended), we have to activate the option *Copy items if needed*, as shown in Figure 8-4 below. We also have to indicate that the files are going to be added to our project by selecting the project's name in the *Add to targets* option (the project used in Figure 8-4 was called *Images*).



Figure 8-4: Options to add files to the project

Once we select these options and click the Finish button, Xcode adds the files to our project.

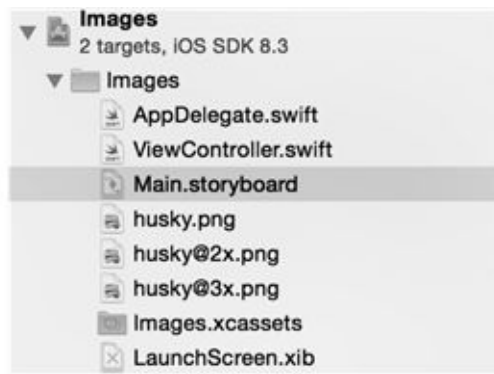


Figure 8-5: Files in our project

When we select the option to copy the files into our project's folder, we end up with two copies of the same files, one in the original folder and another in our project's folder. If we modify the original files and try to add them again to the project, instead of overwriting the files Xcode shows an error. To add the updated files to our project we first have to remove the old ones. To do this we have to select the files in the Navigator Area and press the Delete key on the keyboard. Xcode asks if we want to remove the reference or remove the files. Selecting the option *Move to Trash*, the files are completely removed from the project and we can add them again.

The Basics: All the project's files are stored in the same physical folder, but you can create groups on the Xcode's interface to organize them. New groups are created from the File menu. Select the option New / Group, write the name of the group, and then drag the files inside.

UIImage and UIImageView

Because the purpose of images is not only to be presented on the screen but also to be used in the construction of patterns and customized controls, the UIKit framework provides a basic class called **UIImage** to load images from files and then a separate class called **UIImageView** to present them on the screen. Objects created from the **UIImage** class load one image at a time and have its data ready for other objects to use. The class includes multiple initializers to create these objects from a variety of sources. The following are the most frequently used.

UIImage(named: name)—This initializer creates an object that contains the image from the file specified by the **name** attribute. The attribute is a string with the name of the file (the extension is not required).

UIImage(data: data, scale: scale)—This initializer creates an object that contains

an image generated from the data provided by the **data** attribute and with an associated scale specified by the **scale** attribute. The **data** attribute is an `NSData` object, and the **scale** attribute is a number of type `CGFloat` that represents the scale. We will introduce the `NSData` class in Chapter 21.

A `UIImage` object includes the following properties to get information about the image.

size—This property returns a `CGSize` value with the size of the image.

scale—This property returns a `CGFloat` value with the scale of the image.

imageOrientation—This property returns a value that identifies the orientation of the image. It is an enumeration of type `UIImageOrientation` with the values `Up`, `Down`, `Left`, `Right`, `UpMirrored`, `LeftMirrored`, `RightMirrored`.

Loading images from our code is very simple. The following example loads a file called `oranges.png` that contains a pattern and uses it to provide a background to our main view. The example assumes that we have created and incorporated to our project the files `oranges.png`, `oranges@2x.png`, and `oranges@3x.png` (the files are available in our website).

Listing 8-1: Adding a background pattern to the main view

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        if let mypattern = UIImage(named: "oranges") {
            view.backgroundColor = UIColor(patternImage: mypattern)
        }
    }
}
```

The `UIImage()` initializer implemented in Listing 8-1 looks for a file with the name “oranges” containing an image, and returns a `UIImage` object if the file is found and successfully loaded, or the value `nil` otherwise. In this example, we use the image to create a background pattern. The image is provided as the attribute of the `UIColor` initializer and the color is assigned to the `backgroundColor` property of the main view. When we initialize a `UIColor` object with an image, the object takes that image and prints it over and over again

to fill the area (see `UIColor` in Chapter 4). The result is illustrated in Figure 8-6.



Figure 8-6: Main view with a background pattern

© Pattern Copyright by Natalia de Frutos Ramos (nataliafrutos.wordpress.com)

Do It Yourself: Create a new Single View Application project. Download the three oranges.png files from our website (www.formasterminds.com) or provide your own. Drag the files to your project, as shown in Figure 8-3. Update the `ViewController` class with the code of Listing 8-1. Run the application. You should see something similar to Figure 8-6.

As shown in Listing 8-1, every time we need to load an image we only have to provide the name of the file. The system searches for files with that name and loads the one corresponding to the scale of the device's screen (if the right file can't be found, the system uses the file with lower resolution).

Some elements, such as buttons, use `UIImage` objects to customize their appearance. Once the images are added to the project, we can assign them to the element from the Attributes Inspector panel. The following example shows how to create a custom button using an image.



Figure 8-7: Custom button

To create this button we have to add a normal button to the main view and then modify its properties from the Attributes Inspector panel. As shown in Figure 8-7 below, the type has to be set as Custom (number 1), the title has to be erased (number 2), and the image has to be selected from the Image field (number 3). This example uses the images `buttonnormal.png`, `buttonnormal@2x.png`, and `buttonnormal@3x.png` available in our website.

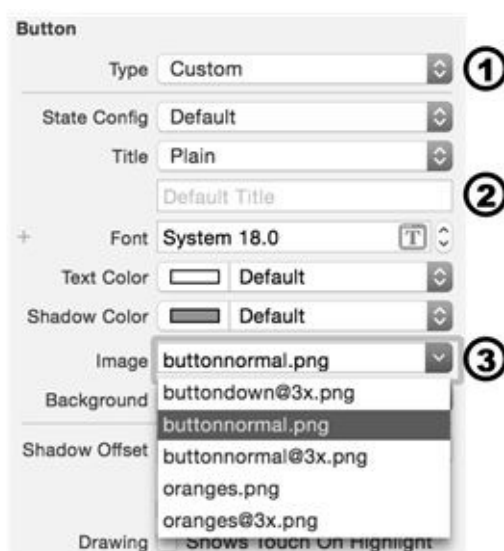


Figure 8-8: *Configuring the button from the Attributes Inspector panel*

The same way we did from code, we only have to select the file's original name; the files with suffixes are automatically loaded if necessary according to the scale of the device's screen.

Buttons can present more images depending on their state (see `UIButton` in Chapter 4). If, for example, we want the button to show a different image when it is pressed, we have to add it for the Highlighted state, as shown in Figure 8-9 below (number 1). Once this option is selected, the Image field is emptied to let us pick the new image for the state. This example uses the images `buttondown.png`, `buttondown@2x.png`, and `buttondown@3x.png` available on our website.



Figure 8-9: Adding an image for the Highlighted state

When the app is executed, the `UIImage` objects that represent the images we have added for the button on the Storyboard are recreated and the images are shown on the screen. The interface shows the button with the image defined for the Default state and changes it to the second image when the button is pressed, as illustrated by Figure 8-10.



Figure 8-10: The button shows a different image when pressed (right)

Do It Yourself: Add a button from the Object Library to the main view of the project created for the previous example. Add constraints to pin it to the bottom and the center of the view. Download the three `buttonnormal.png` files and the three `buttondown.png` files from our website (www.formasterminds.com). Drag the files to your project, as shown in Figure 8-3. Select the button in the Storyboard and modify its properties as shown in Figures 8-8 and 8-9. Run the application. When you press the button, the image should change, as shown in Figure 8-10.

The Basics: If you add the image before erasing the button's title, or accidentally

expand the view, the view's size won't match the size of the image. The best way to correct this problem is to select the button and then click on the option *Size to Fit Content* from the Editor menu at the top of the screen, as we did before for labels.

The `UIImage` objects may be added to views, buttons, or other elements to create customized controls, but if we just want to show the image on the screen, we have to create a `UIImageView` object. The `UIImageView` class includes the following initializers to create these objects.

`UIImageView(image: image)`—This initializer creates a `UIImageView` object with the image provided by the **`image`** attribute. The view's frame is defined at the position 0, 0 and with the size of the image. The attribute is a `UIImage` object.

`UIImageView(image: image1, highlighted: image1)`—This initializer creates a `UIImageView` object with two images, one for the normal state and another for the highlighted state. The view's frame is defined at the position 0, 0 and with the size of the image for the normal state. The **`image1`** and **`image2`** attributes are `UIImage` objects.

The `UIImageView` objects can also be incorporated to the Storyboard from the Object Library.

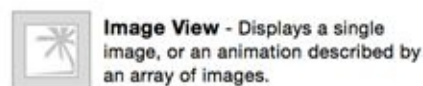


Figure 8-11: Image View option in the Object Library

The class includes properties to retrieve the images and configure the view.

`image`—This property sets or returns the image for the normal state. It is an optional of type `UIImage`.

`highlightedImage`—This property sets or returns the image for the highlighted state. It is an optional of type `UIImage`.

`userInteractionEnabled`—This property sets or returns a Boolean value that determines whether the Image View responds to user events or not. By default is set to `false`.

`highlighted`—This property sets or returns a Boolean value that determines whether the Image View is highlighted or not.

The class also includes properties and methods to create animations.

animationImages—This property sets or returns an array of `UIImage` objects to be used in the animation.

highlightedAnimationImages—This property sets or returns an array of `UIImage` objects to be used in the animation when the state of the Image View is highlighted.

animationDuration—This property sets or returns a value that determines the amount of seconds it takes for the animation to be over. It is of type `NSTimeInterval` (a typealias of `Double`).

animationRepeatCount—This property sets or returns an integer value that determines how many times the animation should be repeated. The value by default is 0, which means the animation will repeat indefinitely.

startAnimating()—This method starts the animation.

stopAnimating()—This method stops the animation.

isAnimating()—This method returns a Boolean value indicating whether the animation is running or not.

Image Views are views that contain images. They are shown on the Storyboard as squared areas that we can drag and then positioned with constraints, as any other view.



Figure 8-12: Adding an Image View to our interface

To assign an image to the view, we can modify its `image` and `highlightedImage` properties from code or do it from the Attributes Inspector panel, as we did for the button before. The Attributes Inspector panel shows the values of these two properties at the top. The image assigned to the `image` property is shown by default and the one assigned to the

`highlightedImage` property is shown when the view is highlighted (the value of its `highlighted` property is `true`).



Figure 8-13: Assigning images to the Image View

Image Views have their own size, independent of the size of the image they contain. When we load an image, we have to tell the system how it has to accommodate the image inside the view through the `contentMode` property that the view inherits from the `UIView` class (see `UIView` class in Chapter 4). This property can take several values, with the most useful being Scale to Fill, Aspect Fit, and Aspect Fill. Scale to Fill shrinks or stretches the image to fit the size of the view, Aspect Fit scales the image to fit the size of the view but keeping its aspect ratio, and Aspect Fill scales the image to always fill the view but also keeping its aspect ratio, which means some parts of the image may lay outside the boundaries of the view and, depending on the configuration, they may be hidden. Figure 8-14 shows what the interface looks like when we add to our `UIImageView` the image of the husky used in a previous example and set the view to different modes. The view is always the same size, but the image is adapted to occupy the space available according to the selected mode.

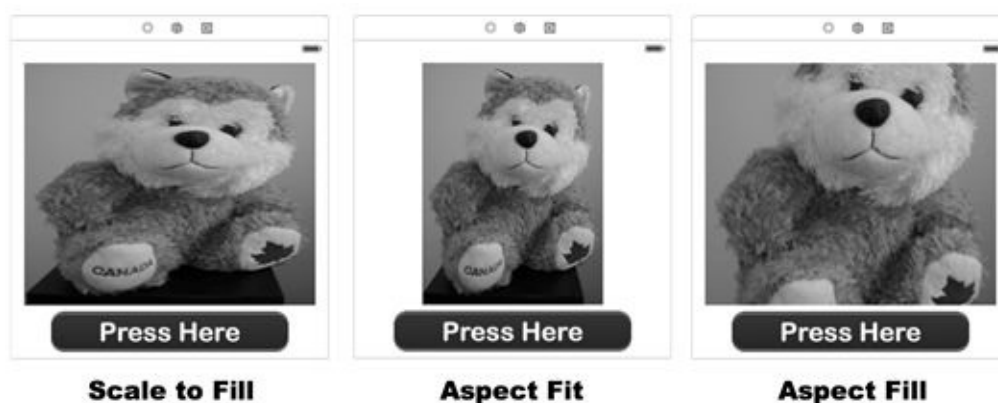


Figure 8-14: Different modes apply to the same Image View

Do It Yourself: Add a `UIImageView` from the Object Library to the main view of the project created for the previous examples. Add Top, Leading, and Trailing Space constraints to pin it to the top, left, and right side of the main view and also a Bottom constraint to pin it to the top of the button. Assign to the button a value of 750 to the vertical Content Hugging priority and a value 1000 to the vertical Content Compression Resistance priority to not let it expand or shrink (the Image View will always occupy the space available between the main view's top margin and the

button). Download the three husky.png files from our website (husky.png, husky@2x.png and husky@3x.png) and drag them to your project. Select the Image View and assign the husky.png image to it from the Attributes Inspector panel (Figure 8-13). Try different modes to see how the image is affected.

IMPORTANT: When we select the mode Aspect Fill, the parts of the image that are outside the boundaries of the view are shown unless we activate the option *Clip Subviews* from the Attribute Inspector panel. This is the option corresponding to the `UIView` property `clipsToBounds`, a Boolean property that indicates whether or not the content of the view should be clipped when it is outside its boundaries.

Scale to Fill is the mode by default, but it is usually not the one we need. We always have to set how the view is going to show the image, especially with Auto Layout that always modifies the view's size to the space available. What mode we should use depends on the purpose of the image. For instance, if we want to show the entire picture of the husky in its original aspect ratio, we have to set the Image View in the Aspect Fit mode. With this configuration, no matter what the space available and the size of the view are, we will always see the entire picture, as shown in Figure 8-15.



Figure 8-15: Image View in Aspect Fit mode

But this may not be the right mode if we want the image to be the background of our interface. In this case, the Image View has to occupy the entire screen and its image always has to fill the view without losing its aspect ratio, therefore the Aspect Fill mode will be more appropriate. Figure 8-16 shows how this mode adapts the image to fill the view.

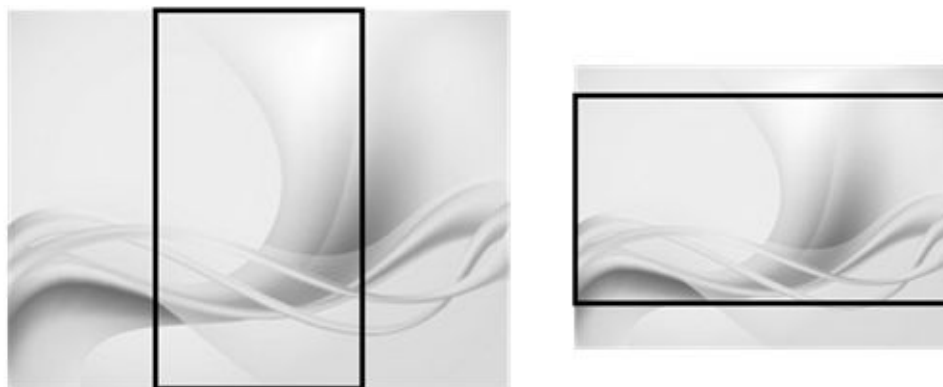


Figure 8-16: *The background image is resized to cover the entire screen*

Creating this background is simple. We have to add a new `UIImageView` on the back, pin it with Space constraints to each side of the main view, and set its mode to Aspect Fill.



Figure 8-17: *Image View in the background*

The Basics: New elements are added at the top of the hierarchy, which means they are going to be shown over the previous elements. To move an element to the back you can use the options from the Editor menu at the top of the screen. Select the element, click to open the Editor menu, click on Arrange, and select the right option until you get the element in the position you want. You can also modify the order of the elements from the Document Outline panel, as we have seen in Chapter 6 (Figures 6-3 and 6-17).

Because of its constraints, the new `UIImageView` added in Figure 8-16 always occupies the entire screen, and because of the Aspect Fill mode its image always fills the entire area, preserving its quality and aspect ratio. Figure 8-17 illustrates what our interface looks like in the simulator after we assign a background image to it (the files are called *background.png* and are available in our website).

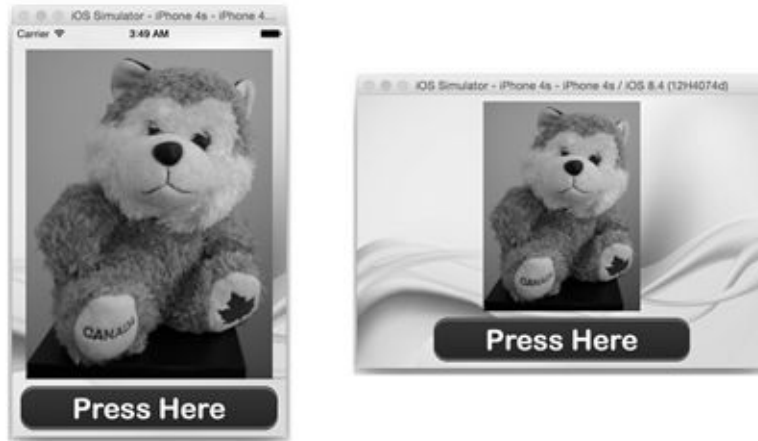


Figure 8-18: Background image in Aspect Fill mode

Do It Yourself: Add a new `UIImageView` to the main view of the previous example. Add Space constraints to pin the view to each side of the main view. Move the element to the back. Download the three background.png files from our website (background.png, background@2x.png, and background@3x.png) and drag them to your project. Assign the image to the Image View and set its mode to Aspect Fill. Run the application. You should see something similar to Figure 8-18.

Image Views, as any other element, may be connected to the view controller with Outlets and then modified from code. For example, we could show a different image when the button is pressed.

Listing 8-2: Replacing the picture when the button is pressed

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var pictureView: UIImageView!
    let door = UIImage(named: "door")
    @IBAction func changePicture(sender: UIButton) {
        pictureView.image = door
    }
}
```

Since we now have a background image, the new view controller of Listing 8-2 does not add the background pattern to the main view as we did before. For this example, we only need an Outlet for the Image View that shows the picture and an Action for the button. The Outlet was called `pictureView` and the Action `changePicture()`. We first create a new `UIImage` object with an image called `door.png` (the three files are available on our website) and then assign this object to the `image` property of the Image View inside the `changePicture()`

method to show the door.png image on the screen when the button is pressed.

If we want to switch between the husky and the door image, we have to create two `UIImage` objects and assign the corresponding object to the `image` property in every turn. In the following example we create these two objects and add a variable called `current` to know what image is currently being shown. The value 1 corresponds to the husky and the value 2 to the door.

Listing 8-3: Switching pictures

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var pictureView: UIImageView!
    var current = 1
    let husky = UIImage(named: "husky")
    let door = UIImage(named: "door")
    @IBAction func changePicture(sender: UIButton) {
        if current == 1 {
            pictureView.image = door
            current = 2
        } else {
            pictureView.image = husky
            current = 1
        }
    }
}
```

The `UIImage()` initializers return optional values, but the `image` property of the `UIImageView` object is an optional too, so we don't have to unwrap them. As a result of this code, when the button is pressed, the image shown on the screen changes.



Figure 8-19: Different images in the same Image View

In this example we are switching the images with a button, but they could be automatically replaced using the animation features provided by the `UIImageView` class. The following example creates an animation with the `husky.png` and `door.png` images. The animation starts when the button is pressed.

Listing 8-4: Animating images inside an Image View

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var pictureView: UIImageView!
    let husky = UIImage(named: "husky")
    let door = UIImage(named: "door")

    override func viewDidLoad() {
        super.viewDidLoad()
        pictureView.animationImages = [door!, husky!]
        pictureView.animationDuration = 4
    }
    @IBAction func changePicture(sender: UIButton) {
        pictureView.startAnimating()
    }
}
```

The animation has to be configured before it is executed. In Listing 8-4 we do this inside the `viewDidLoad()` method. The images we want to include in the animation are assigned to the `animationImages` property of the Image View and the duration of every cycle of the animation is set as 4 seconds. Notice that this time we had to unwrap the `UIImage` objects before to include them in the array because the `animationImages` property is not an optional type.

Once the button is pressed, the `startAnimating()` method of the Image View is executed and the images are automatically replaced over and over again. If we want the animation to stop, we can assign the total number of cycles to the `animationRepeatCount` property or called the `stopAnimating()` method at some point of the execution of the app, as shown next.

Listing 8-5: Controlling the animation

```
@IBAction func changePicture(sender: UIButton) {
    if pictureView.isAnimating() {
        pictureView.stopAnimating()
    }
}
```

```
} else {  
    pictureView.startAnimating()  
}  
}
```

The new `changePicture()` method of Listing 8-5 checks if the view is animating with the `isAnimating()` method and starts or stops the animation depending on the value returned.

Do It Yourself: The examples of Listings 8-2, 8-3, and 8-4 use the same interface than before, but incorporate the `door.png` image that you can download from our website. Copy the images to your project and update the `ViewController` class with the code you want to test. Run the application and press the button to see how they work. Replace the `changePicture()` method in Listing 8-4 with the one in Listing 8-5. Now the animation starts and stops every time the button is pressed.

The array of images assigned to the `animationImages` property doesn't replace the `UIImage` object assigned to the `image` property of the `UIImageView` object. This means that when the animation is over, the `UIImageView` object shows the image stored in the `image` property again. We can use this feature to improve our example adding an image to ask the user to press the button, for example. In the following example, instead of assigning `husky.png` as the Image View's initial image, we assign a new image called `sign.png` (the files are available on our website). This image is now shown as soon as the app is executed, but it is replaced when the animation begins.



Figure 8-20: Initial image for the Image View

Do It Yourself: Download the `sign.png`, `sign@2x.png`, and `sign@3x.png` images from our website and drag them to your project. Select the Image View and assign the `sign.png` image to it. Run the application and press the button. This image should be replaced by the animation and shown back when the animation is over.

Using images we can customize our interface and create a unique design. We can also make it more functional. For example, we could change the images for the button when the animation is running to say Stop instead of Press Here. The following example uses the images `buttonstopnormal.png` and `buttonstopdown.png` available in our website.

Listing 8-6: Modifying the button's images to reflect the state of the animation

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var pictureView: UIImageView!
    @IBOutlet weak var pressButton: UIButton!
    let husky = UIImage(named: "husky")
    let door = UIImage(named: "door")
    let buttonnormal = UIImage(named: "buttonnormal")
    let buttondown = UIImage(named: "buttondown")
    let buttonstopnormal = UIImage(named: "buttonstopnormal")
    let buttonstopdown = UIImage(named: "buttonstopdown")
    override func viewDidLoad() {
        super.viewDidLoad()
        pictureView.animationImages = [door!, husky!]
        pictureView.animationDuration = 4
    }
    @IBAction func changePicture(sender: UIButton) {
        if pictureView.isAnimating() {
            pressButton.setImage(buttonnormal, forState: .Normal)
            pressButton.setImage(buttondown, forState: .Highlighted)
            pictureView.stopAnimating()
        } else {
            pressButton.setImage(buttonstopnormal, forState: .Normal)
            pressButton.setImage(buttonstopdown, forState: .Highlighted)
            pictureView.startAnimating()
        }
    }
}
```

The code of Listing 8-6 adds more functionality but follows the same procedure than the previous example. To modify the images associated with every state of the button, we created an Outlet called `pressButton`. When the button is pressed, the `changePicture()` method is executed and the images are assigned to the button through its `setImage()` method (for more information about `setImage()` see `UIButton` in Chapter 4).



Figure 8-21: Button adapts to the situation

Do It Yourself: Download the `buttonstopnormal.png` and `buttonstopdown.png` images from our website and drag them to your project (your project has to contain all the images we have been using so far to work properly). Copy the code of Listing 8-6 into your view controller. Run the application. You should see the image of the button change accordingly to the state of the animation.

Visual Effects

Modern versions of iOS extensively use transparency in their interfaces, and Apple encourages developers to also adopt transparency in their own apps. The purpose is to produce a sense of depth and make the user feel like the content extends beyond the screen. One of the tricks used to achieve this effect is blurriness. When elements overlap, images and other elements behind are blurred, producing the feeling that they are far away. The UIKit framework includes the `UIVisualEffectView` class to help us add this effect to our images and our interface. The class includes an initializer with an attribute to define the effect.

`UIVisualEffectView(effect: effect)`—This initializer creates a `UIVisualEffectView` object with the effect set by the **`effect`** attribute. The attribute is a `UIVisualEffect` object.

The framework currently offers two subclasses of the `UIVisualEffect` class to define the effect, `UIBlurEffect` and `UIVibrancyEffect`. The following are the initializers to create these objects.

`UIBlurEffect(style: style)`—This initializer creates a `UIBlurEffect` object with the

style defined by the **style** attribute. The attribute is an enumeration of type `UIBlurEffectStyle` with the values `ExtraLight`, `Light`, and `Dark`.

UIVibrancyEffect(forBlurEffect: **effect)**—This initializer creates a `UIVibrancyEffect` object to add vibrancy to the blur effect determined by the **effect** attribute. The vibrancy effect is used to highlight an element inside a blur view.

The Object Library includes the Visual Effect View option to add an effect view with a blur effect to the Storyboard.



Figure 8-22: *Visual Effect View option in the Object Library*

The class creates a view that we have to expand to cover the area we want to blur. For example, we can add it over a background image to give the sense of depth to the entire interface. We add the Image View first, then the effect view, and finally the rest of the interface, as shown in Figure 8-23.

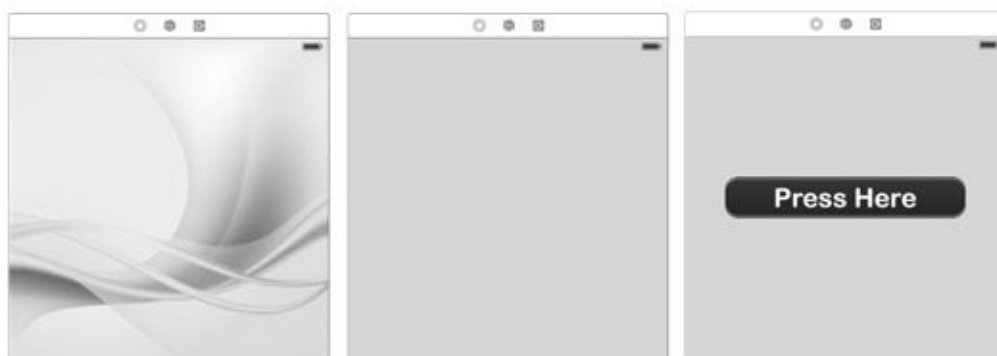


Figure 8-23: *Interface with a Visual Effect View*

The Attributes Inspector panel includes the option to select the style of the blur. The value defines different levels of opacity. Figure 8-24 shows our interface with a `Light` blur effect.



Figure 8-24: *Light blur effect*

Do It Yourself: Create a new Single View Application project. Add the background.jpg and buttonnormal.png images from the previous example to the project. Add an Image View to the main view, pin it to the edges of the main view with Space constraints, and assign the background.jpg image to it. Add a Visual Effect View over the Image View and also pin it to the edges to cover the whole image (Figure 8-23, center). Add a button with the image buttonnormal.png and set Alignment constraints to keep it at the center of the screen. Run the application. You should see something similar to Figure 8-24. Modify the Blur Style option in the Attributes Inspector panel to try different blur styles.

If we want the blur effect to be temporal, we can create the Visual Effect View from code. The following view controller contains an Action for the button included in the interface of Figure 8-23 that adds or removes a Visual Effect View when the button is pressed.

Listing 8-7: *Adding a Visual Effect View over the background image from code*

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var backgroundImage: UIImageView!
    var visualEffect: UIVisualEffectView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let blur = UIBlurEffect(style: UIBlurEffectStyle.Light)
        visualEffect = UIVisualEffectView(effect: blur)
        visualEffect.translatesAutoresizingMaskIntoConstraints = false
    }

    @IBAction func addEffect(sender: UIButton) {
        if visualEffect.superview == nil {
            view.insertSubview(visualEffect, aboveSubview: backgroundImage)
        }
    }
}
```

```

    let constraints1 = NSLayoutConstraint.constraintsWithVisualFormat("H:|-0-[effect]-0-|", options: [], metrics: nil,
views: ["effect": visualEffect])
    view.addConstraints(constraints1)

    let constraints2 = NSLayoutConstraint.constraintsWithVisualFormat("V:|-0-[effect]-0-|", options: [], metrics: nil,
views: ["effect": visualEffect])
    view.addConstraints(constraints2)
} else {
    visualEffect.removeFromSuperview()
}
}
}

```

The code defines a property called **visualEffect** to store a reference to the object and creates the object as soon as the main view is loaded. We have to create the **UIBlurEffect** object first with the style we want (**Light** in this example) and then add it to the initializer of the **UIVisualEffect** class to get the desired effect. Because we are adding the view from code, we also have to program the constraints for it, so we have to set its **translatesAutoresizingMaskIntoConstraints** property to **false** to not let the system define its own constraints (see Chapter 6, Listing 6-2). Because in this example we use the same button to add and remove the effect view from the interface, in the Action for the button we have to check if the view was already added or not. This may be achieved in different ways, but for this example we have decided to read the view's **superview** property. When the view is removed from the interface, this property returns **nil**. If the value is different from **nil**, we add the Visual Effect View to the main view. For this purpose we use the **insertSubview(view1, aboveSubview: view2)** method of the **UIView** class. This method inserts the first view (**view1**) over the second view (**view2**), and this is what we want this time because we need the Visual Effect View to be over the background image. After this, the constraints are defined to pin the Visual Effect View to the sides of the main view and cover the whole screen.

Do It Yourself: Create an Outlet called **backgroundImage** to connect the background image to the view controller. Create an Action for the button called **addEffect()**. Complete the **ViewController** class with the code of Listing 8-7. Run the application. You should see something similar to Figure 8-24.

Assets Catalog

Applications usually require dozens or even hundreds of images. No matter how many groups we create in the Navigator Area to organize these files, it will still be very difficult to identify them or remember what they are for just from looking at their names. The UIKit framework defines a class called **UIImageAsset** with the purpose to create a container

that offers a better alternative. The object created from this class is called Assets Catalog and it can work as a container for regular images, app icons, and launching images. The images are added to the Assets Catalog and then we have to ask for the one we want by mentioning its name. This is similar to what we have done so far, but the advantage is that Xcode provides a visual interface to manage the catalog, so we can see a quick preview of the image, load and organize the versions for each scale, and even create different sets of images for each Size Class.

The Assets Catalog for our application is automatically created and included in the project along with the rest of the files. To open the interface, we have to click the file called Images.xcassets in the Navigator Area. The Editor Area shows an interface similar to Figure 8-25.



Figure 8-25: Empty Assets Catalog

The interface includes two columns: the column on the left presents a list of sets of images and the column on the right displays the content of the selected set. A set is conformed by the three versions of an image, one for each scale, and also additional images that we want to present in different Size Classes. The list also includes a permanent set called AppIcon that contains the images for the icons necessary to publish our app in the App Store.

New sets can be added from the Editor menu at the top of the screen. If we open the Editor menu and click on the option *New Image Set*, a new empty set is created.



Figure 8-26: New set of images

The name of the set is the name we have to use to get the image from code. Xcode calls the new set *Image* but we can click on it and change it any time we want. Once the set is created, we can drag the file to the corresponding squares on the right column of the interface. For example, the file *husky.png* from our previous example goes to the 1x square, the file *husky@2x.png* goes to the 2x square, and the file *husky@3x.png* goes to the 3x square. Figure 8-27 shows the Editor Area after the images are dragged from Finder to the Assets Catalog and the name of the set is changed to “husky”.



Figure 8-27: Husky set ready

Once the images are loaded to the catalog, the set is ready to use. We don’t need to copy the files to the project’s folder anymore or anything like it. All we have to do is to load the image from our app using the name defined for the set (in this case, “husky”). But this process is still a little bit tedious. An easy way to create a new set is to drag the three images for the set to the Assets Catalog. Xcode creates a new set with the images and assigns their names as the name of the set. For instance, we can drag the files *door.png*, *door@2x.png*, and *door@3x.png* from our example and Xcode takes care of creating the set, assigning each image to the corresponding scale, and give it the name “door”.



Figure 8-28: New door set

The creation and configuration of the set is done automatically when we drag the files and drop them inside the Assets Catalog. In fact, we can drag several files together and Xcode takes care of extracting the information and creating all the sets. Figure 8-29 illustrates the Assets Catalog after we drag all the files necessary for the application created from previous examples.

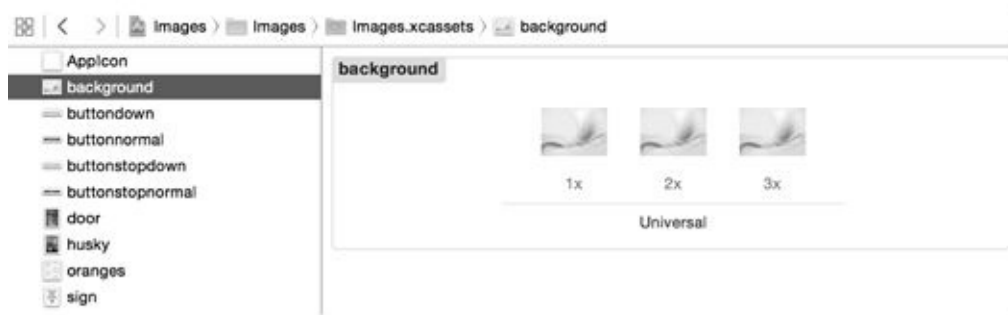


Figure 8-29: All the sets for our application

Do It Yourself: Select and delete the images added to the project's folder for the previous example (select the files from the Navigator Area, press the Delete key on your keyboard, and press the button Move to Trash in the popup window). Click on the Images.xcassets file on the Navigator Area to open the Assets Catalog interface. Drag all the images from Finder and drop them in the interface. Xcode adds the images to the catalog and creates the sets. You should see something similar to Figure 8-29. The interface in the Storyboard created in previous examples reads the images from the files, but these files don't exist anymore. To assign the images from our new Assets Catalog, open the Product menu at the top of the screen and select the option Clean to clean the project. Open the Storyboard, click on each Image View on the interface, and select again the image from the Attributes Inspector panel (no changes are necessary to the code).

Every set of images can include a version of the image for every scale, and also additional versions for different devices and Size Classes. When set is selected, the Attributes Inspector panel on the right shows a list of properties, including Devices and the Width and Height classes assigned to the set. By default, the set is shown in any device (Universal) and in Any width and Any height Size Classes, but we can change that modifying the values of these properties. For example, if we select the option Device Specific, we can select exactly the devices we want to associate to the images. The interface adds new squares where we can drag and drop the images for each device. Figure 8-30, next, shows what the Assets Catalog looks like after we configure the set to work in iPhones and iPads.



Figure 8-30: Set for specific devices

The same happens when we select different Size Classes. New squares are added at the bottom of the set to allow us to incorporate the corresponding files. The horizontal and vertical Size Classes for each image are identified inside the set by the characters * (Any), + (Regular) and – (Compact).



Figure 8-31: Set for specific Size Classes

Now every time we load the image from the application, the system selects the appropriate version according to the scale, the device, and the class.

IMPORTANT: The `UIImageAsset` class provides methods to add and remove images to the catalog from code. The `UIImage` class also includes an initializer to create an image for a specific trait collection and add it to the catalog. For more information, visit our website and follow the links for this chapter.

Icons

As we already mentioned, the Assets Catalog includes a set called *AppIcon* to manage the

icons we must provide for the application. The set includes placeholders for every icon we need and for every scale to cover all the devices and screens available. It even includes information about their size in points that we can use to design the images. The sizes required depend on the versions of the system and the type of devices our app is prepared for.

As any other image, the icons may be created from any image edition software available in the market. We have to create a file for every size required and then drag them to the corresponding square (sample files are available on our website). After the process is over, the AppIcon set should look like Figure 8-32.



Figure 8-32: AppIcon set with the icons for iPhones and iPads

IMPORTANT: The set includes all the icons that are required for the app, but you also have to provide icons for the iTunes store when you submit your app. We will study this process in Chapter 32.

Launching Image

The launching image is the image that is shown as soon as the app is executed. No matter how small the app is, it always takes a few seconds to load and be responsive the first time it is launched. These images are necessary to show the user that the app is responding to the action of tapping its icon and also to give the impression that the app is executed instantly. Xcode provides an additional file to create an image that adapts to every device using Auto Layout. This file is called `LaunchScreen.xib`, and is not an image but a main view to which we can add other elements to build an adaptive launching image. Figure 8-33 shows the view we see in the Editor Area when we click on this file.



Figure 8-33: *Launching Image by default*

By default, Xcode includes two labels showing the name of the app and the developer's information. This is just a placeholder. Apple's guidelines recommend creating a launching image that gives the impression to the user that the app is already running. For example, we can add a `UIImageView` pinned to the edges of the main view with an image that simulates the app's interface. Figure 8-34 shows a possible launching screen for the app created in our last example. We added an Image View to the main view with space constraints and the `background.png` image set in the Aspect Fill mode. Because of the constraints the image will adapt to the space available, and because we use the same image for our app's background, the transition between this image and the app's interface will be barely noticeable by the user.



Figure 8-34: *Launching Image for our app*

The Basics: Although we could add any element from the Object Library panel to the main view, only some of them are appropriate. Remember that the content of this view is shown before the app is launched, so there is no code we can process at that moment.

Chapter 9

Pickers

9.1 Listing Values

Something that users usually do in an application is to select values from a list. In desktop applications there are several tools available to let the user perform this task, but due to the reduced size of the screens, UIKit includes a new type of control for mobile applications. The control is called *Picker*, and what it does is to show a list of values in a graphic that simulates a spinning wheel. The values are presented in rows as if they were printed on the wheel, and the users can select the value they want by rotating the wheel. The framework includes a class to create general-purpose pickers called `UIPickerView` and a subclass of this class called `UIDatePicker` to create pickers for dates and times.

UIPickerView

The `UIPickerView` class provides properties and methods to create a general-purpose picker and manage its values. The values are organized in rows and components (columns). Each component represents a list of values and each row represents an individual value. Figure 9-1 illustrates a picker with two components, each with multiple rows (values).



Figure 9-1: *Picker View with two components*

To identify each row and component, the picker creates indexes starting from 0. The first value of a row will be at index 0, the second at index 1, and so on. Exactly the same as for arrays; which is actually not a coincidence. The items in a picker are organized this way so we can associate the picker with arrays in our code and easily present and retrieve values. Based on this organization, the class defines the following properties and methods to interact with the picker.

numberOfComponents—This property returns a value of type `Int` that represents the number of components in the picker.

numberOfRowsInComponent(component)—This method returns a value of type

Int that represents the numbers of rows in a component. The **component** attribute is a value of type **Int** that indicates the index of the component we want to access.

selectedRowInComponent(component)—This method returns a value of type **Int** that represents the index of the selected row in the component specified by the **component** attribute. The attribute is a value of type **Int** that indicates the index of the component we want to read.

reloadAllComponents()—This method forces the picker to update the values for its components.

reloadComponent(component)—This method forces the picker to update the values for a component. The **component** attribute is a value of type **Int** that indicates the index of the component we want to update.

selectRow(row, inComponent: component, animated: value)—This method selects the row with the index specified by the **row** attribute in the component specified by the **component** attribute. The **row** and **component** attributes are **Int** values that represent the respective indexes, and the **value** attribute is a Boolean value that determines whether the selection should be animate or not. The method rotates the wheel until the row selected is positioned in the center.

The **UIPickerView** class also defines two properties, **delegate** and **dataSource**, to assign delegate objects that will be in charge of configuring the picker and providing the values for the rows. The **delegate** property adopts the **UIPickerViewDelegate** protocol, which includes the following methods.

pickerView(pickerView, titleForRow: row, forComponent: component)—This method is called by the **UIPickerView** object on the delegate when it needs a value for the row indicated by the **row** attribute in the component indicated by the **component** attribute. The **row** and **component** attributes are values of type **Int** with the respective indexes. The method must return a string with the value for the row.

pickerView(pickerView, attributedTitleForRow: row, forComponent: component)—This method is called by the **UIPickerView** object on the delegate when it needs a value for the row indicated by the **row** attribute in the component indicated by the **component** attribute. The **row** and **component** attributes are values of type **Int** with the respective indexes. The value must be returned as an optional of type **NSAttributedString**.

pickerView(pickerView, didSelectRow: row, inComponent: component)—This method is called by the **UIPickerView** object on the delegate when the user selects a row (moves the wheel to get a new value in the center). The **row** and **component** attributes are values of type **Int** with the respective indexes.

The `dataSource` property, on the other hand, adopts the `UIPickerViewDataSource` protocol, which includes the following methods.

`numberOfComponentsInPickerView(pickerView)`—This method is called by the `UIPickerView` object on the data source delegate to know how many components to display. The method must return a value of type `Int` with the number of components we want.

`pickerView(pickerView, numberOfRowsInComponent: component)`—This method is called by the `UIPickerView` object on the data source delegate to know how many rows to display in a component. The **`component`** attribute is a value of type `Int` with the component's index (the method is called for every component in the picker).

The delegate object assigned to the `dataSource` property is a normal delegate, but because its purpose is to provide data it is usually called *data source* (hence the name of the property). As always, the delegates could be any objects we want, but it is common practice to declare the view controller in charge of the picker to be both the delegate and the data source.

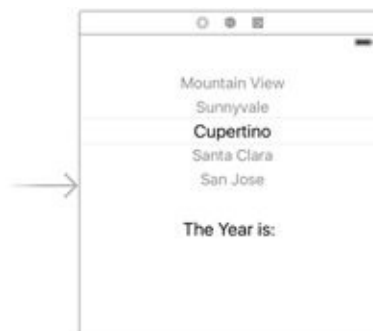


Figure 9-2: Interface with a Picker View

The interface of Figure 9-2 includes a picker and a label that we are going to use to show the selected year. A Picker View can be created from code, using the `UIView` initializer (`var picker = UIPickerView(frame: frame)`) or by dragging the element from the Object Library.



Figure 9-3: Picker View option in the Object Library

When the picker is added to the main view in the Storyboard from the Object Library,

Xcode shows it with some values by default, as illustrated by Figure 9-2, but the real values have to be provided by the code in our view controller.

Listing 9-1: Providing values for the picker

```
import UIKit

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {
    @IBOutlet weak var showYear: UILabel!
    @IBOutlet weak var pickerYears: UIPickerView!
    var years: [String]!

    override func viewDidLoad() {
        super.viewDidLoad()
        pickerYears.delegate = self
        pickerYears.dataSource = self
        years = ["1944", "1945", "1946", "1947", "1948", "1949", "1950"]
    }

    func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int {
        return 1
    }

    func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        return years.count
    }

    func pickerView(pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {
        return years[row]
    }

    func pickerView(pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) {
        let year = years[row]
        showYear.text = "The Year is: \(year)"
    }
}
```

We start the view controller by defining an array called `years` to store the values we want to show in the picker. Next, we create the Outlets for the label and the picker. The Outlet for the picker is only used in this example to declare the view controller as the picker's delegate and the data source. In the `viewDidLoad()` method we also initialize the `years` array with a series of values.

After the properties are initialized, we have to configure the picker and load the data from the delegate and data source methods. We first implement the `numberOfComponentsInPickerView()` method to tell the picker how many components we want. In this case, we return the value 1 because we only need one component to show the list of

years. The next method, `pickerView(picker, numberOfRowsInComponent: component)`, performs a similar function, but this time for the rows. Inside this method we count how many values are in the `years` array and return that number to indicate how many rows we need to show them all. Next is the `pickerView(picker, titleForRow: row, forComponent: component)` method, called by the picker to get the value for each row. The first call will be for the row at index 0, then for the row at index 1, and so on. Using the number of the row we get the corresponding value in the `years` array and return the string. Notice that this method can only return a `String` value. If we had an array of integers, for example, before returning each value we would have to convert it to a `String` using string interpolation or the `String()` initializer (e.g., `String(number)`). The last method, `pickerView(picker, didSelectRow: row, inComponent: component)`, is called by the picker when the user rotates the wheel and selects a new value. The index value of the selected row is reported by the method through its `row` attribute. Using this index value we can retrieve the value from the array and assign it to the label.

Do It Yourself: Create a new Single View Application project. Add a Picker View and a label to the main view, as shown in Figure 9-2. Define constraints to keep them at the top-center of the screen. Connect the label and the picker to Outlets called `showYear` and `pickerYears`, respectively. Complete your code with the code of Listing 9-1 and run the application. Every time you rotate the wheel, the number selected should be shown on the label.

The `pickerView(picker, didSelectRow: row, inComponent: component)` method let us get the value as soon as it is selected, but we can also read the current selected value any time we want using the `UIPickerView` method `selectedRowInComponent(component)`. For the following example we add a button to the interface and an Action to the view controller to change the value of the label every time the button is pressed.

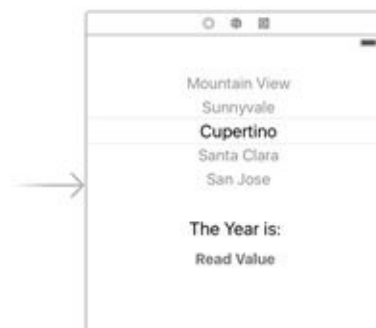


Figure 9-4: *New interface to get the value selected*

In this example, the method for the action replaces the `pickerView(picker, didSelectRow: row, inComponent: component)` method, but they could work together if necessary.

Listing 9-2: Getting the selected value

```
import UIKit

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {
    @IBOutlet weak var showYear: UILabel!
    @IBOutlet weak var pickerYears: UIPickerView!
    var years: [String]!

    override func viewDidLoad() {
        super.viewDidLoad()
        pickerYears.delegate = self
        pickerYears.dataSource = self
        years = ["1944", "1945", "1946", "1947", "1948", "1949", "1950"]
    }
    func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int {
        return 1
    }
    func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        return years.count
    }
    func pickerView(pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {
        return years[row]
    }
    @IBAction func getYear(sender: UIButton) {
        let row = pickerYears.selectedRowInComponent(0)
        showYear.text = "The Year is: \(years[row])"
    }
}
```

Inside the `getYear()` method we call the `selectedRowInComponent()` method to get the index of the selected row. The value 0 between parentheses corresponds to the index value of the component (we only have one component in our picker). The value of the `row` constant is used next to retrieve the value from the array and assign it to the label.

Besides the user rotating the wheel, there is also a method that allows us to select a value from code. This is particularly useful when we want to show a suggested value on the picker or just initiate the picker in a specific position. The following example initializes the picker in the year 1945.

Listing 9-3: Selecting the initial value

```
override func viewDidLoad() {
    super.viewDidLoad()
```

```

pickerYears.delegate = self
pickerYears.dataSource = self
years = ["1944", "1945", "1946", "1947", "1948", "1949", "1950"]

if let index = years.indexOf("1945") {
    pickerYears.selectRow(index, inComponent: 0, animated: false)
}
}

```

Since we want the value to be the initial selected value we have to call the method from the `viewDidLoad()` method. In Listing 9-3 we search for the value “1945” in the array with the `indexOf()` function and then select the row using the index value returned. The value for the **animated** attribute was set to **false** because we want the picker to rotate and show the right value as soon as it is displayed on the screen, but we can take advantage of this attribute to create a smooth transition when the selection has to be done after the picker was already shown.

Do It Yourself: Replace the `viewDidLoad()` method in the previous example by the one in Listing 9-3 and run the application. The initial selected value in the picker should be “1945”.

With a few modifications, we can easily create a picker with more than one component. The following example includes two components to let the user select a year and a city.

Listing 9-4: Creating a picker with multiple components

```

import UIKit

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {
    @IBOutlet weak var showYear: UILabel!
    @IBOutlet weak var pickerYears: UIPickerView!
    var cities: [String]!
    var years: [String]!

    override func viewDidLoad() {
        super.viewDidLoad()
        pickerYears.delegate = self
        pickerYears.dataSource = self
        cities = ["Mountain View", "Sunnyvale", "Cupertino", "Santa Clara"]
        years = ["1944", "1945", "1946", "1947", "1948", "1949", "1950"]
    }

    func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int {

```

```

        return 2
    }

    func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        if component == 0 {
            return cities.count
        } else {
            return years.count
        }
    }

    func pickerView(pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {
        if component == 0 {
            return cities[row]
        } else {
            return years[row]
        }
    }

    @IBAction func getYear(sender: UIButton) {
        let rowCity = pickerViewYears.selectedRowInComponent(0)
        let rowYear = pickerViewYears.selectedRowInComponent(1)
        showYear.text = "The Year is: \(years[rowYear]) in \(cities[rowCity])"
    }
}

```

The view controller of Listing 9-4 provides values for two components (columns) and decides which value return from the delegate methods according to the component in question. The first thing we need is to add a new array for the values of the second component. We called it **cities**. To tell the picker that we want two components, the value 2 was returned from the **numberOfComponentsInPickerView()** method. From this point on, we have to consider the component every time we interact with the picker. If the value of the **component** attribute is 0, we work with the **cities** array, and when the value is 1 (or different than 0) we work with the **years** array. The **getYear()** method was also expanded to read the selected value from each component and print them on the screen. The final application is shown in Figure 9-5 below.



Figure 9-5: *Picker with two components*

UIDatePicker

The `UIDatePicker` class is a subclass of `UIPickerView` designed to simplify the creation of pickers for dates and times. From this class we can create objects that show dates, dates and times, and even create a countdown timer. This picker can be created from code, using the `UIView` initializer (`var picker = UIDatePicker(frame: frame)`) or by dragging the element from the Object Library.

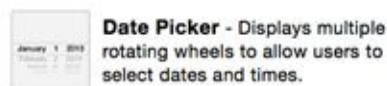


Figure 9-6: *Date Picker option in the Object Library*

The class provides the following properties to configure the picker.

datePickerMode—This property sets or returns the picker’s mode. It is an enumeration of type `UIDatePickerMode` with the values `Time`, `Date`, `DateAndTime`, and `CountDownTimer`.

maximumDate—This property sets or returns the maximum date the picker can show. It is a conditional of type `NSDate`.

minimumDate—This property sets or returns the minimum date the picker can show. It is a conditional of type `NSDate`.

date—This property sets or returns the selected date. It is of type `NSDate`.

minuteInterval—This property sets or returns an integer value that determines the

interval at which the minutes should be shown (1 by default).

countDownDuration—This property sets or returns a value of type `NSTimeInterval` (a typealias of `Double`) that indicates the seconds from which the timer counts down.

setDate(date, animated: value)—This method selects a new date. The **date** attribute is an `NSDate` value that specifies the date we want to select, and the **value** attribute is a Boolean value that determines whether we want the selection to be animated or not.

The class also includes three properties to determine the type of data to show. If the properties are not defined, they take the values by default set on the device.

calendar—This property sets or returns the calendar to use for the picker. It is an implicitly unwrapped optional of type `NSCalendar`.

locale—This property sets or returns the locale (regional information) to use for the picker. It is an optional of type `NSLocale`.

timeZone—This property sets or returns the time zone of the date shown by the picker. It is an optional of type `NSTimeZone`.

As any other element, the properties of a Date Picker added to the Storyboard may be configured from the Attributes Inspector panel. Depending on the selected mode, some values are considered and others ignored. For example, the Date and Date and Time modes take into consideration the minimum and maximum dates, but ignore the Interval value. To set the minimum and maximum dates we have to select the option first and then specify the value, as shown in Figure 9-7.

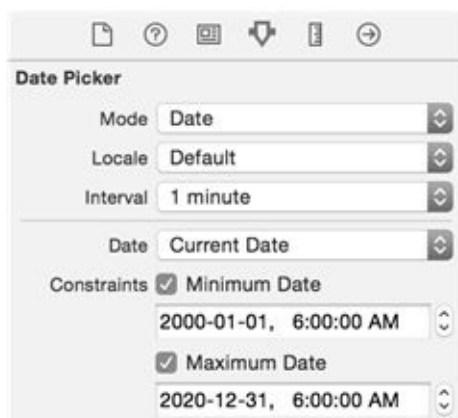


Figure 9-7: Date Picker properties

A Date Picker doesn't work with delegates. We interact with the picker directly through

its properties and methods. But there is no difference in how this picker presents the values and how the user selects them. Therefore, we can work with an interface similar to the one used for Picker Views. For our example, we need a Date Picker, a label, and a button, as illustrated in Figure 9-8.

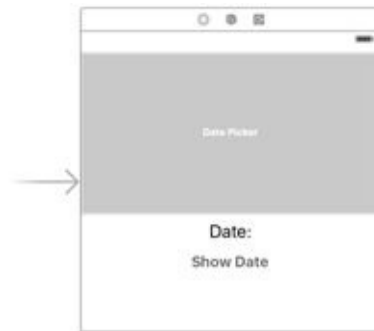


Figure 9-8: Interface to test Date Pickers

The purpose of this interface is to learn how to get and process the values selected in the picker. Because of this, the view controller only needs to include the Outlets for the picker and the label, and the Action for the button.

Listing 9-5: Printing the selected date

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var picker: UIDatePicker!
    @IBOutlet weak var showDate: UILabel!
    @IBAction func getDate(sender: UIButton) {
        let selectedDate = picker.date
        let format = NSDateFormatter()
        format.dateStyle = NSDateFormatterStyle.MediumStyle
        showDate.text = "Date: \(format.stringFromDate(selectedDate))"
    }
}
```

Getting the current selected date from the picker is as simple as reading its `date` property. This property is of type `NSDate`, which means that we have to create an `NSDateFormatter` object to format the value before presenting it on the screen (see `NSDate` in Chapter 4). The entire process is done inside the `getDate()` method, executed when the button is pressed. The `NSDateFormatter` object is created with the `MediumStyle` style, the string is generated with the `stringFromDate()` method, and finally the value is assigned to the label's `text` property.

As well as with `UIPickerView` objects, we can also declare an initial value for Date

Pickers. The value may be assigned directly to the `date` property or by calling the `setDate()` method. The only difference is that with the method we have the chance to animate the selection. The `viewDidLoad()` method below initializes the picker with the date 08-13-2010.

Listing 9-6: *Setting the initial selected date*

```
override func viewDidLoad() {
    super.viewDidLoad()

    let calendar = NSCalendar.currentCalendar()
    let components = NSDateComponents()
    components.year = 2010
    components.month = 8
    components.day = 13

    if let newDate = calendar.dateFromComponents(components) {
        picker.date = newDate
    }
}
```

The code of Listing 9-6 uses an `NSDateComponents` object and the `dateFromComponents()` method of the `NSCalendar` class to create a date. Because the `dateFromComponents()` method returns an optional we have to unwrap it before assigning it to the `date` property. Once we add the `viewDidLoad()` method to the previous example, the initial interface looks like Figure 9-9.



Figure 9-9: *Date Picker with an initial date*

Do It Yourself: Create a new Single View Application project. Add a Date Picker, a label, and a button to the main view, as shown in Figure 9-9. Define constraints to keep them at the top-center of the screen. Connect the label and the picker to Outlets called `showDate` and `picker`, respectively. Connect the button to an Action called `getDate()`

and complete the method with the code of Listing 9-5. You can also add the `viewDidLoad()` method of Listing 9-6 to define an initial value. Run the application, select a date, and press the button to assign it to the label.

IMPORTANT: The `UIDatePicker` class doesn't provide the functionality to create a countdown timer. When the picker is set to Count Down Timer mode, it shows hours and minutes to let the user select a starting time, but then is up to us to create the timer and display the time on the screen. We will study the class used to create timers in the next chapter.

Chapter 10

Timers

10.1 NSTimer

Timers are objects created from the `NSTimer` class that perform an action after a specific period of time. There are two types of timers: repeating and non-repeating. Repeating timers perform the action and then reschedule themselves to do it again in an infinite loop. Non-repeating timers, on the other hand, perform the action once and then invalidate themselves. The class includes the following properties and methods to create and manage timers.

valid—This property returns a value to indicate whether or not the timer is still running.

fireDate—This property sets or returns the date in which the timer is going to be fired. It is a value of type `NSDate`.

timeInterval—This property returns the time interval in seconds for repeating timers. It is a value of type `NSTimeInterval` (a type alias of `Double`).

userInfo—This property returns the object set by the timer to provide additional information.

tolerance—This property sets or returns a time period of tolerance in seconds to provide the system with more flexibility. It is a value of type `NSTimeInterval` (a type alias of `Double`). The value by default is 0.

scheduledTimerWithTimeInterval(time, target: object, selector: method, userInfo: info, repeats: repeat)—This type method returns repeating and non-repeating timers depending on the values of its attributes. The **time** attribute is an `NSTimeInterval` value (a type alias of `Double`) that represents the seconds the timer has to wait before performing the action, the **method** attribute is the name of the method that is going to be executed when the time runs out, the **info** attribute is a custom object with information about the timer, and the **repeat** attribute is a Boolean value to indicate the type of timer (repeating or non-repeating).

fire()—This method fires the timer without considering the time remaining.

invalidate()—This method invalidates the timer (stops the timer).

The `scheduledTimerWithTimeInterval()` method creates a timer according to the value of its attributes and automatically adds it to an internal loop that will process it when the time runs out. The following example shows how to create a repeating timer.

Listing 10-1: Creating a repeating timer

```
import UIKit
```

```
class ViewController: UIViewController {
    var counter: Int = 1
    var timer: NSTimer!
    override func viewDidLoad() {
        super.viewDidLoad()
        timer = NSTimer.scheduledTimerWithTimeInterval(1.0, target: self, selector: "report:", userInfo: nil, repeats: true)
    }
    func report(sender: NSTimer) {
        print("\(counter) times")
        if ++counter > 10 {
            timer.invalidate()
        }
    }
}
```

The code of Listing 10-1 creates the timer as soon as the view is loaded. This timer executes the `report()` method every second until it is invalidated. The `report()` method keeps track of the number of times it was called with the `counter` property. When the value of this property is greater than 10, it calls the `invalidate()` method on the timer to stop it.

The method designated as the action for the timer receives a reference to the timer object that we can use to access the timer's properties or invalidate it, but in this example we have decided to assign the timer to a property called `timer` in case later we want to stop it from another method.

In the last example we just printed a message on the console, but timers may be implemented along with other objects in more practical situations. The following example updates the date on the screen combining a timer with a Date Picker.

Listing 10-2: Updating time with a timer

```
import UIKit
```

```
class ViewController: UIViewController {
    @IBOutlet weak var dates: UIDatePicker!
    override func viewDidLoad() {
        super.viewDidLoad()
        NSTimer.scheduledTimerWithTimeInterval(60.0, target: self, selector: "update:", userInfo: nil, repeats: true)
    }
    func update(sender: NSTimer) {
```

```
    dates.date = NSDate()  
}  
}
```

Listing 10-2 assumes we have created a Date Picker and connected it to the view controller with an Outlet called `dates`. The timer for this example performs its action every 60 seconds (1 minute). The method executed by the timer creates a new `NSDate` object with the current date and assigns it to the picker's `date` property to update the value.

Do It Yourself: Create a new Single View Application project. Add a Date Picker to the main view. Select its mode to Date and Time. Connect the picker to the view controller with an Outlet called `dates`. Complete the view controller with the code of Listing 10-2. Run the application and wait a minute to see the timer updating the date on the screen.

Chapter 11

Scroll Views

11.1 Scroll Views

Scroll Views are views that display content larger than the area they cover. They incorporate functionality to let the user scroll the content or zoom in and out any part of it. The UIKit framework provides the `UIScrollView` class to create these types of views. A few classes inherit from `UIScrollView` to take advantage of its capabilities, like the `UITextView` class studied in Chapter 7 (see Figure 4-7), but Scroll Views are more generic and can manage any type of content we need. Figure 11-1 illustrates what the user sees when scrolling within a picture that is larger than the Scroll View's area.



Figure 11-1: User scrolling through the content area

The content area is like a canvas and the Scroll View like a window that lets us see one part of the canvas at a time. Therefore, there are two areas to define: the area of the Scroll View and the area of its content. The area occupied by the Scroll View is usually defined in the Storyboard with constraints, as any other view, but its content area has to be defined in coordinates from the view controller. The following are the most frequently used properties and methods included in the `UIScrollView` class to manage the view and its content.

contentSize—This property sets or returns the size of the content area. It is a structure of type `CGSize`.

contentOffset—This property sets or returns the **x** and **y** coordinates that determine the positions of the piece of image the Scroll View is currently showing on the screen. It is a structure of type `CGPoint`, with a value by default of 0, 0.

contentInset—This property sets or returns the content's padding. This is the space between the edges of the Scroll View and its content. It is a structure of type `UIEdgeInsets` with the properties **top**, **left**, **bottom**, and **right**, and a value by default of 0, 0, 0, 0.

scrollEnabled—This property is a Boolean value that determines if scrolling is enabled.

pagingEnabled—This property is a Boolean value that determines whether the Scroll View considers its content as pages or not.

showHorizontalScrollIndicator—This property is a Boolean value that determines whether the horizontal scroll indicator is visible or not.

showVerticalScrollIndicator—This property is a Boolean value that determines whether the vertical scroll indicator is visible or not.

zoomScale—This property sets or returns the current scale of the content. It is of type `CGFloat` with a value by default of 1.0.

maximumZoomScale—This property sets or returns the maximum possible scale for the content. It is of type `CGFloat` with a value by default of 1.0.

minimumZoomScale—This property sets or returns the minimum possible scale for the content. It is of type `CGFloat` with a value by default of 1.0.

zoomToRect(area, animated: animated)—This method zooms the content to make visible the area specified by the **area** attribute. This attribute is a value of type `CGRect` and the **animated** attribute is a Boolean value to indicate if the process should be animated.

scrollRectToVisible(area, animated: animated)—This method scrolls the content to make visible the area specified by the **area** attribute. This **area** attribute is a value of type `CGRect` and the **animated** attribute is a Boolean value to indicate if the process should be animated.

Scroll Views can also designate a delegate for configuration and to report changes on their state. The delegate must adopt the `UIScrollViewDelegate` protocol and implement its methods. The following are the most frequently used.

viewForZoomingInScrollView(scrollView)—This method is called by the Scroll View on its delegate to know which view on its content is going to be zoomed in or out.

scrollViewDidScroll(scrollView)—This method is called by the Scroll View on its delegate when the user scrolls the content.

scrollViewDidZoom(scrollView)—This method is called by the Scroll View on its delegate when the user zooms the content.

Scroll Views are no more than normal views with additional functionality. The Object Library offers an option to add it to the Storyboard.

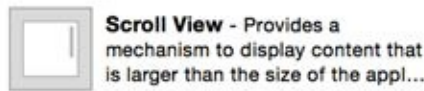


Figure 11-2: *Scroll View option in the Object Library*

A Scroll View may occupy the whole area of the main view or share the screen with other views, depending on what we want for our interface. Figure 11-3 shows what the main view in the Storyboard looks like after a Scroll View is added and pinned to the edges with Space constraints.

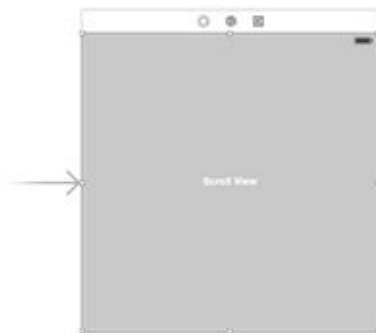


Figure 11-3: *Scroll View in the Storyboard*

Scrolling

Once a Scroll View is added, we have to connect it to the view controller and manage its content from code. The following example connects the Scroll View of Figure 11-3 to the **ViewController** class with an Outlet called **mainScroll**, defines the size of its content area, and adds an image to it.

Listing 11-1: *Configuring the Scroll View*

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var mainScroll: UIScrollView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let imgView = UIImageView(image: UIImage(named: "doll"))
        let imgWidth = imgView.frame.size.width
        let imgHeight = imgView.frame.size.height

        mainScroll.contentSize = CGSize(width: imgWidth, height: imgHeight)
    }
}
```

```

mainScroll.addSubview(imgView)
}
}

```

The content area of a Scroll View is defined with coordinates, starting from the top-left corner (coordinates 0, 0) and ending at the bottom-right corner (coordinates of which are determined by the size of the area). Every time a Scroll View is included in the interface we have to declare the size of its content area, otherwise the system won't know how far the user can go. The content is created from other views, such as Image Views or controls, and the views are added to the content area as subviews of the Scroll View. The position of the views in the content area is determined by the values of their **frame** properties.

The example of Listing 11-1 creates the Image View to add to the content area and then declares the size of the area as the size of the image (we didn't have to declare the frame of the Image View because the initializer takes the values from the image and automatically assigns them to the **frame** property). After the content area is ready, the Image View is added as a subview of the Scroll View with the **addSubview()** method (see Chapter 4, Listing 4-50). Because the size of the content area is the same as the size of the image it contains, the user can scroll the content and see the whole image. Figure 11-4 shows the image while it is being scrolled inside the Scroll View.



Figure 11-4: Scroll View showing different parts of an image

Do It Yourself: Create a new Single View Application project. Add a Scroll View to the initial view (Figure 11-3). Create an Outlet for the Scroll View called **mainScroll** and complete your view controller with the code of Listing 11-1. Download the image **doll.jpg** from our website and add it to the Assets Catalog. Run the application and scroll the image.

The content area of a Scroll View may contain as many views as we want. The following example includes a second image with a logo at the top-left corner of the area

(the image is called logo.png and is available on our website).

Listing 11-2: Adding more views to the content area

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var mainScroll: UIScrollView!

    override func viewDidLoad() {
        super.viewDidLoadDidLoad()
        let logoView = UIImageView(frame: CGRect(x: 25, y: 25, width: 249, height: 249))
        logoView.image = UIImage(named: "logo")
        let imgView = UIImageView(image: UIImage(named: "doll"))
        let imgWidth = imgView.frame.size.width
        let imgHeight = imgView.frame.size.height

        mainScroll.contentSize = CGSize(width: imgWidth, height: imgHeight)
        mainScroll.addSubview(imgView)
        mainScroll.addSubview(logoView)
    }
}
```

The example of Listing 11-2 creates the Image View for the logo with a frame in the position 25, 25, and then adds it to the Scroll View. Because the image with the logo is added after the image with the doll, it is shown at the top, but it is fixed at the position 25, 25 in the content area and therefore it scrolls along with the rest of the content, as shown in Figure 11-5.



Figure 11-5: Two images in the content area

Zooming

Setting the size of the content area gives enough information to the Scroll View to let the user scroll the content, but to zoom in and out we have to define the maximum and minimum scales allowed with the `minimumZoomScale` and `maximumZoomScale` properties, and also declare which view is going to participate in the process adopting the `UIScrollViewDelegate` protocol and implementing its `viewForZoomingInScrollView()` method.

Listing 11-3: Allowing zooming

```
import UIKit

class ViewController: UIViewController, UIScrollViewDelegate {
    @IBOutlet weak var mainScroll: UIScrollView!
    var imageView: UIImageView!
    override func viewDidLoad() {
        super.viewDidLoad()
        mainScroll.delegate = self
        imageView = UIImageView(image: UIImage(named: "doll"))
        let imageWidth = imageView.frame.size.width
        let imageHeight = imageView.frame.size.height
        mainScroll.minimumZoomScale = 1.0
        mainScroll.maximumZoomScale = 4.0
        mainScroll.contentSize = CGSize(width: imageWidth, height: imageHeight)
        mainScroll.addSubview(imageView)
    }
    func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView? {
        return imageView
    }
}
```

The view controller of Listing 11-3 adopts the `UIScrollViewDelegate` protocol and declares itself as the delegate of the `mainScroll` view. It also declares a minimum scale of 1.0 and a maximum scale of 4.0, and stores a reference to the Image View in the `imageView` property to be able to return it from the `viewForZoomingInScrollView()` method. When the Scroll View detects that the user is trying to zoom in or out, it calls this method on its delegate to know which view in the content area is going to be affected by the action and then begins to zoom until a limit is reached.

The zoom scale is considered to be 1.0 at the initial state. If we set absolute values for the `minimumScale` and `maximumScale` properties when the initial image is bigger than the area of the Scroll View, as in the previous example, the user won't be able to zoom out enough

to see the whole image on the screen. There are different ways to accommodate these values, depending on the effects we want to achieve in our application. For instance, the following example calculates the right minimum scale from the initial size of the image.

Listing 11-4: Calculating the minimum and maximum scales

```
import UIKit

class ViewController: UIViewController, UIScrollViewDelegate {
    @IBOutlet weak var mainScroll: UIScrollView!
    var imgView: UIImageView!
    override func viewDidLoad() {
        super.viewDidLoad()
        mainScroll.delegate = self
        view.setNeedsLayout()
        view.layoutIfNeeded()
        imgView = UIImageView(image: UIImage(named: "doll"))
        let imageWidth = imgView.frame.size.width
        let imageHeight = imgView.frame.size.height
        let scrollWidth = mainScroll.frame.size.width
        let scrollHeight = mainScroll.frame.size.height
        let minScale = max(scrollWidth / imageWidth, scrollHeight / imageHeight)
        let maxScale = max(minScale * 4.0, 1.0)
        mainScroll.minimumZoomScale = minScale
        mainScroll.maximumZoomScale = maxScale
        mainScroll.contentSize = CGSize(width: imageWidth, height: imageHeight)
        mainScroll.addSubview(imgView)
    }
    func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView? {
        return imgView
    }
}
```

To calculate what should be the minimum scale so the user can zoom out to see the whole image on the screen, we need to obtain the current values of the Scroll View's **width** and **height** properties. The problem is that the values of these properties are not updated when the interface is presented on the screen but during the system's updating cycles. To force the system to update the values at the time we need them, we have to call the main view's **setNeedsLayout()** and **layoutIfNeeded()** methods (see Chapter 6). If we don't execute these methods before reading the view's frame, the values returned might be those defined in the Storyboard.

The minimum scale is calculated in Listing 11-4 by dividing one of the Scroll View's

dimensions by the same dimension of the Image View. For example, if we divide the width of the Scroll View by the width of the Image View, we get the scale necessary to let the user zoom out until the whole width of the image is visible. But because we want the user to be able to see the whole image, we have to calculate the scale for both dimensions and get the greater value. We do this comparing the result of both formulas with the `max` function (`max(scrollWidth / imageWidth, scrollHeight / imageHeight)`). From the result we get the maximum scale by multiplying the value times 4.

The Basics: If you want the image to be presented from the beginning at the minimum scale, you can set the initial zoom scale with the `zoomScale` property. Insert the statement `mainScroll.zoomScale = minScale` at the end of the `viewDidLoad()` method of Listing 11-4.

Pages

The `UIScrollView` class includes functionality to split the content area into pages. The size of the pages is determined by the size of the Scroll View, so when the user swipes his finger on the screen, the current visible portion of the content area is completely replaced by the part that represents the next page. To activate this mode we have to assign the value `true` to the `pagingEnabled` property (or activate the option on the Attributes Inspector panel) and configure the content to represent the virtual pages. The following is a simple example that presents three images, `spot1.png`, `spot2.png`, and `spot3.png`, one per page.

Listing 11-5: Organizing the content area in pages

```
import UIKit
```

```
class ViewController: UIViewController {
    @IBOutlet weak var mainScroll: UIScrollView!

    override func viewDidLoad() {
        super.viewDidLoad()
        let images = ["spot1", "spot2", "spot3"]
        mainScroll.pagingEnabled = true
        view.setNeedsLayout()
        view.layoutIfNeeded()
        let scrollWidth = mainScroll.frame.size.width
        let scrollHeight = mainScroll.frame.size.height

        var posX: CGFloat = 0
        for img in images {
            let imageView = UIImageView(frame: CGRect(x: posX, y: 0, width: scrollWidth, height: scrollHeight))

```

```

imageView.image = UIImage(named: img)
imageView.contentMode = .ScaleAspectFit
imageView.clipsToBounds = true

mainScroll.addSubview(imageView)
mainScroll.contentSize = CGSize(width: scrollWidth * CGFloat(images.count), height: scrollHeight)
posX = posX + scrollWidth
}
}
}

```

When paging is activated, the user doesn't scroll an image anymore but instead it moves from one page to another. Therefore, the Image Views have to be created of the size of the Scroll View and positioned side by side to represent the pages. This is why in Listing 11-5 we declare the frame of every Image View. The values for the frame's **width** and **height** properties are determined by the size of the Scroll View (**scrollWidth** and **scrollHeight**), and the coordinates for the position are calculated according to the page the views represent. The first Image View is added at the coordinate 0, 0, but the second Image View is displaced to the right a distance determined by the width of the Scroll View. To calculate this value we use the **posX** variable. In every cycle of the loop, the variable is incremented with the value of the **scrollWidth** property to establish the horizontal position of the next view. As a result, the images are positioned on the content area one after another, like the frames of a film.

In this example, we also set the **contentMode** of each Image View to **ScaleAspectFit** to show the whole image on the screen and the **clipsToBounds** property to **true** to make sure that no matter the content mode the image will never be drawn outside its boundaries.

Do It Yourself: Create a new Single View Application project. Add a Scroll View to the initial view (Figure 11-3). Create an Outlet for the Scroll View called **mainScroll**. Complete your view controller with the code of Listing 11-5. Download the images **spot1.png**, **spot2.png**, and **spot3.png** from our website and add them to the Assets Catalog. Run the application. When you scroll the content the interface should transition from right to left or left to right, jumping from one image to another.

The UIKit framework includes a special type of control called Page Control that is particularly useful in these kinds of applications. The control presents dots on the screen that represent each page available and change their colors according to the page selected or visible. There is an option available in the Object Library to add a Page Control to the Storyboard.

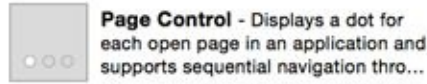


Figure 11-6: *Page Control option in the Object Library*

The control is created from the `UIPageControl` class. The class provides the following properties for configuration.

currentPage—This property sets or returns the number of the current page. It is a value of type `Int`.

numberOfPages—This property sets or returns the number of pages represented by the control (the number of dots shown on the screen). It is a value of type `Int`.

hidesForSinglePage—This property is a Boolean value that determines whether or not the control should be hidden when there is only one page.

pageIndicatorTintColor—This property sets or returns the color of the dots when the pages they represent are not visible. It is an optional of type `UIColor`.

currentPageIndicatorTintColor—This property sets or returns the color of the dot that represents the visible page. It is an optional of type `UIColor`.

To introduce the Page Control in the interface we could reduce the size of the Scroll View to make some room or put the control over the Scroll View. The last option looks more modern and may include a translucent bar to improve the appearance.

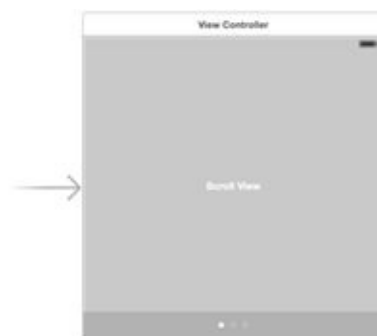


Figure 11-7: *Page Control added to the main view*

The interface presented in Figure 11-7 includes the translucent bar with the Page Control inside. The bar is created from a simple `UIView` object with a gray background and an Alpha value of 0.7 (to make it translucent), and positioned at the top of the Scroll View. This position is achieved by turning the view into a subview of the main view. When we add a view over the Scroll View, the new view is included as a subview of the Scroll View

and therefore it becomes part of its content area. To take it out of the Scroll View, we have to open the Document Outline panel and drag the item representing the bar to a position below the Scroll View and inside the main view (see Document Outline panel, Figure 6-57). Once the view is in the right place in the hierarchy, we can define the constraints to pin it to the bottom of the main view from the Document Outline panel, as explained in Chapter 6.

The Page Control is configured by default for three pages, but we can change the initial values from the Attributes Inspector panel. This is usually not necessary since the content to display with this type of control is loaded dynamically and its properties are configured from the view controller, as shown in the following example.

Listing 11-6: Calculating the page

```
import UIKit

class ViewController: UIViewController, UIScrollViewDelegate {
    @IBOutlet weak var mainScroll: UIScrollView!
    @IBOutlet weak var pageCounter: UIPageControl!
    var page: Int = 0
    override func viewDidLoad() {
        super.viewDidLoad()
        let images = ["spot1", "spot2", "spot3"]
        pageCounter.numberOfPages = images.count
        pageCounter.pageIndicatorTintColor = UIColor.blackColor()
        pageCounter.currentPageIndicatorTintColor = UIColor.whiteColor()
        mainScroll.delegate = self
        mainScroll.pagingEnabled = true
        view.setNeedsLayout()
        view.layoutIfNeeded()
        let scrollWidth = mainScroll.frame.size.width
        let scrollHeight = mainScroll.frame.size.height
        var posX: CGFloat = 0
        for img in images {
            let imgView = UIImageView(frame: CGRect(x: posX, y: 0, width: scrollWidth, height: scrollHeight))
            imgView.image = UIImage(named: img)
            imgView.contentMode = .ScaleAspectFit
            imgView.clipsToBounds = true
            mainScroll.addSubview(imgView)
            mainScroll.contentSize = CGSize(width: scrollWidth * CGFloat(images.count), height: scrollHeight)
            posX = posX + scrollWidth
        }
    }
}
```

```

func scrollViewDidScroll(scrollView: UIScrollView) {
    let pageWidth = mainScroll.frame.size.width
    let getPage = round(mainScroll.contentOffset.x / pageWidth)
    let currentPage = Int(getPage)
    page = currentPage
    pageCounter.currentPage = page
}
}

```

To represent the pages on the screen with the Page Control we not only have to configure the control but also get the number of the current visible page. The `UIScrollView` class doesn't offer any property or method to help us get this value; we have to calculate it ourselves. Fortunately, the delegate offers a method called `scrollViewDidScroll()` that is executed every time the user scrolls the content. Also, the system automatically updates the `contentOffset` property with the current displacement of the content area every time a scroll takes place. Implementing these tools we can calculate the current page. The value is obtained dividing the value of the `x` property of the `contentOffset` structure by the size of the page (the width of the Scroll View). Because the result is a floating-point number, we round it, turn it into an integer, and assign it to the `currentPage` property of the Page Control to show the current page on the screen.

Do It Yourself: Add a view over the Scroll View of the project created for the previous example. Select the view, go to the Attributes Inspector panel, and give it a gray background and an alpha value of 0.7. Open the Document Outline panel and drag this view outside the Scroll View and inside the main view (see Figure 6-57). Control-drag a line from the new view to the main view in the Document Outline panel to create the Space constraints to pin it to the bottom (Figure 11-7). Add a Page Control to the gray view and connect it to an Outlet called `pageCounter`. Complete your view controller with the code of Listing 11-6 and run the application. The Page Control should indicate the current visible page.

Letting the user zoom in or out the entire content area when the Scroll View is configured to work with pages would defeat the purpose of pages, but it is possible to let the user zoom each page individually. Scroll Views may be created inside other Scroll Views, and we can use this feature to build complex interfaces where a view may have part of the content with vertical scroll and other parts with horizontal scroll. We can also use this feature to allow zooming inside each page of a Scroll View. All we have to do is embed the content of each page (the Image Views in our example) inside Scroll Views, set their maximum and minimum scales, and implement the `viewForZoomingInScrollView()` method of the `UIScrollViewDelegate` protocol to tell the Scroll Views that zooming is allowed, as we did in the example of Listing 11-4.

Listing 11-7: Zooming the pages in and out

```
import UIKit

class ViewController: UIViewController, UIScrollViewDelegate {
    @IBOutlet weak var mainScroll: UIScrollView!
    @IBOutlet weak var pageCounter: UIPageControl!
    var imageViews: [UIImageView] = []
    var page: Int = 0
    override func viewDidLoad() {
        super.viewDidLoad()

        let images = ["spot1", "spot2", "spot3"]
        pageCounter.numberOfPages = images.count
        pageCounter.pageIndicatorTintColor = UIColor.blackColor()
        pageCounter.currentPageIndicatorTintColor = UIColor.whiteColor()
        mainScroll.delegate = self
        mainScroll.pagingEnabled = true
        view.setNeedsLayout()
        view.layoutIfNeeded()

        let scrollWidth = mainScroll.frame.size.width
        let scrollHeight = mainScroll.frame.size.height

        var posX: CGFloat = 0
        for img in images {
            let childScroll = UIScrollView(frame: CGRect(x: posX, y: 0, width: scrollWidth, height: scrollHeight))
            childScroll.contentSize = CGSize(width: scrollWidth, height: scrollHeight)
            childScroll.minimumZoomScale = 1.0
            childScroll.maximumZoomScale = 4.0
            childScroll.delegate = self
            let imgView = UIImageView(frame: CGRect(x: 0, y: 0, width: scrollWidth, height: scrollHeight))
            imgView.image = UIImage(named: img)
            imgView.contentMode = .ScaleAspectFit
            imgView.clipsToBounds = true
            imageViews.append(imgView)
            childScroll.addSubview(imgView)
            mainScroll.addSubview(childScroll)
            mainScroll.contentSize = CGSize(width: scrollWidth * CGFloat(imageViews.count), height: scrollHeight)
            posX = posX + scrollWidth
        }
    }

    func scrollViewDidScroll(scrollView: UIScrollView) {
```

```

let pageWidth = mainScroll.frame.size.width
let getPage = round(mainScroll.contentOffset.x / pageWidth)
let currentPage = Int(getPage)
if currentPage != page {
    let scroll = imageViews[page].superview as! UIScrollView
    scroll.setZoomScale(1.0, animated: true)
    page = Int(currentPage)
    pageCounter.currentPage = page
}
}
func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView? {
    return imageViews[page]
}
}

```

On the view controller of Listing 11-7, Scroll Views are created to embed every Image View and then are added as subviews of the `mainScroll` view. Notice that the minimum and maximum scales are set only for the nested Scroll Views, not to the `mainScroll` view, because we only want the Scroll Views assigned to every page to be able to zoom.

There is also an aesthetic change inside the `scrollViewDidScroll()` method. This method is executed every time the user scrolls the pages, but that scroll does not always produce the current page to be replaced. We check this condition comparing the current page with the previous one and set the zoom scale back to 1.0 if the new page is different. This returns the images to the initial state every time the user moves to another page.

The previous examples create applications that let the user transition from one image to another with the move of a finger. But there is a problem. Because the size of the pages is determined from the size of the Scroll View, the application doesn't work anymore if the device is rotated or the size of the Scroll View changes for some reason. To solve this problem we have to implement the methods defined for Size Classes to detect and report changes in size (Chapter 6, Listing 6-9). The following example implements the `viewWillTransitionToSize()` method to update the frames of the Image Views and their Scroll Views when the device is rotated.

Listing 11-8: Adapting the content area to a new orientation

```

import UIKit

class ViewController: UIViewController, UIScrollViewDelegate {
    @IBOutlet weak var mainScroll: UIScrollView!
    @IBOutlet weak var pageCounter: UIPageControl!
    var imageViews: [UIImageView] = []
    var page: Int = 0

```

var rotating = false

```
override func viewDidLoad() {
    super.viewDidLoad()
    let images = ["spot1", "spot2", "spot3"]
    pageCounter.numberOfPages = images.count
    pageCounter.pageIndicatorTintColor = UIColor.blackColor()
    pageCounter.currentPageIndicatorTintColor = UIColor.whiteColor()
    mainScroll.delegate = self
    mainScroll.pagingEnabled = true
    view.setNeedsLayout()
    view.layoutIfNeeded()
    for img in images {
        let childScroll = UIScrollView(frame: CGRectZero)
        childScroll.minimumZoomScale = 1.0
        childScroll.maximumZoomScale = 4.0
        childScroll.delegate = self
        let imgView = UIImageView(frame: CGRectZero)
        imgView.image = UIImage(named: img)
        imgView.contentMode = .ScaleAspectFit
        imgView.clipsToBounds = true
        imageViews.append(imgView)
        childScroll.addSubview(imgView)
        mainScroll.addSubview(childScroll)
    }
    updateSize()
}

func updateSize() {
    let scrollWidth = mainScroll.frame.size.width
    let scrollHeight = mainScroll.frame.size.height

    var posX: CGFloat = 0
    for imgView in imageViews {
        let scroll = imgView.superview as! UIScrollView
        scroll.frame = CGRect(x: posX, y: 0, width: scrollWidth, height: scrollHeight)
        scroll.contentSize = CGSize(width: scrollWidth, height: scrollHeight)
        imgView.frame = CGRect(x: 0, y: 0, width: scrollWidth, height: scrollHeight)
        posX = posX + scrollWidth
    }
    mainScroll.contentSize = CGSize(width: scrollWidth * CGFloat(imageViews.count), height: scrollHeight)
    let scrollView = imageViews[page].superview as! UIScrollView
    mainScroll.scrollToVisible(scrollView.frame, animated: true)
}
```



```

func scrollViewDidScroll(scrollView: UIScrollView) {
    if !rotating {
        let pageWidth = mainScroll.frame.size.width
        let getPage = round(mainScroll.contentOffset.x / pageWidth)
        let currentPage = Int(getPage)
        if currentPage != page {
            let scroll = imageViews[page].superview as! UIScrollView
            scroll.setZoomScale(1.0, animated: true)
            page = Int(currentPage)
            pageCounter.currentPage = page
        }
    }
}

func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView? {
    return imageViews[page]
}

override func viewWillTransitionToSize(size: CGSize, withTransitionCoordinator coordinator:
UIViewControllerTransitionCoordinator) {
    rotating = true
    coordinator.animateAlongsideTransition(nil, completion: {(context:
UIViewControllerTransitionCoordinatorContext!) in
        let scroll = self.imageViews[self.page].superview as! UIScrollView
        scroll.setZoomScale(1.0, animated: true)
        self.updateSize()
        self.rotating = false
    })
}
}

```

An important change we have to make in this new example is to move the definition of the frames for the Scroll Views and their Image Views to a separate method. This is because every time the size of the `mainScroll` view changes, the code has to update the values of the frames. We still define the pages inside the `viewDidLoad()` method, but we call our `updateSize()` method at the end to define their frames. This method is called again when the user rotates the device. Every time the system detects a rotation or changes on the interface, it calls the `viewWillTransitionToSize()` method. In this method we call the `animateAlongsideTransition()` method of the transition coordinator to update the values of the frames as soon as the transition is over (the statements were declared inside the closure for the completion handler). There are two more important things we do inside this method: we restore the zoom of the current page back to its initial scale and modify the value of a Boolean property called `rotating` to inform the rest of the code that the interface is rotating. This is because we don't want to calculate the number of the current page inside the

`scrollViewDidScroll()` method until all the values for the frames are updated. The `updateSize()` method also includes a call to the `scrollRectToVisible()` method of the `mainScroll` view at the end to scroll the content to the right position after a rotation (when the device is rotated, the values of the `contentOffset` property are not updated by the system).

Do It Yourself: The last two examples use the same interface created before. Copy the code you want to try inside your view controller and run the application.

Scrolling the Interface

Scroll Views let us display any content that doesn't fit entirely on the screen, including the interface itself. There are times when our interface contains so many elements and controls that there is not enough space to show them all, some get hidden when the device is rotated, or are overlapped by the keyboard. Scroll Views provide the functionality we need to display large interfaces inside a single main view, but they present a challenge. The content area of a Scroll View lays its views using a coordinate system and absolute values, while the interface is organized with constraints. If we add an element to the Scroll View, this element is not incorporated as a normal subview but as part of its content area and therefore it is positioned and sized according to the values of its `frame` property. Fortunately, there is a way to connect the content of a Scroll View with the rest of the views on the interface and adapt their size to the space available. The trick is to create the right constraints to set the size of the content area and report it to the Scroll View. There are different ways to do it, but the most simple is to add a single view to the Scroll View and then use it as the container for the rest of the interface. Figure 11-8 illustrates the process. The Scroll View is added to the main view first (left), then a single view is added to the Scroll View (center), and finally the view is extended to occupy the entire Scroll View area.

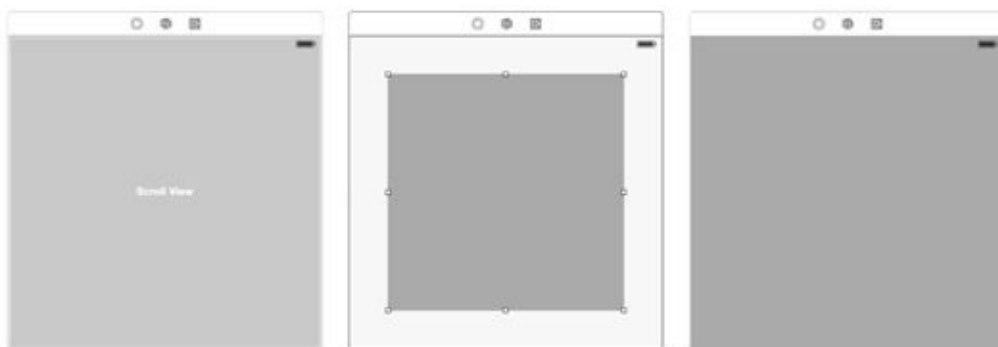


Figure 11-8: *Scroll View with a container view inside*

Both views have to be pinned to the edges of their containers with Space constraints.

The Scroll View is pinned to the edges of the main view and the single view is pinned to the edges of the Scroll View. Figure 11-9 illustrates the constraints required.

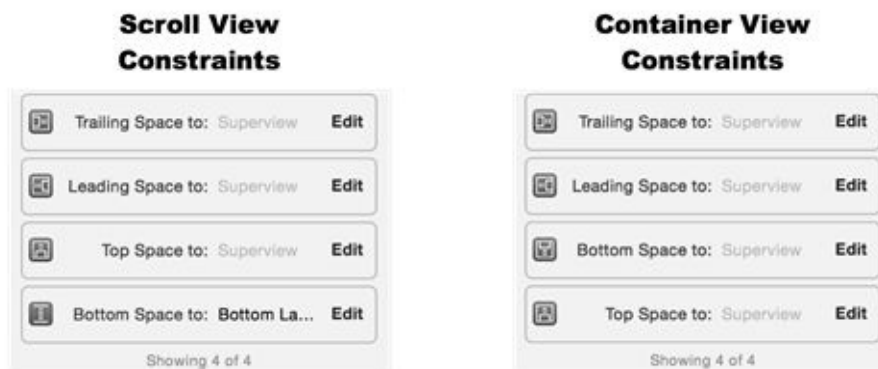


Figure 11-9: Initial constraints for the views

The constraints for both views are the same, but their purpose is different. The Space constraints for the Scroll View define its size on the interface, but the Space constraints for the container view communicate the size of the view to the Scroll View so that the Scroll View always knows the current size of its content area.

Do It Yourself: Create a new Single View Application project. Add a Scroll View to the main view. Select the Scroll View, open Pin menu and create left, right, top, and bottom constraints to pin it to each side of the main view. Add an empty view inside the Scroll View, expand it to occupy the Scroll View's area (Figure 11-8, right), and create left, right, top, and bottom constraints to pin it to the sides of the Scroll View. You will see that the constraints are shown in color red, because the system doesn't have enough information to establish the size of the content area.

Now that the initial conditions were established, we have to create the constraints necessary to determine the size of the content view inside the Scroll View. The vertical size is determined by the elements inside the container view and their constraints, but the horizontal size has to be determined according to the space available. This requires the creation of two more constraints that pin the container view to the left and right sides of the main view. Figure 11-10 illustrates the process. We have to control-drag a line from the container view to the main view on the Document Outline panel and create a Leading and a Trailing constraint with a value of 0.

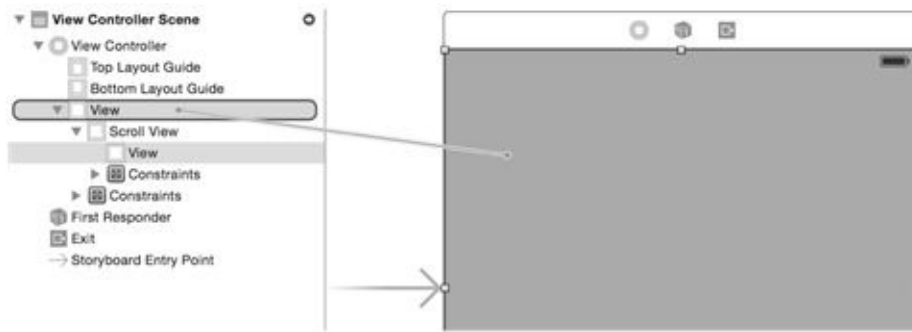


Figure 11-10: Additional constraints for the container view

Every time the device is rotated or the space available changes, these constraints change the width of the content view. This new size is communicated to the Scroll View by the Space constraints added before. Figure 11-11 shows the final set of constraints necessary for the container view.

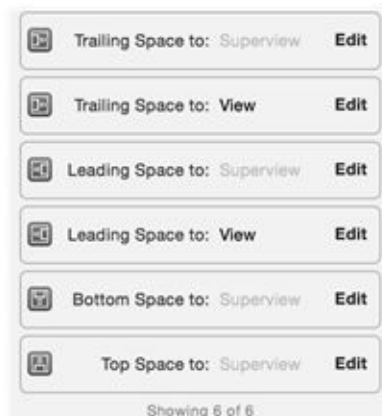


Figure 11-11: Constraints required by the container view

Do It Yourself: Control-drag a line from the container view to the item that represents the main view in the Document Outline panel (Figure 11-10). Create a Leading and a Trailing constraint. When you add the Space constraints to the main view using the Document Outline, the system creates the constraints to the margin of the view, not to the edges. You have to select each constraint, deactivate the option *Relative to margin* (Figure 6-19), and set the constant value as 0. After this, the constraints of your container view should look like the ones in Figure 11-11.

So far, we have determined the width of the content area. To determine the height we have to add the elements of the interface inside the container view and set constraints that link them to all sides of the view. In the following example we have added a Text Field, an Image View, a Text View, and a button. Figure 11-12 shows this interface, including all the constraints. Notice that we have added constraints from the elements to every side of the container view, including the bottom.



Figure 11-12: Interface inside the container view

These interfaces are usually bigger than the standard sizes of the main views defined by the Storyboard. Xcode offers the alternative to set a specific size for the main view, so we can design the long interfaces in the Storyboard. The tool is called *Simulated Size* and it is available in the Size Inspector panel when the Scene is selected. It includes two options, Fixed and Freeform. When we select Freeform, the panel lets us change the values of the main view's width and height, as shown in Figure 11-13 (the values for the interface of Figure 11-12 were set to 400 and 600, respectively).

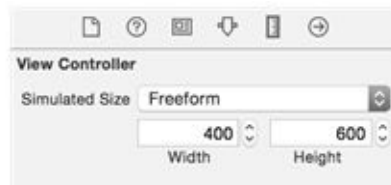


Figure 11-13: Simulated Size tool

Once all the elements of our interface are added to the container view and all the constraints for these elements are defined, the Scroll View is ready to determine the size of its content area and let us scroll to see the parts that are not visible. Figure 11-14 shows the application in the simulator.



Figure 11-14: *Scrolling the entire interface*

Do It Yourself: Select the Scene in the Storyboard. Open the Size Inspector panel and select the Freeform option from the Simulated Size tool. Set the size of the main view as 400 x 600. Expand the container view to fit this new size. Add the elements shown in Figure 11-12 to the container view, including the Text View at the bottom with a gray background. Set the proper constraints to pin the elements to each other and to the sides of the view (Figure 11-12). Run the application. You should be able to scroll the interface vertically, as shown in Figure 11-14.

Scroll Views are also useful when we need to move the interface to help the user achieve a purpose or keep an input field visible when the keyboard is activated. The following example scrolls the interface of Figure 11-12 when the Text View is selected and the keyboard is opened.

Listing 11-9: *Scrolling the Text View to the top*

```
import UIKit

class ViewController: UIViewController, UITextViewDelegate {
    @IBOutlet weak var mainScroll: UIScrollView!
    @IBOutlet weak var descriptionView: UITextView!

    override func viewDidLoad() {
        super.viewDidLoad()
        descriptionView.delegate = self
    }

    func textViewDidBeginEditing(textView: UITextView) {
        let app = UIApplication.sharedApplication()
        let barHeight = app.statusBarFrame.size.height
        let height = descriptionView.frame.origin.y - barHeight
        mainScroll.setContentOffset(CGPoint(x: 0, y: height), animated: true)
    }
}
```

```
}  
}
```

To perform the task when the Text View is selected, the view controller has to adopt the `UITextViewDelegate` protocol and implement the `textViewDidBeginEditing()` method (see Chapter 7). Inside this method we calculate how much we have to scroll the interface to keep the Text View visible when the keyboard is opened. The value is obtained from the distance between the top of the main view and the vertical position of the Text View (`descriptionView.frame.origin.y`). We subtract to this value the height of the status bar to avoid the two elements to overlap, and then use the result to set the new value for the `contentOffset` property. In consequence, when the user taps on the Text View, the `textViewDidBeginEditing()` method is executed and the content of the Scroll View is displaced to show the Text View at the top of the screen.



Figure 11-15: *Scrolling the interface when the keyboard is opened*

Setting the value of the `contentOffset` property forces the Scroll View to scroll the content no matter its size, but if the user scrolls the view while the keyboard is open, the Scroll View adjusts the interface to the size of the content area and the Text View is hidden again behind the keyboard. The solution is to add some padding at the bottom of the Scroll View to make it think that the content area is bigger than it is. The `UIScrollView` class provides the `contentInset` property for this purpose. This property takes a structure of type `UIEdgeInsets` that contains four properties to set the padding for every side of the Scroll View: **top**, **left**, **bottom**, and **right**.

Listing 11-10: *Adding padding at the bottom of the Scroll View*

```
import UIKit  
  
class ViewController: UIViewController, UITextViewDelegate {  
    @IBOutlet weak var mainScroll: UIScrollView!  
    @IBOutlet weak var descriptionView: UITextView!
```

```

override func viewDidLoad() {
    super.viewDidLoad()
    descriptionView.delegate = self
}

func textViewDidBeginEditing(textView: UITextView) {
    let app = UIApplication.sharedApplication()
    let barHeight = app.statusBarFrame.size.height
    let height = descriptionView.frame.origin.y - barHeight
    mainScroll.setContentOffset(CGPoint(x: 0, y: height), animated: true)

    let padding = view.frame.size.height - descriptionView.frame.size.height
    mainScroll.contentInset = UIEdgeInsets(top: 0, left: 0, bottom: padding, right: 0)
}
}

```

The total space we need to let the user scroll the content until the Text View is visible is calculated subtracting the height of the Text View to the height of the main view. This value is assigned to the **bottom** property of a new **UIEdgeInsets** structure and the structure is assigned to the **contentInset** property to set the new padding.

Do It Yourself: Create an Outlet called **descriptionView** for the Text View in the interface of Figure 11-12. Complete the view controller with the code of Listing 11-9 or 11-10 to test the examples. Run the application and tap on the Text View to activate the keyboard. The interface should automatically scroll (Figure 11-15).

IMPORTANT: This is a simple approach to deal with the keyboard; it doesn't contemplate how to revert the situation and it uses approximate values, but it gives you an idea of the things you can do with a Scroll View. In Chapter 18 we will study how to work with keyboard notifications and how to get the right values to position the elements on the screen.

Chapter 12

Gesture Recognizers

12.1 Gesture Recognizers

Gestures are actions performed by the user on the screen, such as tapping, swiping, or pinching. These gestures are difficult to detect because all that the screen returns is the position of the fingers, not information about how the fingers were moved. This is why Apple introduced gesture recognizers. A gesture recognizer is an object that performs all the necessary calculations to detect a gesture. The UIKit framework defines a generic class called **UIGestureRecognizer** to provide the basic functionality for gesture recognizers and also includes multiple subclasses with the additional properties and methods to create recognizers for specific gestures. The following is the list of subclasses available.

UITapGestureRecognizer—This class creates a gesture recognizer that looks for single or multiple taps. The class includes the properties **numberOfTapsRequired** and **numberOfTouchesRequired** to determine the number of taps that have to occur and the number of fingers that have to tap for the gesture to be recognized. Both properties are of type **Int**.

UIPinchGestureRecognizer—This class creates a gesture recognizer that looks for a pinching gesture (zoom out and zoom in). The class includes the properties **scale** and **velocity** to set a scaling factor and return the velocity of the pinch. The properties are of type **CGFloat**.

UIRotationGestureRecognizer—This class creates a gesture recognizer that looks for a rotation gesture. The class includes the properties **rotation** and **velocity** to set the rotation of the gesture in radians and return the velocity in radians per second. They are of type **CGFloat**.

UISwipeGestureRecognizer—This class creates a gesture recognizer that looks for a swiping gesture. The class includes the properties **direction** and **numberOfTouchesRequired** to set the direction allowed and the number of fingers that must touch the screen for the gesture to be recognized. The **direction** property is a structure (or a set of structures) of type **UISwipeGestureRecognizerDirection** with the properties **Right**, **Left**, **Up**, and **Down**.

UIPanGestureRecognizer—This class creates a gesture recognizer that looks for a panning gesture. The class includes the method **translationInView(view)** to return the current position of the gesture relative to the starting position, the method **velocityInView(view)** to return the velocity of the gesture expressed in points per second, and the properties **maximumNumberOfTouches** and **minimumNumberOfTouches** to set the maximum and minimum number of fingers that must be touching the view for the gesture to be recognized.

UIScreenEdgePanGestureRecognizer—This class creates a gesture recognizer that looks for a panning gesture that starts near the edge of the screen. The class includes the **edge** property to determine the edges that recognize the gesture. The property is a structure or a set of structures of type **UIRectEdge** with the properties **None**, **Top**, **Left**, **Bottom**, **Right**, and **All**.

UILongPressGestureRecognizer—This class creates a gesture recognizer that looks for a long-press gesture (the user presses one or more fingers on the screen for a certain period of time). The class includes the properties **minimumPressDuration** (a **Double** value to indicate the minimum time the fingers must press the screen), **numberOfTouchesRequired** (the number of fingers that must touch the screen), **numberOfTapsRequired** (the number of taps), and **allowableMovement** (a **CGFloat** value that indicates the maximum movement allowed).

The **UIGestureRecognizer** class provides an initializer to create objects from every subclass and also general properties and methods. The following are the most frequently used.

UIGestureRecognizer(target: **target, action: **action**)**—This initializer creates a gesture recognizer configured with the target and the action specified by its attributes. The **target** attribute is the object where the action will be executed, and the **action** attribute is the method to be executed when the gesture is detected.

state—This property returns the current state of the gesture recognizer. It is an enumeration of type **UIGestureRecognizerState** with the values **Possible**, **Began**, **Changed**, **Ended**, **Cancelled**, **Failed**, and **Recognized**.

view—This property returns a reference to the view the gesture recognizer is attached to.

enabled—This property sets or returns a Boolean value that indicates whether or not the gesture recognizer is enabled.

locationInView(view)—This method returns a **CGPoint** value to indicate the coordinates where the gesture occurred in the view specified by the **view** attribute.

locationOfTouch(touch, inView: **view)**—This method returns a **CGPoint** value to indicate the coordinates where a touch occurred in the view specified by the **view** attribute. The **touch** attribute is an integer with a value corresponding to the index of the touch we want to check (touches are stored in an array).

numberOfTouches()—This method returns the number of touches involved in the gesture.

Gesture recognizers may be added to an element from code or on the Storyboard from the Object Library. Figure 12-1 shows the options available.

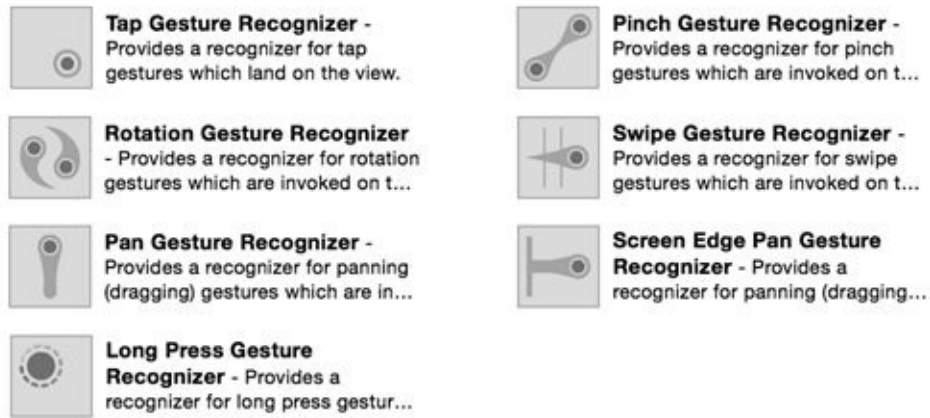


Figure 12-1: *Gesture Recognizer options in the Object Library*

Although gesture recognizers may be added to any of the views provided by the UIKit framework, not all of them are ready to recognize gestures. The `UIView` class offers two Boolean properties we can modify called `userInteractionEnabled` and `multipleTouchEnabled` to activate detection of gestures and multiple touch. If we want to use gestures in a view that deactivates these options by default, like an Image View, for example, we have to change their values from code or from the Attributes Inspector panel, as shown in Figure 12-2.



Figure 12-2: *Gesture options*

The gesture recognizers are added to the Storyboard as any other view. All we have to do is to drag the gesture recognizer from the Object Library to the view in the Storyboard and then connect it to an Action in the view controller. After the gesture recognizer is dropped over the view, Xcode adds an icon to the top of the Scene to represent it. Figure 12-3 shows what the Scene looks like after we add a Pan Gesture to a view on the interface.



Figure 12-3: *Gesture Recognizer's icon*

As an example, we are going to fade an image in and out responding to a pan gesture. The interface includes an Image View with the picture of the husky (husky.png) and a Pan Gesture.



Figure 12-4: Image View with a Pan Gesture

The view controller for this example needs an Outlet for the Image View and an Action for the gesture. To create the Action we may control-drag a line from the gesture's icon (Figure 12-3) or from the item that represents the gesture on the Document Outline panel.

Listing 12-1: Responding to a gesture

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var picture: UIImageView!
    var previous: CGFloat = 0
    @IBAction func fadingOut(sender: UIPanGestureRecognizer) {
        let translation = sender.translationInView(picture)
        let delta = translation.x - previous
        let width = picture.frame.size.width
        let alpha = picture.alpha + (delta / width)
        if alpha > 0.1 && alpha < 1 {
            picture.alpha = alpha
        }
        if sender.state == .Ended {
            previous = 0
        } else {
            previous = translation.x
        }
    }
}
```

Every type of gesture returns different kinds of information. In the case of the pan gesture, the object includes a few methods that calculate and return the current position and velocity of the gesture. Every time a gesture begins or a change occurs on its state, the object calls the Action method with a reference of itself (**sender: UIPanGestureRecognizer**). From this method we can read and process the information provided by the gesture object

and perform custom tasks. In Listing 12-1, we call the `translationInView()` method to get the value of the current position and calculate the difference between the old horizontal position and the new one. From this difference and the width of the image we can get a proportional value that added to the alpha value lets us associate the movement of the finger with changes in the transparency of the picture on the screen.

Because the limits of the gesture's position may vary, we check that the alpha value obtained from those numbers doesn't go over 1.0 or below 0.1 (at 0.0 the Image View doesn't recognize the gesture anymore), and then assign it to the `alpha` property of the Image View to reflect the changes. To keep track of the old position and be able to compare it with the new one, we store it in the `previous` property. Every time the action is called to report changes in the gesture, the new value is stored on this property. But because the position of a pan gesture is considered to be 0 at the point when the gesture begins, we can't keep using the values of a previous gesture and therefore the `previous` property must return to 0 when the gesture is over. This is detected at the end reading the value of the gesture's `state` property.

Do It Yourself: Create a new Single View Application project. Download the `husky.png` image from our website and add it to the Assets Catalog. Add an Image View to the main view and assign the image of the husky to it. In the same panel, activate the option called *User Interaction Enabled* (Figure 12-2). Drag the option Pan Gesture from the Object Library and drop it over the Image View. You should see the icon pictured in Figure 12-3. Create an Outlet for the image called `picture`. Control-drag a line from the gesture's icon to the view controller to create an Action called `fadingOut()`. Complete the view controller with the code of Listing 12-1. Run the application. The picture should fade in and out when you move your finger to the left or right.

The Basics: Gesture recognizers can't be added to different views. If we want to use the same gesture recognizer for different views, we have to create new objects. But several gesture recognizers of the same type may be connected to the same Action.

Gesture recognizers may be added to any view from code. The object is created first with the `UIGestureRecognizer` initializer and then configured and added to the view. The `UIView` class offers the following methods to manage gesture recognizers.

addGestureRecognizer(gesture)—This method adds a gesture recognizer to the view. The `gesture` attribute is an object of any of the subclasses of the `UIGestureRecognizer` class.

removeGestureRecognizer(gesture)—This method removes a gesture recognizer from the view. The `gesture` attribute is a reference to the gesture added before with the

addGestureRecognizer() method.

The following example adds a tap gesture to a Scroll View to zoom in and out of specific points on the image. The example assumes that we have added a Scroll View to the main view, as we did in the previous chapter for the interface of Figure 11-3.

Listing 12-2: Adding gestures from code

```
import UIKit

class ViewController: UIViewController, UIScrollViewDelegate {
    @IBOutlet weak var scroll: UIScrollView!
    var image: UIImageView!
    var zooming = false
    override func viewDidLoad() {
        super.viewDidLoad()
        scroll.delegate = self
        scroll.minimumZoomScale = 1.0
        scroll.maximumZoomScale = 4.0
        image = UIImageView(image: UIImage(named: "doll"))
        scroll.contentSize = CGSize(width: image.frame.size.width, height: image.frame.size.height)
        scroll.addSubview(image)
        let gesture = UITapGestureRecognizer(target: self, action: "zoomPicture:")
        scroll.addGestureRecognizer(gesture)
    }
    func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView? {
        return image
    }
    func zoomPicture(sender: UITapGestureRecognizer) {
        if !zooming {
            let position = sender.locationInView(scroll)
            scroll.zoomToRect(CGRect(x: position.x, y: position.y, width: 1, height: 1), animated: true)
            zooming = true
        } else {
            scroll.setZoomScale(1.0, animated: true)
            zooming = false
        }
    }
}
```

The **UIGestureRecognizer** initializer requires two values, a reference to the object where the

action will be performed and the method that is going to be executed when the gesture is detected. Because we are going to perform all the tasks on the view controller, we declared the target as `self` and created a method called `zoomPicture()` to perform the action.

This example follows the procedure studied in Chapter 11 to configure the Scroll View and allow the user to scroll and zoom the picture. When the Scroll View is ready, the code creates a `UITapGestureRecognizer` object and adds it to the view with the `addGestureRecognizer()` method. Because of this, every time the user taps on the Scroll View, the `zoomPicture()` method is executed. In this method we zoom in or out, depending on the value of the `zooming` property used as control. To zoom in, the code gets the position of the tap in the view's coordinates from the `locationInView()` method and then zooms in to a rectangle in that position. To zoom out, we just set the scale back to 1.0.

Do It Yourself: Create a new Single View Application project. Add a Scroll View to the main view. Connect the Scroll View with an Outlet called `scroll`. Download the `doll.jpg` image from our website and add it to the Assets Catalog. Complete the view controller with the code of Listing 12-2. Run the application and tap on the image. The image should zoom in where you tapped.

The Basics: New devices incorporate a 3D touch feature that measures the pressure applied to the display. This allows developers to provide new functionality to their applications. Apple offers multiple classes to recognize and respond to 3D gestures. The topic is beyond the scope of this book. For more information visit our website and follow the links for this chapter.

Chapter 13

Navigation

13.1 Multiple View Controllers

Apps that only require one main view and a simple interface are very hard to find. Because of the limited amount of space, developing apps for mobile devices demands the creation of multiple main views to represent virtual screens that replace one another responding to user interaction (see Figure 5-11). Practical applications contain multiple main views connected with each other following a predetermined path that users can take to navigate and get the information they want.

Xcode templates, like the Single View Application template we have been using so far to create our projects, provide initial main views to start from. The rest of the views are added later to the Storyboard as needed from options included in the Object Library. The option to add single main views that are controlled by `UIViewController` objects is called *View Controller*.

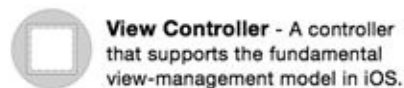


Figure 13-1: View Controller option in the Object Library

Dragging this option to the Storyboard creates a new main view. After the views are added, they can be dragged from the top bar to different positions to visually represent what users will see when they navigate through the interface. Figure 13-2 shows what the Storyboard looks like after we add a new view and move it to the right side of the initial view.



Figure 13-2: Second view added to the Storyboard

We can add all the views necessary to build our interface. Every view represents a new screen and they will be shown on the device's screen one at a time.

Do It Yourself: Create a new Single View Application project. Drag the View Controller option from the Object Library to the Storyboard (Figure 13-1). Arrange the main views to look like Figure 13-2.

The option to add a new main view to the Storyboard is called View Controller because it creates a main view that is controlled by an object of a subclass of the `UIViewController` class, but this doesn't mean that the option creates the view controller for the view. Every time we add a new Scene to the Storyboard we have to create the file for the subclass ourselves. Files are created from the File menu at the top of the screen. If we click on the File menu and go over the New option, a submenu shows the option File... Once we click on this option, a window is opened to select the type of file we want to create (we can also use a shortcut pressing the Command and N keys).

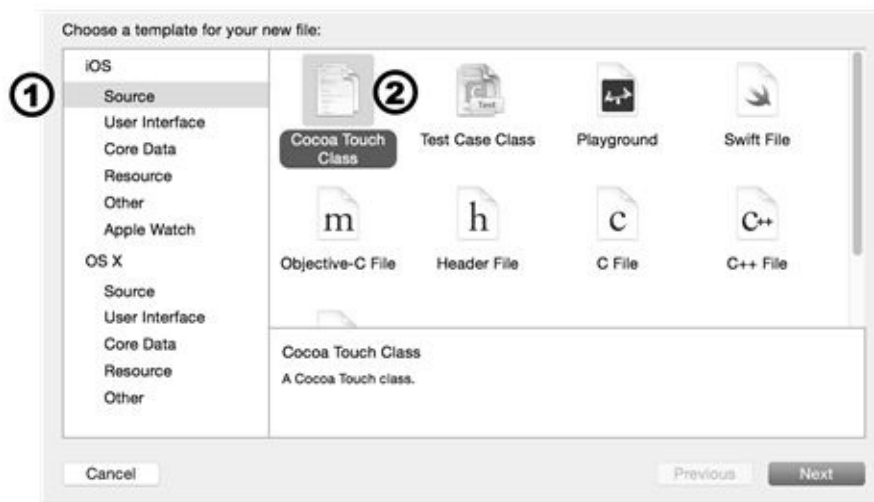


Figure 13-3: *Templates for new files*

There are several options available in this window. The `UIViewController` class is part of the Cocoa Touch API, therefore to create a file with a `UIViewController` subclass for the new main view we have to select the Source option under the iOS menu (Figure 13-3, number 1) and then click on Cocoa Touch Class (Figure 13-3, number 2). Another window opens to ask for the name and the class we want to create our subclass from.



Figure 13-4: *Defining the subclass*

The first value we have to insert is the name of the class (Figure 13-4, number 1). This value is used as the name of the file and as the name for the subclass. It is good practice to start with a word that describes the purpose of the main view and end with the words “ViewController” to reflect the type of class declared inside the file (for this example we called it `SecondViewController`). The next option is the superclass for our subclass. For single main views this is the `UIViewController` class. The last two options are usually left as is. One determines if the process will create a new main view (this is not necessary because we have already created the view in the Storyboard), and the other determines the language Xcode is going to use to create the file (the last language we use is always shown by default). After the options are selected, the file is created and included with the rest.

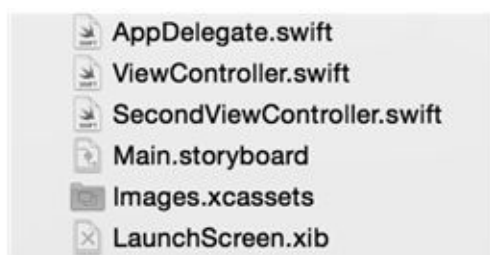


Figure 13-5: New file called *SecondViewController.swift* added to the project's files

At this point, we have the main view in the Storyboard and the file with its view controller, but they are not connected yet. Xcode doesn't assign the new file to its main view automatically; we have to do it ourselves from the Identity Inspector panel. This is one of the configuration panels available in the Utilities Area panel when the Scene is selected. To open the panel, select the Scene and click on the third button at the top of the Utilities Area (circled in Figure 13-6).

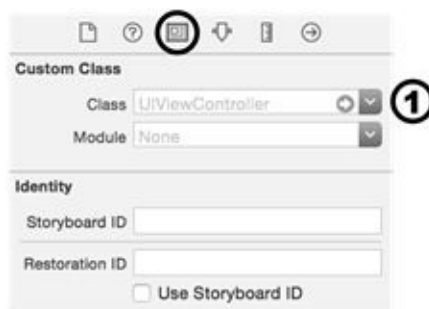


Figure 13-6: Identity Inspector panel

The subclass that controls the view is specified by the first option (Figure 13-6, number 1). We can write the name of the class or select it from the list opened by the arrow on the right side of the field. Figure 13-6 illustrates what the panel looks like when we select the `SecondViewController` class.

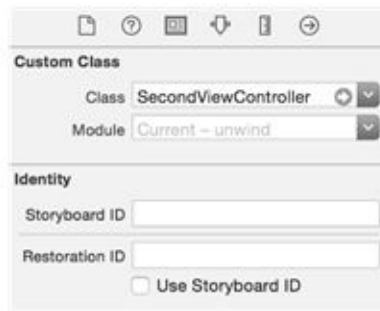


Figure 13-7: Class assigned to the main view

Now the main view has its own view controller and we are ready to go. The procedure to create the interface for each view and connect it to our code is the same as before. We have to drag the elements we want to add to each view and then connect them to the code in their respective view controllers (**ViewController** for the view on the left and **SecondViewController** for the view on the right). Xcode simplifies this process by showing the corresponding view controller file in the Assistant Editor area when a main view is selected in the Storyboard (Figure 5-20).

Do It Yourself: Create the file for the main view added before and call it `SecondViewController.swift`. Select the Scene by clicking on the bar at the top or on the icon that represents the view controller (Figure 5-13) and then open the Identity Inspector panel in the Utilities Area, as shown in Figure 13-6. Assign the new subclass you have just created to the main view (Figure 13-6, number 1). Open the Assistant Editor (Figure 5-20). Click on the main views on the Storyboard to select them. If their corresponding view controller classes are not shown in the Assistant Editor, check that the option at the top is set to Automatic (Figure 6-99).

The Basics: The Assistant Editor shows files depending on the option selected from its menu (Figure 6-100). If you want the assistant to automatically show the view controller file corresponding to the selected Scene, you have to activate the Automatic option from this menu (Figure 6-99).

Segues

Once we have the main views necessary for our interface we have to connect them together. For this purpose, the UIKit framework introduces the **UIStoryboardSegue** class. From this class we can create objects called *Segues* that are in charge of connecting main views and transitioning from one view to another. There are different types of Segues

available.

Show—This Segue produces a transition from right to left. The view controllers are stored in an array to keep track of the sequence.

Show Detail—This Segue shows a detail view (a second view in the same screen) when there is enough space available (e.g., in iPads), or replaces the current view by the new one like a Show Segue when the space is limited (e.g., in iPhones).

Present Modally—This Segue presents the new main view over the current view. The type of transition is set by the `modalTransitionStyle` property of the view controller that triggers the transition. The possible values are `CoverVertical` (the view appears from the bottom), `FlipHorizontal` (the views are turned around), `CrossDissolve` (the views are dissolved), and `PartialCurl` (the old view is curled up to reveal the new view).

Present As Popover—This Segue presents a main view as a popover (studied in Chapter 16).

Custom—This Segue allows developers to provide a custom transition.

Segues are fully integrated to the Storyboard and are created by control-dragging lines between Scenes, as we do to create constraints or Actions and Outlets, but because they are triggered by user interaction they can be created from one Scene to another or from a specific element, such as a button, depending on how we want the user to be able to navigate. The following example uses a simple interface with a list of buttons inside the initial view (the view pointed by the arrow) and an Image View inside the second view. The purpose is to provide buttons in one view the user can tap to select a picture and then transition to a second view that shows the picture. The second view also includes a Slider to rate the image, as illustrated by Figure 13-8.

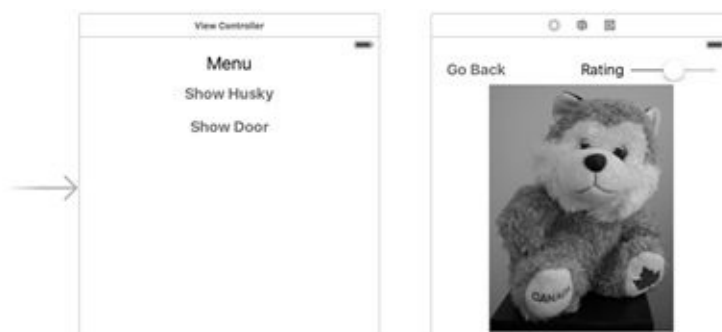


Figure 13-8: *Interface with two screens*

If we run the app at this point, only the initial view is shown on the screen and the buttons don't perform any action. To let the user transition to the second view we have to create a Segue from the button Show Husky to the second view. The process is illustrated

by Figure 13-9.



Figure 13-9: Control-Drag to create a Segue from a button to the second view

The line has to be dragged from the button to the main view. When the mouse's button is released, Xcode opens a menu to select the type of Segue we want to create.

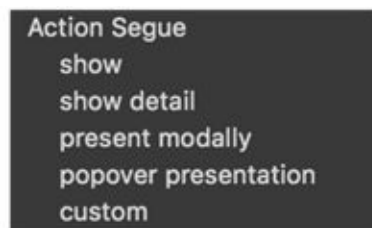


Figure 13-10: Popup menu listing the type of Segues available

Independent view controllers like those implemented in this example can only be connected using a Modal Segue (Presentation) or a popover Segue (Popover Presentation) because their goal is to show information that is not associated with the app's main functionality, such as the app's settings or a login form. The Modal Segue opens the new view over the current view, and the popup Segue generates a view that partially covers the current view (depending on the space available). What Segue to implement depends on the requirements of our application and the type of information contained in the view to be presented. For this example we decided to use a Modal Segue (also known as Presentation or Present Modally).



Figure 13-11: Segue on the Storyboard

IMPORTANT: This chapter is an introduction to Segues and navigation. We will see more complex examples in further chapters and study Modal Segues and popovers in more detail in Chapter 16.

Xcode inserts an arrow in the Storyboard between both views to indicate the connection and represent the Segue. After adding the Segue, the Storyboard shows two arrows, one pointing to the view on the left and another connecting both views. The single arrow on the left indicates what view is going to be shown as soon as the app is executed, and the arrow in the middle is pointing to the view that is going to be shown when the Segue is triggered (the Show Husky button is pressed). The arrows illustrate the path or possible paths the user can follow while interacting with our application. This is the main function of the Storyboard: to help us create a story.

Do It Yourself: Add the label “Menu” and two buttons called Show Husky and Show Door to the initial view. Add a button called Go Back, the label “Rating”, a Slider, and an Image View to the second view. From the Utilities Area set the minimum value for the Slider to 0 and the maximum value to 5. Add the image of the husky to the Assets Catalog and assign it to the Image View. Apply all the necessary constraints to get an interface similar to Figure 13-8. Control-drag a line from the button Show Husky to the second view, as shown in Figure 13-9. Select the Segue Present Modally. Run your app and press the Show Husky button. You should see the second view transitioning from the bottom of the screen to the top.

The Basics: Segues may be deleted and created again to modify the path, and you can also drag the arrow indicating the initial view to another view if later you decide the application should start somewhere else.

The Modal Segue was introduced with the purpose to connect single views and present additional information, but still offers a few options for configuration. Among those options is a list of transition processes we can choose from. The transition by default is called *Cover Vertical* and it scrolls the view from bottom to top, but we can change it from the Utilities Area. When we click on a Segue on the Storyboard, the Utilities Area presents a configuration panel to edit the Segue.

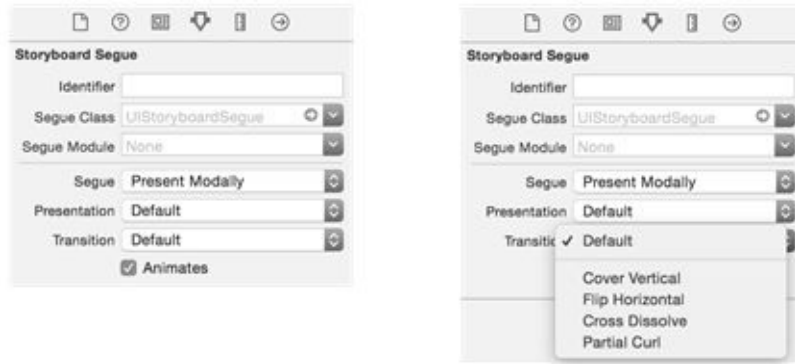


Figure 13-12: *Configuring the Segue*

From the panel shown in Figure 13-12 we can specify a name to reference the Segue from code, set our own custom Segue or select the type of transition. The picture on the right shows the Transition menu with the types available. Selecting any of these options changes the way the view is presented on the screen.

IMPORTANT: Custom transitions may be defined by creating a subclass of the `UIStoryboardSegue` class and overwriting its `perform()` method. The topic is beyond the scope of this book. For more information, visit our website and follow the links for this chapter.

The Basics: By default, modal views are presented full screen, but Segues include an option called Presentation to select other types of presentation (Figure 13-12). This is particularly useful for iPads and their big screens. We will study how to create applications for iPads in Chapter 16.

Unwind Segues

Segues allow us to move forward in the Storyboard, from the view the users are seeing on the screen to the view they want to see. The process to move backwards requires a special kind of Segue called *Unwind Segue* that is created following a few unconventional steps. We have to write an Action on the view that triggers the action Segue and then connect to that action the element in the second view that triggers the Unwind Segue. This is because the view controller responsible for transitioning back to the previous view is the one that presented the second view. In our example, this is the `ViewController` class.

Listing 13-1: Adding the Action for the Unwind Segue

```
import UIKit
```

```

class ViewController: UIViewController {
    @IBAction func goBack(segue: UIStoryboardSegue) {
    }
}

```

Actions are methods that are executed after the user interacts with an element on the interface. So far we have taken advantage of Storyboard features to generate the code in our class, but we can also write the Action ourselves and then create the connection. This is the procedure we have to follow to create the Unwind Segue. Listing 13-1 shows an example of how to write the Action in the view controller that presents the second view. The method can have any name we want but it has to include a parameter of type `UIStoryboardSegue` to provide access to the view controller that is been removed. Once the code for the Action is ready, we have to connect the element in the second view to it. This is done by control-dragging a line from the element to a button at the top of the Scene called *Exit*.

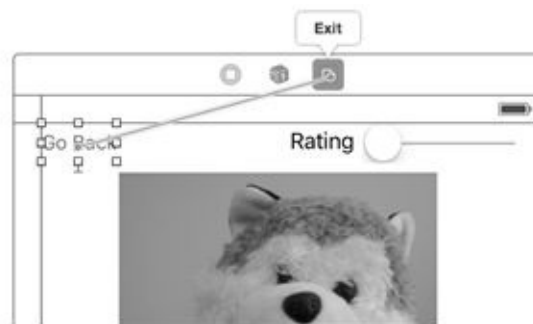


Figure 13-13: Exit button

In Figure 13-13 the Go Back button in the second view is connected to the Exit button. When the mouse is released over the Exit button, a menu shows the Actions available.



Figure 13-14: Menu to select the Action

Once the `goBack` Action is selected from this menu, the Go Back button is connected to the Action defined before in the **ViewController** class and the process is over. The system will remove the view every time the button is pressed.

Do It Yourself: Write the Action in your **ViewController** class as shown in Listing 13-1. Control-drag a line from the Go Back button in the second view to its Exit button, as

shown in Figure 13-13. Select the **goBack** action from the menu (Figure 13-14). Run the application and press the Show Husky button to present the second view on the screen. Press the Go Back button to remove the view.

The Action in our **ViewController** class was declared with no statements. This is enough to perform the Unwind Segue, but we can take advantage of this method and its parameter to retrieve information about the view controllers that participate in the transition. The **UIStoryboardSegue** class includes the following properties for this purpose.

sourceViewController—This property returns a reference to the view controller that is displayed at the beginning of the transition. The value returned is a generic **UIViewController** object that we have to cast to the corresponding type.

destinationViewController—This property returns a reference to the view controller that is displayed at the end of the transition. The value returned is a generic **UIViewController** object that we have to cast to the corresponding type.

identifier—This property returns the string we used in Interface Builder to identify the Segue. Using this property we are able to know what Segue was triggered and get information about the view that is going to be shown on the screen.

The **sourceViewController** and **destinationViewController** properties are required depending on the process. If we want to access the view controller that is going to be removed, we read the **sourceViewController** property (the origin of the transition), but if we want to access the view controller that is going to be presented on the screen, we access the **destinationViewController** (the destination of the transition). To illustrate this process, we are going to expand the **SecondViewController** class to process the input from the Slider and add a property we can read when the Unwind Segue is triggered.

Listing 13-2: Processing changes in the Slider

```
import UIKit

class SecondViewController: UIViewController {
    @IBOutlet weak var sliderRating: UISlider!
    var rating: Int = 0
    override func viewDidLoad() {
        super.viewDidLoad()
        sliderRating.value = Float(rating)
    }
    @IBAction func changeRating(sender: UISlider) {
        let value = round(sender.value)
```

```

sliderRating.value = value
rating = Int(value)
}
}

```

The code of Listing 13-2 adds an integer property called **rating** to the **SecondViewController** class to keep track of the rating selected by the user. The code also includes an Outlet for the Slider to set its initial value equal to the value of **rating** and an Action to update the value of this property every time the Slider is moved (see **UISlider** in Chapter 4 and Chapter 5, Listing 5-11).

Notice that the Slider works with consecutive floating-point values and our rating is established by integers between 0 and 5. Before to assign the selected value to the **rating** property we have to round the number to the nearest integer with the **round()** function and assign it back to the Slider to reflect the change on the screen. This makes the indicator jump from one integer to another, helping the user identify the rating that wants to assign to the picture.

Now we can read the **rating** property from the unwind Action in the **ViewController** class.

Listing 13-3: Reading properties from the source

```

import UIKit

class ViewController: UIViewController {
    var ratingHuscky: Int = 0
    @IBAction func goBack(segue: UIStoryboardSegue) {
        let controller = segue.sourceViewController as! SecondViewController
        ratingHuscky = controller.rating
    }
}

```

The Action for the Unwind Segue receives its **UIStoryboardSegue** object. Reading the **sourceViewController** property of this object we can get a reference to the view controller that is being removed and access its properties. In our example, this is the **SecondViewController**. We get this value, cast it as **SecondViewController**, and store it in the **controller** constant. Now that we have access to the view controller of the second view, we read its **rating** property and store its value in the **ratingHusky** property. Copying the value to a property in the **ViewController** object is necessary because the **SecondViewController** object and all of its properties are destroyed as soon as the execution of the Action is over and the view is removed.

This is the first part of the process; we let the user select a rating and when it moves back to the menu, we preserve the value in the **ratingHusky** property. If we run the

application at this moment and move the Slider to specify a different rating, once we go back to the initial view and tap the Show Husky button a second time, the Slider is again in its initial position. This is because the value of the Slider is always set as the value of the `rating` property when the view is loaded. To get the Slider to show the value previously selected, we have to send the value of the `ratingHusky` property in the `ViewController` object to the `SecondViewController` object every time the Show Husky button is pressed. The `UIViewController` class provides the following methods for this purpose.

`prepareForSegue()`—This method is called by the system in the view controller object when a Segue is triggered and before the transition is initiated. We can overwrite this method in our view controller to access the destination view controller and modify its properties.

`performSegueWithIdentifier(identifier, sender: sender)`—This method triggers a Segue identified with the string specified by the `identifier` attribute. The `sender` attribute is the object that triggers the Segue (usually declared as `self`).

The method we need to send information to the second view controller is `prepareForSegue()`. This is one of those methods that are called by the system in the view controller and can be overwritten to perform custom tasks. In this case we can overwrite it to access the view controller that is going to be opened and modify its properties before its view is shown on the screen. The following example expands the code in our `ViewController` class to update the value of the `rating` property in the `SecondViewController` class and get the Slider to show the right value.

Listing 13-4: Sending values to the second view controller before its view is loaded

```
import UIKit

class ViewController: UIViewController {
    var ratingHusky: Int = 0

    @IBAction func goBack(segue: UIStoryboardSegue) {
        let controller = segue.sourceViewController as! SecondViewController
        ratingHusky = controller.rating
    }

    override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        let controller = segue.destinationViewController as! SecondViewController
        controller.rating = ratingHusky
    }
}
```

Now the application is complete. The `prepareForSegue()` method is executed after the Show Husky button is pressed and before the second view is shown on the screen, updating the value of the `rating` property with the value of the `ratingHusky` property. As soon as the second view is loaded, the value of the `rating` property is assigned to the Slider to set its position. When the user presses the Go Back button, the Exit action is executed in the `ViewController` class and the process is reversed: the `ratingHusky` property is updated with the current value of the `rating` property. The result is that the rating set by the user is always preserved in the `ratingHusky` property and used to update the Slider every time the image is shown on the screen.

Do It Yourself: Copy the code of Listing 13-4 in your `ViewController` class. Create an Outlet called `sliderRating` and an Action called `changeRating()` for the Slider in the second view. Complete the `SecondViewController` class with the code of Listing 13-2. Run the application. The Slider should always reflect the rating previously selected.

IMPORTANT: Never try to access Outlets of one view controller from another. Always transfer data through normal properties because there is no guarantee that the Outlets will be connected to the objects on the Storyboard until the view is fully loaded. If you need to modify an element directly, declare a property, assign the new value to that property, and then assign the value of that property to the Outlet from any of the methods studied before, such as `viewDidLoad()` or `viewDidAppear()`, as we did in Listing 13-2.

The same procedure can be applied to the second button of the menu to let the user select another image. All we need to do is add a new Scene to the Storyboard and connect it with the Show Door button using another Modal Segue, as shown in Figure 13-15.

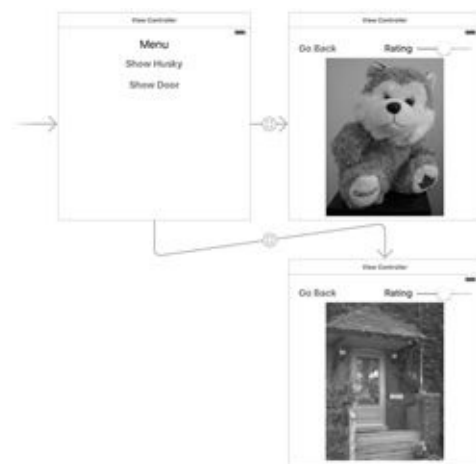


Figure 13-15: Interface with three views

The process to add the Scene is the same as before. We drag the View Controller option to the Storyboard and build the interface. As well as the second view, this view also needs its own view controller. For this example we called it **ThirdViewController**. Because we use the same types of elements and process the same information, the code for this view controller is the same defined for the **SecondViewController** class in Listing 13-2, including the Outlet and Action for the Slider. What changes is how the **ViewController** class manages the information that is coming from and going to these view controllers. Now we have two Segues, one that opens the view with the husky and another that opens the view with the door, and therefore the code in the **ViewController** class has to recognize the Segue that is being triggered to know how to proceed. This is the reason why the Segue's configuration panel includes a field called *Identifier* (Figure 13-12). The string introduced in this field is assigned to the **identifier** property of the **UIStoryboardSegue** object and that is how we know what Segue was triggered.

As we explained before, to change the Segue's configuration we have to select it and open the Utilities Area (Figure 13-12). For the normal Segues is easy, we just have to select them from the Storyboard and modify their values, but the only way to select an Unwind Segue is from the Document Outline panel. The items that represent the Unwind Segues in this panel are preceded by a small arrow, as illustrated by Figure 13-16 (number 1).

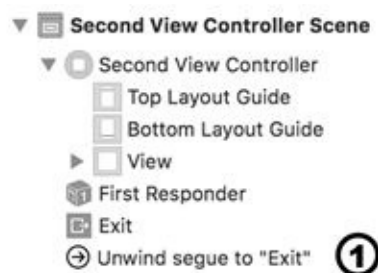


Figure 13-16: *Unwind Segue in the Document Outline panel*

For our example, we called the Segues that point to each view *showHusky* and *showDoor* and the Unwind Segues *removeHusky* and *removeDoor*, respectively. Once we have all the Segues identified, our **ViewController** class can check their values and modify the right properties. The next example shows a possible implementation.

Listing 13-5: *Processing values according to the Segue*

```
import UIKit
```

```
class ViewController: UIViewController {
    var ratingHuscky: Int = 0
    var ratingDoor: Int = 0
    @IBAction func goBack(segue: UIStoryboardSegue) {
        if segue.identifier == "removeHusky" {
```

```

let controller = segue.sourceViewController as! SecondViewController
ratingHusky = controller.rating
} else if segue.identifier == "removeDoor" {
let controller = segue.sourceViewController as! ThirdViewController
ratingDoor = controller.rating
}
}

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showHusky" {
        let controller = segue.destinationViewController as! SecondViewController
        controller.rating = ratingHusky
    } else if segue.identifier == "showDoor" {
        let controller = segue.destinationViewController as! ThirdViewController
        controller.rating = ratingDoor
    }
}
}

```

The `prepareForSegue()` method is executed every time a normal Segue is triggered. The method compares the value of the `identifier` property of the `segue` object with the strings "showHusky" and "showDoor". If the value is equal to "showHusky", it gets the reference to the `SecondViewController` object and updates the value of its `rating` property with the value of the `ratingHusky` property. If, on the other hand, the value of `identifier` is equal to "showDoor", it gets the reference to the `ThirdViewController` object and updates its `rating` property with the value of the `ratingDoor` property. A similar control is performed for the `goBack` action, although this time we have to check the identifiers for the Unwind Segues ("removeHusky" and "removeDoor").

Do It Yourself: Add a new view to the Storyboard and recreate the same interface of the second view but with the image of the door (you can copy and paste the elements from one view to another). Add a Segue from the Show Door button to the new view. Create a file with a subclass of `UIViewController` called `ThirdViewController` and assign it to the view. Create an Outlet called `sliderRating` and an Action called `changeRating` for the Slider and complete the code of the `ThirdViewController` with the code of Listing 13-2. Select the Segues on the Storyboard and assign the identifiers "showHusky" and "showDoor", respectively. Select the Unwind Segues from the Document Outline panel and assign the identifiers "removeHusky" and "removeDoor", respectively. Copy the code of Listing 13-5 in the `ViewController` class. Run the application. The Sliders should always reflect the rating previously selected for each image.

Segues in Code

Of course, implementing the previous technique we could, in theory, build a fully functional interface and a whole application. But designing an application of this kind that contemplates every possible feedback from the user or new information available would be impossible (in our example, we would have to add buttons and views for every new image incorporated to the app or generated by the user). Real applications demand some sort of process in which the content of a view is generated automatically according to the information requested by the user or produced by the code. In this scenario, a view provides an updated list of available options and a single Segue connects this view with the view in charge of presenting the information for every option. The Segue is therefore not triggered by a specific element but from code calling the `performSegueWithIdentifier()` method introduced before.

As we will see in Chapter 14, the list of options is usually presented with a table. A table organizes the information and helps users find what they are looking for, especially when the list of options is extensive, but there are other alternatives available that are applicable in more simple situations. The following example uses a Picker View to present a predefined list of pictures the user can pick from.

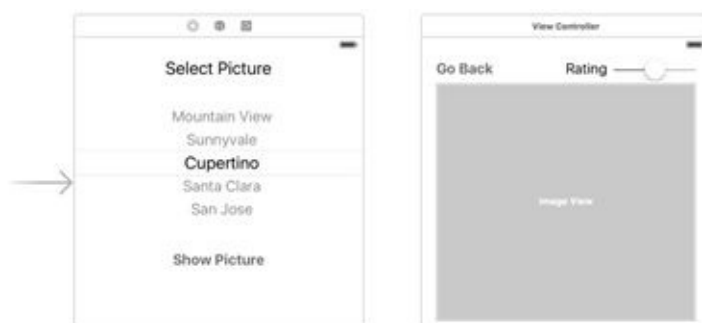


Figure 13-17: Interface that adapts to the options available

The initial view for this new application contains a Picker View and a button. When the button is pressed, the code for this view controller executes the `performSegueWithIdentifier()` method to trigger a Segue that transitions to the second view. There are different ways to create this Segue. The most simple is to control-drag a line from the icon that represents the view controller on the top-left side of the Scene to the second view, as shown in Figure 13-18.



Figure 13-18: Creating a Segue from one view to another

IMPORTANT: Segues that are not associated with an element must always have an identifier so they can be referenced from code (see Figure 13-12).

The Basics: Xcode lets us zoom in and out the Storyboard. You can select the level of zoom by pressing the Command and – keys or Command and + keys. When the Storyboard is zoomed out, you can't add or select the elements of the interface, but you can move the Scenes around and connect one Scene to another with Segues.

No matter how the Segue is triggered, the system always calls the `prepareForSegue()` method before performing the transition. In our example we use this method to report to the second view the item selected on the picker. With this information, the second view loads the corresponding image and sets the Slider to represent its rating. The following is the code for the `ViewController` class.

Listing 13-6: Processing the options

```
import UIKit

class ViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {
    @IBOutlet weak var pickerPictures: UIPickerView!
    var picturesList: [String]!
    var ratings: [Int]!
    var selectedPicture: Int!

    override func viewDidLoad() {
        super.viewDidLoad()
        pickerPictures.delegate = self
        pickerPictures.dataSource = self
        picturesList = ["Husky", "Door"]
        ratings = [0, 0]
        selectedPicture = 0
    }

    func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int {
        return 1
    }

    func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        return picturesList.count
    }

    func pickerView(pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {
        return picturesList[row]
```

```

}
@IBAction func getPicture(sender: UIButton) {
    selectedPicture = pickerPictures.selectedRowInComponent(0)
    performSegueWithIdentifier("showPicture", sender: self)
}
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    let controller = segue.destinationViewController as! SecondViewController
    controller.rating = ratings[selectedPicture]
    controller.picture = picturesList[selectedPicture]
}
@IBAction func goBack(segue: UIStoryboardSegue) {
    let controller = segue.sourceViewController as! SecondViewController
    ratings[selectedPicture] = controller.rating
}
}

```

Most of the code of Listing 13-6 is conformed by the delegate methods that configure the picker (see `UIPickerView` in Chapter 9), but we also have the `goBack()` action implemented before for the Unwind Segue and the `prepareForSegue()` method to send the information to the second view when the normal Segue is triggered. To store the data, we have created three properties: `picturesList`, `ratings`, and `selectedPicture`. The `picturesList` property is an array of strings to store the names of the pictures available, the `ratings` property is an array of integers to store the rating for each picture, and the `selectedPicture` property is a single integer to store the index of the picture selected from the picker. This last property lets us keep track of the picture we are currently working on. When the `prepareForSegue()` method sends the information to the second view it takes the values from the `picturesList` and `ratings` arrays corresponding to the index determined by the `selectedPicture` property. Likewise, when the `goBack()` method processes the rating set by the user, it stores its value in the position of the `ratings` array also corresponding to the `selectedPicture` property.

The interface includes the Show Picture button to select the picture. This button has been connected with an Action called `getPicture()` that gets the value selected in the picker, assigns it to the `selectedPicture` property, and then executes the `performSegueWithIdentifier()` method to trigger the Segue and load the second view (the Segue was identified in Interface Builder with the name “showPicture”).

Unwind Segues can also be created from the view controller to the Exit button and then triggered by code. To create an Unwind Segue that is not connected to an element, we have to control-drag a line from the icon that represents the view controller at the top of the Scene to the Exit button, as shown in Figure 13-19.



Figure 13-19: *Creating Unwind Segues for the view controller*

The view controller for the second view now has to execute the `performSegueWithIdentifier()` method to trigger the Unwind Segue and let the user go back to the previous screen. For this purpose, we have included an Action for the Go Back button in the `SecondViewController` class (the Unwind Segue was identified with the string “goBack”).

Listing 13-7: *Showing the data*

```
import UIKit

class SecondViewController: UIViewController {
    @IBOutlet weak var sliderRating: UISlider!
    @IBOutlet weak var pictureView: UIImageView!
    var rating: Int = 0
    var picture: String!

    override func viewDidLoad() {
        super.viewDidLoad()
        sliderRating.value = Float(rating)
        pictureView.image = UIImage(named: picture.lowercaseString)
    }

    @IBAction func changeRating(sender: UISlider) {
        let value = round(sender.value)
        sliderRating.value = value
        rating = Int(value)
    }

    @IBAction func goBack(sender: UIButton) {
        performSegueWithIdentifier("goBack", sender: self)
    }
}
```

Besides triggering the Unwind Segue, the code of Listing 13-7 also introduces other changes necessary to process the image received from the initial view. The view controller includes an additional property to store the name of the image selected by the user and load its file. We called this property `picture`, and the Outlet for the Image View `pictureView`. These properties are used to load the file, create the `UIImage` object and assign it to the Image View. Notice that the value of the `picture` property has to be lowercased to match the

names of the files.

Do It Yourself: Create a new Single View Application project. Add a second view with a class called `SecondViewController`. Design the interface according to Figure 13-17. The initial view has to include a picker and a button, and the second view's interface is the same implemented for the previous examples. Create a Present Modally Segue from the initial view to the second view, as shown in Figure 13-18. Assign the identifier "showPicture" to the Segue. Create an Unwind Segue for the second view as shown in Figure 13-19. Open the Document Outline panel, select this Segue and give it the identifier "goBack" from the Attributes Inspector panel. Copy the code of Listing 13-6 into the `ViewController.swift` file and the code of Listing 13-7 into the `SecondViewController.swift` file. Connect the elements to their Outlets and Actions. Run the application. You should be able to select the picture you want to see from the picker and change the rating.

13.2 Navigation Controllers

The view controllers we used so far are called Content View Controllers, because they are in charge of managing their own content. They are created from custom subclasses of the `UIViewController` class and are focused on presenting and controlling a single main view and its elements. For this reason, these view controllers are limited when it comes to replacing their own main view by another or provide the means to organize the interface. This is why the UIKit framework includes several subclasses of the `UIViewController` class that include their own functionality and introduce better alternatives to work with multiple views. The view controllers created from these classes are called *Container View Controllers*, because they contain other view controllers and organize their views. The most widely used container is the Navigation Controller, created from the `UINavigationController` class. This class creates a container view controller that organizes the views in sequential order and provide the tools to navigate through the hierarchy.

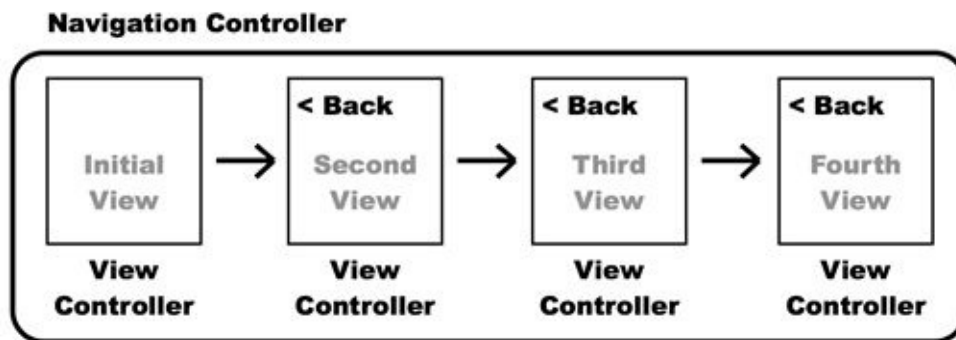


Figure 13-20: View controllers managed by a Navigation Controller

As shown in Figure 13-20, the views managed by a Navigation Controller are presented in sequence. The first view opens the second view, and then the second view opens the third view. When the user decides to go back, the sequence is reversed, the current view is removed, and the previous view is shown on the screen. To keep track of the view controllers in the sequence, the Navigation Controller stores a reference to each one of them in an array called *stack* and then puts or pulls view controllers from the stack every time their views are presented or removed from the screen. The possible sequences are defined in the Storyboard, but the stack is generated in real time. If we define different paths, as illustrated by Figure 13-21, the Navigation Controller only stores on the stack the view controllers on the path the user decides to follow.

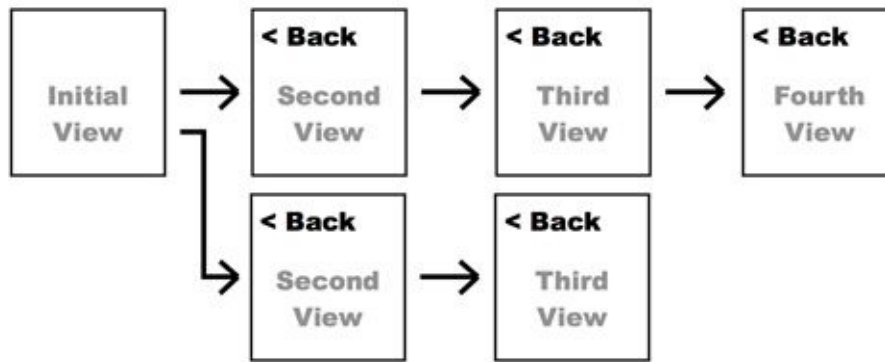


Figure 13-21: Navigation Controller with multiple paths

The difference between building a structure like this with a Navigation Controller rather than the single view controllers and Modal Segues we used before is that Navigation Controllers incorporate additional functionality that make it easy for us to create a story and help the user navigate through our interface. They use a Segue of type Show that transitions from right to left (or left to right when the view is being removed) reflecting on the screen the sequential order of the views in the stack. They also incorporate bars to identify each view and their predecessors, and include their own properties and methods to control the navigation. The following are the most frequently used.

viewControllers—This property is an array that contains references to the view controllers managed by the `UINavigationController` object.

topViewController—This property returns the view controller at the top of the stack (usually the view controller displayed on the screen).

visibleViewController—This property returns the view controller currently shown on the screen. The view controller may belong to the Navigation Controller or to a view opened modally.

popViewControllerAnimated(animated)—This method removes the top view controller from the stack and makes the previous one active. The **animated** attribute is a Boolean value that indicates whether or not the transition should be animated.

pushViewControllerAnimated(controller, animated: animated)—This method adds to the stack the view controller referenced by the **controller** attribute. The **animated** attribute is a Boolean value that indicates whether or not the transition should be animated.

popToRootViewControllerAnimated(animated)—This method removes all the view controllers from the stack except the initial view controller. The **animated** attribute is a Boolean value that indicates whether or not the transition should be animated.

Navigation Controllers in the Storyboard

There are two ways to add a Navigation Controller to the Storyboard. If we have already designed the interface, Xcode offers an option to embed the views in the Storyboard inside a Navigation Controller. We have to select the view we want to be the initial view for the Navigation Controller and then go to the Editor menu at the top of the screen, open the option Embed In and then select Navigation Controller. The other alternative is to drag the Navigation Controller option from the Object Library. This option creates the Navigation Controller along with its initial view. Figure 13-22 illustrates what a single view looks like after it is embedded in a Navigation Controller.

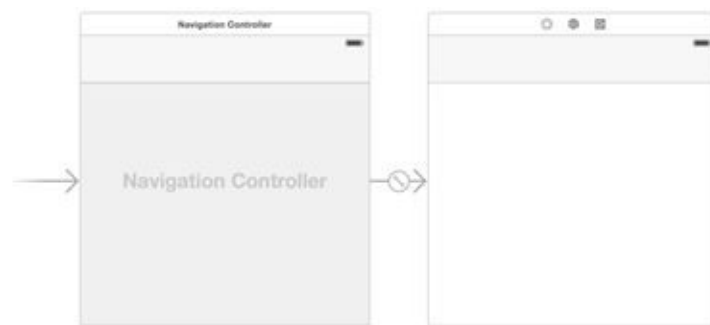


Figure 13-22: Initial view embedded in a Navigation Controller

The view on the left is an empty Scene that represents the Navigation Controller and the view on the right is our initial view. The initial arrow is now pointing to the Navigation Controller and a Segue connects this controller to the initial view. This Segue is called *Root View Controller Segue* and has the purpose to indicate which is the first view the Navigation Controller has to show. When the app is executed, the system loads the Navigation Controller first and then searches for its root view controller and shows it on the screen.

IMPORTANT: If you opt for the option to add the Navigation Controller from the Object Library, there is an additional step you have to follow. When the initial view is deleted and a new one is added to the storyboard, the arrow pointing to the initial view disappears. To get back the arrow and indicate to the system which is the initial view in the Storyboard, we have to select the new view, go to the Attributes Inspector panel, and activate the option *Is Initial View Controller*.

More views are added to the interface as before, but instead of using a Modal Segue to connect the views we have to create Segues of type Show. Figure 13-23, next, illustrates what we see when we add a second view controller and connect it with this Segue. The initial view includes a label with the text “Root View” and a button with the title “Open

Second View”. The button was connected to the second view with a Show Segue (the first option of the popup menu). When the user presses the button, the second view is shown on the screen transitioning from right to left.



Figure 13-23: *Navigation stack in the Storyboard*

Do It Yourself: Create a new Single View Application project. Select the main view, open the Editor menu at the top of the screen, go to Embed In, and select the option Navigation Controller. The Storyboard should look like Figure 13-22. Drag the View Controller option from the Object Library to add another Scene to the Storyboard. Add a label and a button to the initial view and a label to the second view, as illustrated in Figure 13-23. Control-drag a line from the button in the initial view to the second view to create the Segue (Figure 13-9). Select the type Show for the Segue. Create a file with the view controller for the second view called `SecondViewController.swift` as we did before for previous examples (Figures 13-3, 13-4 and 13-7). Run the application and press the button to see the transition.

The following figure shows what we see when we run the application. The picture on the left is the initial screen. If we press the Open Second View button, the Show Segue is triggered and the second view is shown on the screen with a transition from right to left.

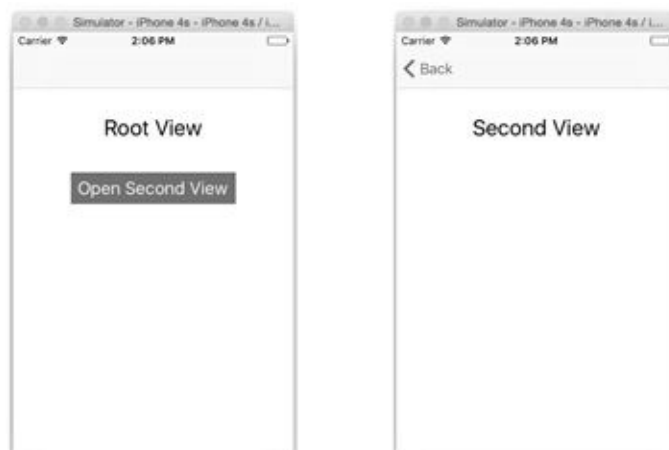


Figure 13-24: *Navigation Controller on the simulator*

Navigation Bar

Besides the new type of Segue and transition, the Navigation Controller introduces a Navigation Bar at the top of every view it manages to help the user navigate. The bar is an object of the `UINavigationController` class, which includes a few properties and methods for configuration.

barStyle—This property determines the style of the bar. It is an enumeration of type `UIBarStyle` with the values `Default` and `Black`.

barTintColor—This property sets or returns the color applied to the bar's background.

tintColor—This property sets or returns the color applied to the bar's items.

translucent—This property takes a Boolean value to indicate whether the Navigation Bar is translucent or not.

titleTextAttributes—This property sets or returns the attributes applied to the bar's title. Its value is a dictionary with the same constants used to define the attributes for an `NSMutableAttributedString` object.

setBackgroundImage(image, forBarMetrics: metrics)—This method assigns an image to the bar's background. The **image** attribute is a `UIImage` object and the **metrics** attribute is an enumeration of type `UIBarMetrics` that associates the image with the state of the bar. The possible values are `Default`, `DefaultPrompt`, `Compact`, and `CompactPrompt`.

The `UINavigationController` class also includes specific properties and methods to reference and configure the Navigation Bar.

navigationBar—This property returns a reference to the `UINavigationController` object that represents the Navigation Bar created for the Navigation Controller.

hidesBarsOnTap—This property takes a Boolean value that indicates whether or not the bars should be hidden when the user taps on the screen.

hidesBarsOnSwipe—This property takes a Boolean value that indicates whether or not the bars should be hidden when the user swipes the finger on the screen.

hidesBarsWhenKeyboardAppears—This property takes a Boolean value that indicates whether or not the bars should be hidden when the keyboard is visible.

hidesBarsWhenVerticallyCompact—This property takes a Boolean value that indicates whether or not the bars should be hidden when the vertical Size Class is Compact.

setNavigationBarHidden(value, animated: animated)—This method hides or shows the Navigation Bar. The **value** attribute is a Boolean value that indicates whether or not the bar is hidden, and the **animated** attribute is another Boolean value that indicates if the process is going to be animated.

IMPORTANT: Navigation Bars and their content are positioned by the system at the top of the screen. They don't require constraints, but the rest of the elements in the interface may be pinned with constraints to the edge of the bar or the edge of the view, depending on whether you want to let the content go behind the bar or not.

The Navigation Bar is created by the `UINavigationController` object and stored in its `navigationBar` property, but the bar can be configured from different places in the code. One option is to create a subclass of the `UINavigationController` class (Figures 13-3 and 13-4) and assign it to the Navigation Controller in the Storyboard (Figures 13-6 and 13-7). This is the same procedure used to create the view controller classes for our view controllers, but instead of selecting the `UIViewController` class as the superclass we use the `UINavigationController` class instead. After the custom class is created, we can configure its bar.

Listing 13-8: *Configuring the Navigation Bar from a subclass of UINavigationController*

```
import UIKit

class NavigationViewController: UINavigationController {
    override func viewDidLoad() {
        super.viewDidLoad()
        hidesBarsOnSwipe = true
        let bar = navigationBar
        bar.barStyle = UIBarStyle.Default
        bar.barTintColor = UIColor(red: 0.9, green: 0.9, blue: 1.0, alpha: 1)
    }
}
```

The properties of the `UINavigationController` class are accessed directly from the class

itself, but to modify the properties of the bar we have to get its reference from the `navigationBar` property. In Listing 13-8 we store this reference in the `bar` constant and then set the style for the bar and its background color.

The creation of a subclass of the `UINavigationController` class to configure the Navigation Bar is usually not necessary because the `UIViewController` class offers a property called `navigationController` to get a reference to the Navigation Controller the view controller belongs to. From this reference, we can read the `navigationBar` property of the Navigation Controller to access and configure the bar from any of the view controllers managed by the Navigation Controller. For example, we can do it from the view controller that controls the initial view.

Listing 13-9: Configuring the Navigation Bar from the view controller of the initial view

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let nav = navigationController!
        nav.hidesBarsOnSwipe = true

        let bar = nav.navigationBar
        bar.barStyle = UIBarStyle.Default
        bar.barTintColor = UIColor(red: 0.9, green: 0.9, blue: 1.0, alpha: 1)
    }
}
```

The process is the same but we have to get the reference to the Navigation Controller object from the `navigationController` property first and then access its `navigationBar` property.

The previous two examples configure the Navigation Bar of a specific Navigation Controller, but our interface may contain several of them. If we want all the Navigation Controllers to share the same configuration, we can take advantage of a method accessible from the `UINavigationController` class called `appearance()`. This method is defined in the `UIAppearance` protocol and the protocol is adopted by the `UINavigationController` class. From the reference returned by the `appearance()` method we can change the class' configuration and then every object created from the `UINavigationController` class will have by default that configuration. Because this procedure modifies the `UINavigationController` class, we have to perform it before any bar is created. And the best place to do it is from the application's delegate (the `AppDelegate.swift` file).

Listing 13-10: Configuring the Navigation Bar from the app's delegate

```

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let bar = UINavigationController.appearance()
        bar.barStyle = UIBarStyle.Default
        bar.barTintColor = UIColor(red: 0.9, green: 0.9, blue: 1.0, alpha: 1)
        return true
    }
}

```

As we have seen in Chapter 5, any code we want to process right after the application is executed has to be written inside the `application(application, didFinishLaunchingWithOptions: launchOptions)` method of the `AppDelegate` class (see Listing 5-1). In this example, the `appearance()` method is executed first from the `UINavigationController` class to get a reference to the Navigation Bar and then the bar is configured as always.

Do It Yourself: The examples of this section assume that you have created an interface similar to Figure 13-21 and the view controllers `ViewController` and `SecondViewController` were assigned to the initial view and the second view, respectively. Select the initial Scene to see the `ViewController` class on the Assistant Editor. Complete the class with the code of Listing 13-9 and run the application. You should see a violet Navigation Bar on the screen.

Items

Navigation bars are just empty views, their content is not generated by the bar itself but by objects of the `UINavigationItem` class. In turn, these objects are containers for other views that can represent things like the title of the main view and buttons. Figure 13-25 illustrates how the structure is organized.



Figure 13-25: *Elements of the Navigation Bar*

Navigation Items are created automatically by the system along with the Navigation Bar, but the Storyboard only includes the one necessary for the initial view; we have to add the Navigation Items for the rest of the views from the Object Library. Figure 13-26 shows this option.

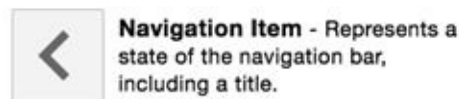


Figure 13-26: *Navigation Item option in the Object Library*

Navigation items may include a title and a prompt, displayed at the center of the bar, and buttons on the sides. We can add all the buttons we need (taking into consideration the limited space available), but one button is generated automatically by the Navigation Controller to help the user navigate back to the previous view. By default, this button has the title Back, as shown in Figure 13-24, but we can modify its title, along with the title and prompt for the view, from the Attributes Inspector panel in the Utilities Area, as shown in Figure 13-27.

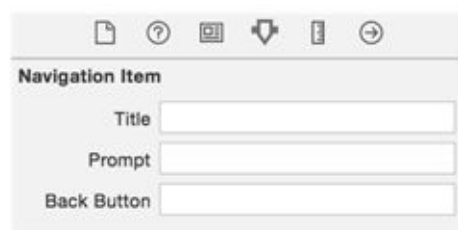


Figure 13-27: *Configuring the Navigation Item*

When a Navigation Item is selected, three fields are shown in the Attributes Inspector panel: Title, Prompt, and Back Button. The values inserted in the first two fields are shown in the current view, but the value of the third field corresponds to the title of the Back button the system is going to show in the next view to go back to this one. For example, if we select the navigation item of the initial view and insert the title “Menu” for the view and the title “Go Back” for the button, we will see something similar to Figure

13-28. The title for the view is shown at the center of the Navigation Bar and the Back button now says “Go Back”.

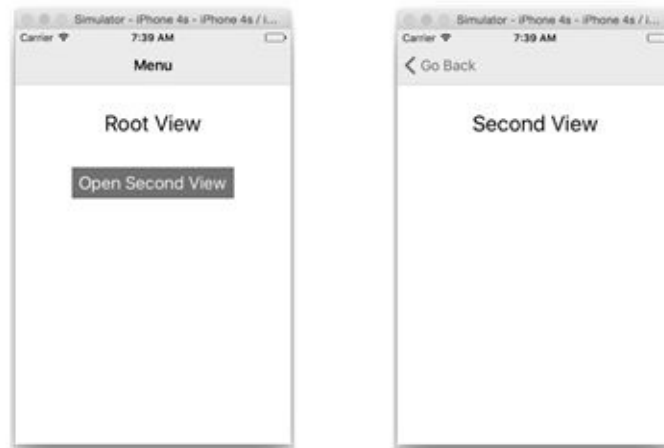


Figure 13-28: *Navigation Items in the simulator*

Do It Yourself: As we mentioned, Xcode includes the Navigation Item for the initial view but not for the rest of the views added to the Navigation Controller. To add a title and buttons to the second view of our example, you have to drag the Navigation Item option from the Object Library to its Navigation Bar (Figure 13-26). After both views have their Navigation Item, click on the one you want to configure and modify its values from the Attributes Inspector panel (Figure 13-27). Run the application. You should see something similar to Figure 13-28.

IMPORTANT: If a title is provided but no value is defined for the Back button, the button adopts the title of the view as its own title (the Back button in our example will be shown with the title “Menu”). This follows Apple’s guidelines that recommend using the view’s title as the title of the Back button to help users determine their position in the interface and navigate back.

Custom buttons can also be added from the Object Library. These are not regular buttons; they are created from the `UIBarButtonItem` class. The option is called Bar Button Item.

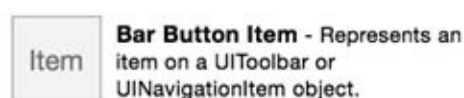


Figure 13-29: *Bar Button Item option in the Object Library*

The bar buttons may be added to the left or the right side of the bar, but if we add it to the left in a view that is not the initial view, the button replaces the Back button. Figure 13-30 shows the interface of our example after the Bar Button Item was dragged to the right side of the initial view’s Navigation Bar.

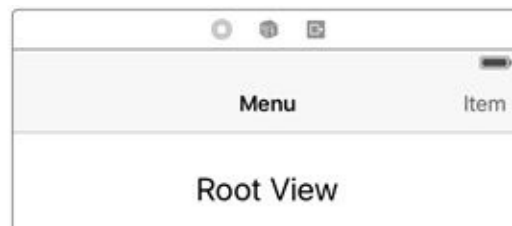


Figure 13-30: Bar Button Item in the Navigation Bar

The title of the button by default is “Item”, but we can change it from the Attributes Inspector panel. From this panel we can also assign a designed to the button, a custom image, and a color. Figure 13-31 shows what we see when the Camera option of the System Item property is selected.

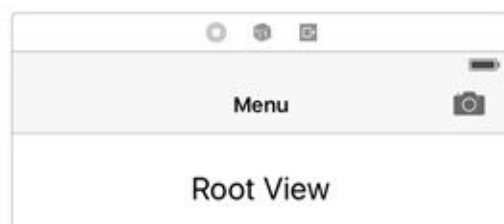


Figure 13-31: Predefined Bar Button Item

Bar buttons are created from the `UIBarButtonItem` class instead of the `UIButton` class because they were specifically designed to fit in the limited space of a Navigation Bar or a Tool Bar, but they behave exactly like any other button. We can create Segues or associate them with Actions in our code. The following example adds an Action for the button in the `ViewController` class to print a message on the console when the button is pressed.

Listing 13-11: Adding an Action for a bar button

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let nav = navigationController!
        nav.hidesBarsOnSwipe = true
        let bar = nav.navigationBar
```

```

bar.barStyle = UIBarStyle.Default
bar.barTintColor = UIColor(red: 0.9, green: 0.9, blue: 1.0, alpha: 1)
}
@IBAction func showCamera(sender: UIBarButtonItem) {
    print("Activate Camera")
}
}

```

Do It Yourself: Drag a Bar Button Item to the right side of the Navigation Bar in the initial view. Select the button and change its properties from the Attributes Inspector panel. Try the other predefined buttons available. Control-drag a line from the button to the `ViewController` class to create an action called `showCamera()`. Complete the class with the code of Listing 13-11.

Of course, the items of a Navigation Item object may be modified from code. The `UIViewController` class offers a property called `navigationItem` to access the object and its properties. The following are the most frequently used.

title—This property sets or returns the title. It is an optional of type `String`.

prompt—This property sets or returns the prompt. It is an optional of type `String`.

backBarButtonItem—This property sets the Back button for the view. It is an optional of type `UIBarButtonItem`.

hidesBackButton—This property is a Boolean value that determines whether or not the Back button is shown. The property must be defined in the view that shows the button.

leftBarButtonItems—This property is an array of `UIBarButtonItem` objects that defines the bar buttons for the left side of the bar.

rightBarButtonItems—This property is an array of `UIBarButtonItem` objects that defines the bar buttons for the right side of the bar.

leftItemsSupplementBackButton—This property is a Boolean value used to indicate whether the Back button is shown along with the items on the left side of the bar or not. By default, the Back button is not shown when there are custom buttons in its place. We can change this behaviour by setting this property to `true`.

The following example defines the view's title and the title for the Back button from code. The view's title is assigned directly to the `title` property of the `UINavigationController` object, but there is no property for the title of the Back button; we have to create a new

`UIBarButtonItem` object with the title we want and use it to replace the original.

Listing 13-12: Configuring the navigation item form code

```
import UIKit
```

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let nav = navigationController!
        nav.hidesBarsOnSwipe = true
        let bar = nav.navigationBar
        bar.barStyle = UIBarStyle.Default
        bar.barTintColor = UIColor(red: 0.9, green: 0.9, blue: 1.0, alpha: 1)

        let item = navigationItem
        item.title = "First View"
        let button = UIBarButtonItem()
        button.title = "Close"
        navigationItem.backBarButtonItem = button
    }
    @IBAction func showCamera(sender: UIBarButtonItem) {
        print("Activate Camera")
    }
}
```

In Listing 13-12 we get the reference to the navigation item from the `navigationItem` property of the view controller, define the title for the view, and then create the `UIBarButtonItem` object to replace the Back button. The `UIBarButtonItem` class is a subclass of the `UIBarItem` class, which provides some properties to configure basic aspects of bar buttons.

title—This property sets or returns the button’s title. It is an optional of type `String`.

image—This property sets or returns the image for the button. It is an optional of type `UIImage`.

tag—This property sets or returns the button’s tag value. It is of type `Int`.

In Listing 13-12 we use the `title` property to define the button’s title. At the end, the button is assigned to the `backBarButtonItem` property of the navigation item, effectively replacing the original button for this one. Now every time we open to the second view, the

button says “Close”.

The button assigned to the `backBarButtonItem` property replaces the Back button and therefore only requires a value for its `title` property (the action to return to the previous view is assigned automatically by the system), but normal buttons need more information, such as the method they are going to execute and where that method is (the target/action mechanism introduced by the `UIControl` class studied in Chapter 4). The `UIBarButtonItem` class includes two practical initializers to provide this information in the initialization process.

`UIBarButtonItem(barButtonItem: type, target: target, action: action)`—This initializer creates a system button with the target and action defined by its attributes. The **`type`** attribute is an enumeration of type `UIBarButtonItem` with the values `Done`, `Cancel`, `Edit`, `Save`, `Add`, `FlexibleSpace`, `FixedSpace`, `Compose`, `Reply`, `Action`, `Organize`, `Bookmarks`, `Search`, `Refresh`, `Stop`, `Camera`, `Trash`, `Play`, `Pause`, `Rewind`, `FastForward`, `Undo`, `Redo`, and `PageCurl`. The **`target`** action is a reference to the object where the action is executed. And the **`action`** attribute is a string with the name of the method to be executed when the button is pressed.

`UIBarButtonItem(title: title, style: style, target: target, action: action)`—This initializer creates a custom button with the title, target, and action defined by its attributes. The **`title`** attribute is an optional value of type `String`. The **`style`** attribute is an enumeration of type `UIBarButtonItemStyle` with the values `Plain` (normal button) and `Done`. And the **`target`** and **`action`** attributes fulfill the same function as in the previous initializer.

The Basics: The `FlexibleSpace` and `FixedSpace` values available for system buttons instead of creating buttons they generate empty space to separate the rest of the buttons. These options are also available in the Object Library.

Besides these initializers, we can also use an empty initializer, as we did in Listing 13-12, and define the values later from the properties of the class.

`style`—This property sets the style of the button. It is an enumeration of type `UIBarButtonItemStyle` with the values `Plain` (normal button) and `Done`.

`target`—This property sets the target of the action.

`action`—This property sets the name of the method to be executed when the button is pressed.

All the tools described above to configure the Navigation Bar provide enough flexibility to generate a Navigation Bar with the style we want for our application. The following code illustrates how to configure the bar and its items.

Listing 13-13: Configuring the Navigation Bar and its items

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let nav = navigationController!
        nav.hidesBarsOnSwipe = true
        let bar = nav.navigationBar
        bar.tintColor = UIColor.whiteColor()
        bar.setBackgroundImage(UIColor(named: "backgroundbar"), forBarMetrics: UIBarMetrics.Default)
        let titleFont = UIFont(name: "Verdana-Bold", size: 25.0)
        let color = UIColor.whiteColor()
        let shadow = NSShadow()
        shadow.shadowColor = UIColor.blackColor()
        shadow.shadowOffset = CGSize(width: 2, height: 2)
        let attributes = [NSFontAttributeName: titleFont!, NSForegroundColorAttributeName: color,
        NSShadowAttributeName: shadow]
        bar.titleTextAttributes = attributes
        let item = navigationItem
        let rightButton = UIBarButtonItem(barButtonItemSystemItem: .Play, target: self, action: "printMessage")
        item.rightBarButtonItemItems = [rightButton]
    }
    func printMessage() {
        print("Message")
    }
}
```

The code of Listing 13-13 defines a few properties of the Navigation Bar we didn't use before, such as the `tintColor`, to give a new color to the buttons, and `setBackgroundImage()` to set a custom image for the bar's background (the `backgroundbar.png` image is available on our website). We also assign attributes to the bar's title. This procedure is similar to the one we used to create attributed text for `NSMutableAttributedString` objects in Chapter 4. First, we have to define the values for the attributes, including a `UIFont` object, a color, and an `NSShadow` object, after that we have to create the attributes dictionary, and finally we have to assign this dictionary to the `titleTextAttributes` property of the bar to style the text.

Notice that the buttons assigned to the `leftBarButtonItemItems` or the `rightBarButtonItemItems` arrays

replace those defined in the Storyboard, including the Back button, and therefore the button of a camera we added before is ignored. Figure 13-32 next shows the result, with some of the properties defined by the code of Listing 13-13 and others, like the views' titles or the Back button's title, defined from Interface Builder.



Figure 13-32: *Navigation Bar with a custom style*

Do It Yourself: Erase the button of the camera added before and its Action in the view controller. Update the `ViewController` class with the code of Listing 13-13. Run the application. You should see something similar to Figure 13-32.

IMPORTANT: If a navigation item doesn't exist, the system creates one automatically when the `navigationItem` property is accessed for the first time, so you don't have to add the object from code as you do in the Storyboard.

The Basics: The background image set by the `setBackgroundImage()` method is stretched or shrunk to fit the size of the bar. If the image is too small, the system creates duplicates as many times as necessary to cover the bar.

Toolbar

Besides the Navigation Bar, UIKit includes another class called `UIToolbar` to add an additional bar at the bottom of the view. Figure 13-33 illustrates the option available in the Object Library.

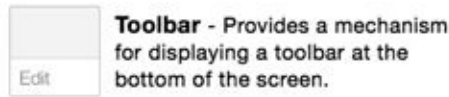


Figure 13-33: *Toolbar option in the Object Library*

Toolbars don't require navigation items but they work with `UIBarButtonItem` objects to represent their buttons. These buttons are added from the Object Library with the option illustrated in Figure 13-29 and are configured from the Utilities Area with the same properties we use for the buttons of a Navigation Bar.

A Toolbar is added from Interface Builder to the main view that requires it. The `UIViewController` class offers the following property to manage the buttons of a Toolbar.

toolbarItems—This property is an array of `UIBarButtonItem` objects that defines the bar buttons.

For configuration, the `UIToolbar` class offers the following properties and methods (some of these values are also available from the Attributes Inspector panel).

barStyle—This property determines the style of the bar. It is an enumeration of type `UIBarStyle` with the values `Default` and `Black`.

barTintColor—This property sets or returns the color applied to the bar's background.

tintColor—This property sets or returns the color applied to the bar's items.

translucent—This property takes a Boolean value that indicates if the bar is translucent or not.

setBackgroundImage(backgroundImage: *image*, forToolbarPosition: *position*, barMetrics: *metrics*)—This method sets the background image for the bar. The ***image*** attribute is a `UIImage` object that represents the image. The ***position*** attribute is used to give a hint to the system about where the bar is located. It is an enumeration of type `UIBarPosition` with the values `Any`, `Bottom`, `Top`, and `TopAttached`. The ***metrics*** attribute is an enumeration of type `UIBarMetrics` that associates the image with the state of the bar. The values available are `Default`, `DefaultPrompt`, `Compact`, and `CompactPrompt`.

To access the Toolbar from our code we have to create an Outlet. The following

example assumes that a Toolbar was added from Interface Builder at the bottom of the initial view and it was connected to the view controller with an Outlet called **myToolbar**.

Listing 13-14: Configuring the Toolbar from the view controller

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var myToolbar: UIToolbar!

    override func viewDidLoad() {
        super.viewDidLoad()

        let nav = navigationController!
        nav.hidesBarsOnSwipe = true

        let bar = nav.navigationBar
        bar.tintColor = UIColor.whiteColor()
        bar.setBackgroundImage(UIColor(named: "backgroundbar"), forBarMetrics: UIBarMetrics.Default)
        let titleFont = UIFont(name: "Verdana-Bold", size: 25.0)
        let color = UIColor.whiteColor()
        let shadow = NSShadow()
        shadow.shadowColor = UIColor.blackColor()
        shadow.shadowOffset = CGSize(width: 2, height: 2)

        let attributes = [NSFontAttributeName: titleFont!, NSForegroundColorAttributeName: color,
        NSShadowAttributeName: shadow]
        bar.titleTextAttributes = attributes

        let item = navigationItem
        let rightButton = UIBarButtonItem(barButtonSystemItem: .Play, target: self, action: "printMessage")
        item.rightBarButtonItemItems = [rightButton]

        myToolbar.setBackgroundImage(UIColor(named: "backgroundbar"), forToolbarPosition: .Bottom,
        barMetrics: .Default)
        myToolbar.tintColor = UIColor.whiteColor()
    }

    func printMessage() {
        print("Message")
    }
}
```

Listing 13-14 assigns a background image to the bar and a color for its buttons. Now both bars, the Navigation Bar at the top and the Toolbar at the bottom, share the same graphic design.

Do It Yourself: Drag a Toolbar from the Object Library to the bottom of the initial view. Add the necessary constraints to pin it to the bottom and the sides of the view.

Control-Drag a line from the bar to the `ViewController` class to create an Outlet called `myToolbar`. Update the class with the code of Listing 13-14. Run the application to see what the bars look like.

IMPORTANT: Unlike Navigation Bars, Toolbars require pin constraints to pin them to the bottom or top of the view. But like Navigation Bars, they are translucent and the rest of the elements can be pinned to the edge of the view and go behind the bar.

The Basics: The `UIToolbar` class also adopts the `UIAppearance` protocol and therefore it implements its `appearance()` method, letting us configure all the bars at once from the app's delegate, as we did in Listing 13-10 for the Navigation Bar.

Custom Navigation

It is possible to hide the Navigation Bar and provide our own navigation tools. The option is available in the Attributes Inspector panel when the Navigation Controller is selected.

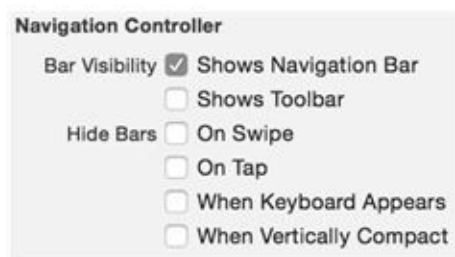


Figure 13-34: Options to show or hide the bars

Clicking on the Shows Navigation Bar option deactivates the bar and removes it from the view controllers in the Storyboard. Once the Navigation Bar is deactivated, it is our responsibility to provide the tools the user needs to navigate. Titles and buttons may be created with the already familiar `UILabel` and `UIButton` classes, but the functionality of navigating back to previous views requires calling some of the `UINavigationController` methods introduced before.

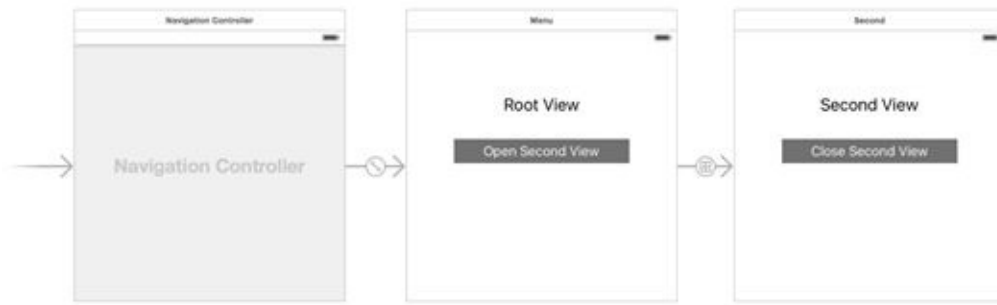


Figure 13-35: *Interface without the Navigation Bar*

The interface of Figure 13-35 is simple. With the Navigation Bar removed we don't have a Back button anymore, so we added a button in the second view called Close Second View to remove it. The class for this view has to include an Action for the button that calls the `popViewControllerAnimated()` method of its Navigation Controller. This method removes the current view and forces the Navigation Controller to show the previous in the stack.

Listing 13-15: *Defining our own method to go back*

```
import UIKit

class SecondViewController: UIViewController {
    @IBAction func saveValue(sender: UIButton) {
        navigationController!.popViewControllerAnimated(true)
    }
}
```

IMPORTANT: Removing or adding a bar affects the constraints already defined inside the view. Some may be stretched to take the new space available and others may be preserved with wrong values or hiding the elements behind the bar.

Orientation

As we studied in Chapter 6, the orientations available for each view controller can be determined by overwriting the `supportedInterfaceOrientations()` method (Listing 6-12). This is possible with independent view controllers but not with view controllers managed by a Navigation Controller because these types of containers control the orientation for themselves and their views. If we need to set different orientations for each view we have to overwrite the `supportedInterfaceOrientations()` method in a subclass of the `UINavigationController` class, get the orientation of the view controller shown on the screen, and return that orientation to assign it to the Navigation Controller. The following is the code for the

Navigation Controller.

Listing 13-16: Getting the orientation from the top view controller

```
import UIKit

class NavigationViewController: UINavigationController {
    override func supportedInterfaceOrientations() -> UIInterfaceOrientationMask {
        return topViewController!.supportedInterfaceOrientations()
    }
}
```

Every time the system detects a rotation in a view managed by the Navigation Controller, it calls the `supportedInterfaceOrientations()` method to check if the new orientation is valid or not. But instead of returning a specific orientation from this method, which would make all the views managed by the Navigation Controller share the same value, the code gets a reference to the view controller currently on display from the `topViewController` property, reads the orientations available for that view, and returns them. With this process, the Navigation Controller is still setting the orientation of its views, but it takes the value from the views themselves. In the view controllers of the views, we have to overwrite the `supportedInterfaceOrientations()` method to determine the orientations allowed to that specific view controller, as we did in the examples of Chapter 6. For instance, we could configure the initial view of our example to work in portrait mode and the second view to work in portrait and landscape. The following is the code for the initial view.

Listing 13-17: Setting the orientations available for the initial view

```
import UIKit

class ViewController: UIViewController {
    override func supportedInterfaceOrientations() -> UIInterfaceOrientationMask {
        let orientation = UIInterfaceOrientationMask.Portrait
        return orientation
    }
}
```

And the following is the view controller class for the second view.

Listing 13-18: Setting the orientations available for the second view

```
import UIKit

class SecondViewController: UIViewController {
```

```

@IBAction func saveValue(sender: UIButton) {
    navigationController!.popViewControllerAnimated(true)
}
override func supportedInterfaceOrientations() -> UIInterfaceOrientationMask {
    let orientation = UIInterfaceOrientationMask.All
    return orientation
}
}

```

Do It Yourself: Create a subclass of the `UINavigationController` class called `NavigationViewController` and assign it to the Navigation Controller of our example (as we explained for the example of Listing 13-8). Update the `NavigationViewController` class, the `ViewController` class, and the `SecondViewController` class with the codes of Listings 13-16, 13-17, and 13-18. Run the application. Rotate the views. The initial view should always be shown in portrait, while the second view should rotate to every possible orientation except upside down (depending on the app's settings, as explained in Figure 6-98).

Sharing Data

Single view controllers connected with Modal Segues usually share information between them and the view controllers they are connected to, but not with the rest of the view controllers in the interface. View controllers managed by a Navigation Controller, on the other hand, fulfill a common purpose and therefore they frequently share common data. When working with Navigation Controllers we shouldn't send information from one view controller to another but instead create a unique source of data. The most frequently used object to store common data is the `AppDelegate` object. The advantage presented by this object is that it is the delegate of the `UIApplication` object created to run the application and therefore it is accessible from anywhere in the code.

As an example, we are going to improve the previous interface to recreate the application developed at the beginning of this chapter. The interface includes two views embedded in a Navigation Controller, one with the buttons to select the picture and the other with an Image View and a Slider to show the picture and rate it.

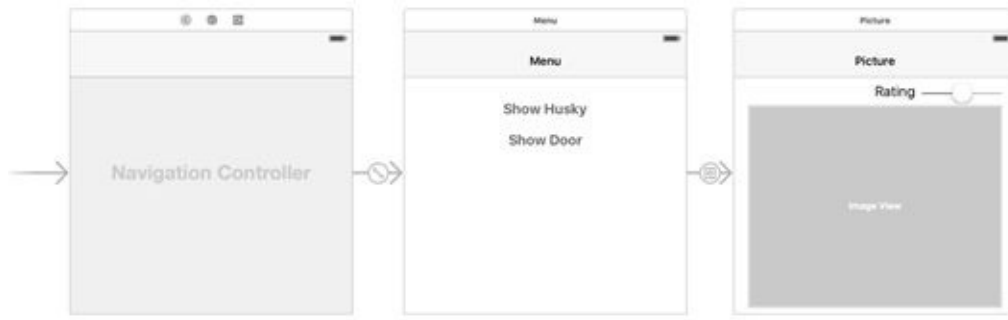


Figure 13-36: *Picture application created with a Navigation Controller*

The application works as before. The system shows the initial view with the menu first and then when the user clicks on any of the buttons it transitions to the second view to show the selected picture. This time we use a Navigation Bar to offer a Back button that lets the user go back to the menu to select another picture.

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add a second view to the interface. Connect the initial Scene to the second view with a Show Segue (Figure 13-18). Assign the string “showPicture” to the Segue’s identifier (Figure 13-12). Create a new file with a view controller called `SecondViewController` and assign it to the second view (Figures 13-6 and 13-7). Add a Navigation Item to the Navigation Bar of the second view (Figure 13-26). Select the Navigation Items and assign the title “Menu” to the initial view and “Picture” to the second view (Figure 13-27). Add buttons called Show Husky and Show Door to the initial view and assign the values 0 and 1 to their `tag` properties (Figure 5-33). Add an Image View, a label with the text “Rating”, and a Slider to the second view. Set the minimum and maximum values for the Slider to 0 and 5. Add the pictures `husky.png` and `door.png` to the Assets Catalog. The interface should look like Figure 13-36.

The code is also similar, but the `picturesList`, `ratings`, and `selectedPicture` properties we used before to store the information are going to be defined in the application’s delegate, so every time the rating is modified in the second view we don’t need to send the data back to the initial view controller, all we have to do is access the app’s delegate and change the values of its properties. The following code illustrates how the properties should be declared and initialized in the delegate.

Listing 13-19: *Defining common data in the app’s delegate*

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```

var window: UIWindow?
var picturesList: [String]!
var ratings: [Int]!
var selectedPicture: Int!

func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject:
AnyObject]?) -> Bool {
    picturesList = ["Husky", "Door"]
    ratings = [0, 0]
    selectedPicture = 0
    return true
}
}

```

The `picturesList` property is an array with the names of the pictures available, the `ratings` property is another array with the rating corresponding to each picture, and the `selectedPicture` property stores the index of the picture selected by the user.

Once defined in the app's delegate, the properties are available everywhere. The first place where they are required is in the view controller of the initial view. Here we have to update the `selectedPicture` property every time a button is pressed to communicate to the rest of the code what picture was selected.

Listing 13-20: Storing the index of the selected picture

```

import UIKit

class ViewController: UIViewController {
    var mydelegate: AppDelegate!
    override func viewDidLoad() {
        super.viewDidLoad()
        let app = UIApplication.sharedApplication()
        mydelegate = app.delegate as! AppDelegate
    }
    @IBAction func goToPicture(sender: UIButton) {
        mydelegate.selectedPicture = sender.tag
        performSegueWithIdentifier("showPicture", sender: self)
    }
}

```

The first thing we do in the `ViewController` class of Listing 13-20 is to get a reference to the application's delegate. The process is the same introduced in Chapter 5 (Listing 5-2). We have to get a reference to the `UIApplication` object created for the application and then

read its **delegate** property. The property we need to modify in the delegate is the **selectedPicture** property that stores the index of the selected picture. For this we have connected both buttons, Show Husky and Show Door, to one Action called **goToPicture()**. When a button is pressed, we take the value of its **tag** property and assign it to the **selectedPicture** property. At the end we call the **performSegueWithIdentifier()** method to transition to the second view. Notice that this time we don't need to send any value to **SecondViewController** because this view controller also takes the values from the app's delegate, as shown next.

Listing 13-21: Storing the rating in the app's delegate

```
import UIKit

class SecondViewController: UIViewController {
    @IBOutlet weak var sliderRating: UISlider!
    @IBOutlet weak var pictureView: UIImageView!
    var mydelegate: AppDelegate!

    override func viewDidLoad() {
        super.viewDidLoad()

        let app = UIApplication.sharedApplication()
        mydelegate = app.delegate as! AppDelegate
        let selected = mydelegate.selectedPicture
        let picture = mydelegate.picturesList[selected]
        let rating = mydelegate.ratings[selected]
        sliderRating.value = Float(rating)
        pictureView.image = UIImage(named: picture.lowercaseString)
    }

    @IBAction func changeRating(sender: UISlider) {
        let value = round(sender.value)
        sliderRating.value = value
        let selected = mydelegate.selectedPicture
        mydelegate.ratings[selected] = Int(value)
    }
}
```

In the **SecondViewController** class we have to read the three properties of the app's delegate to configure the view. We get the index of the selected picture first and then use it to read the picture's name and rating from the **picturesList** and **ratings** arrays. This information is used next to update the Slider and the Image View before the interface is shown on the screen. The **selectedPicture** and **ratings** properties are also used in the Action of the Slider to update the value of the rating with the one selected by the user.

Do It Yourself: Modify the `AppDelegate` class in the `AppDelegate.swift` file with the code of Listing 13-19. Connect the Show Husky and Show Door buttons of the initial view to an Action called `goToPicture()`. Complete the `ViewController` class with the code of Listing 13-20. Connect the Slider and the Image View of the second view to Outlets called `sliderRating` and `pictureView`, respectively. Create an Action for the Slider called `changeRating()`. Complete the `SecondViewController` class with the code of Listing 13-21. Run the application and modify the rating of the pictures to verify that the values are preserved.

The use of the app's delegate to store data is only recommended when the amount of data is not significant. For large amounts of data it is better to create our own objects or structures that the application can access every time necessary and where we can safely manage the information available. To do this in Swift is very easy. All the files are considered to be in the global space and are executed as soon as the application is launched. If we want to create a global object or structure that we can access everywhere in our code, all we have to do is define it and initialize it in a new Swift file.

As any other files, Swift files are created from the File menu or pressing Command + N on the keyboard, but instead of selecting Cocoa Touch Class from the options we have to click on the Swift File option (Figure 13-3). The name of the file in this case is just for reference, because except for a few comments and `import` statements, no code is created for the template. Once the file is added to the list of files in our application we can define our custom classes or structures inside. For our example, we have created a file called `ApplicationData.swift` and define a structure inside with the same name and all the properties necessary in our application.

Listing 13-22: Defining a global structure to store common data

```
struct ApplicationData {  
    var picturesList: [String]  
    var ratings: [Int]  
    var selectedPicture: Int  
  
    init() {  
        picturesList = ["Husky", "Door"]  
        ratings = [0, 0]  
        selectedPicture = 0  
    }  
}  
  
var AppData = ApplicationData()
```

The Basics: The `ApplicationData` structure represents the model in our MVC (see Chapter 5). MVC is an architectural paradigm that classifies the parts of a program as

Model (data), View (graphics), and Controller (processing code). In our example, the views are created from the Storyboard, the controllers are the view controllers we associate with those views, and the model is now the **AppData** structure defined in Listing 13-22.

The structure declared in Listing 13-22 contains the same information as before, but it provides a unique place to store and manage all the information for our app. Now we can access this data from everywhere in the code by just reading and writing the content of the **AppData** variable. The following example shows how the **ViewController** of our application gets the data from **AppData**.

Listing 13-23: *Accessing the global structure from the `ViewController` class*

```
import UIKit

class ViewController: UIViewController {
    @IBAction func goToPicture(sender: UIButton) {
        AppData.selectedPicture = sender.tag
        performSegueWithIdentifier("showPicture", sender: self)
    }
}
```

Since we don't have to get a reference to the app's delegate anymore, all we need in the **ViewController** class is the Action for the buttons. When a button is pressed, the **goToPicture()** method is executed as always, but this time instead of accessing the **selectedPicture** property from the app's delegate we do it from the **AppData** structure.

The code for the **SecondViewController** was also simplified. Getting the data from a global structure makes everything easier.

Listing 13-24: *Accessing the global structure from the `SecondViewController` class*

```
import UIKit

class SecondViewController: UIViewController {
    @IBOutlet weak var sliderRating: UISlider!
    @IBOutlet weak var pictureView: UIImageView!
    override func viewDidLoad() {
        super.viewDidLoad()

        let selected = AppData.selectedPicture
        let picture = AppData.picturesList[selected]
        let rating = AppData.ratings[selected]
    }
}
```

```

sliderRating.value = Float(rating)
pictureView.image = UIImage(named: picture.lowercaseString)
}
@IBAction func changeRating(sender: UISlider) {
    let value = round(sender.value)
    sliderRating.value = value
    let selected = AppData.selectedPicture
    AppData.ratings[selected] = Int(value)
}
}

```

Do It Yourself: Erase the properties in the **AppDelegate** class. Open the File menu, click on New, and select the option File to create a new Swift file. In the emergent window click on the Swift File icon and insert the name **ApplicationData**. Open the new file and replace its code by the code of Listing 13-22. Replace the **ViewController** class by the class of Listing 13-23. Replace the **SecondViewController** class by the class of Listing 13-24. Run the application. It should work as before but now the data is taken from the **AppData** structure.

IMPORTANT: The models used for the examples of this book implement a very simple programming pattern. This pattern is suitable for small applications, but there are multiple patterns available and possible configurations that professional applications may require. There are several alternatives available and we will see some examples later (see Chapter 21, Listing 21-22), but the topic is beyond the scope of this book. For more information on programming patterns and data management, visit our website and follow the links for this chapter.

13.3 Tab Bar Controllers

A Tab Bar Controller is a simple container view controller that designates two areas on the screen: one for the views and a smaller one at the bottom of the screen for a bar with tabs that users can tap to select the view they want to see. Each tab is associated with only one view and therefore we can use the tabs to move from one view to another.



Figure 13-37: Application based on a Tab Bar Controller

As with Navigation Controllers, there are two ways to add a Tab Bar Controller to the Storyboard. We can embed a view inside a Tab Bar Controller from the Editor menu (Editor/Embed/Tab Bar Controller) or drag the Tab Bar Controller option from the Object Library to the Storyboard. Figure 13-38 shows what we see when we embed a single view in a Tab Bar Controller.

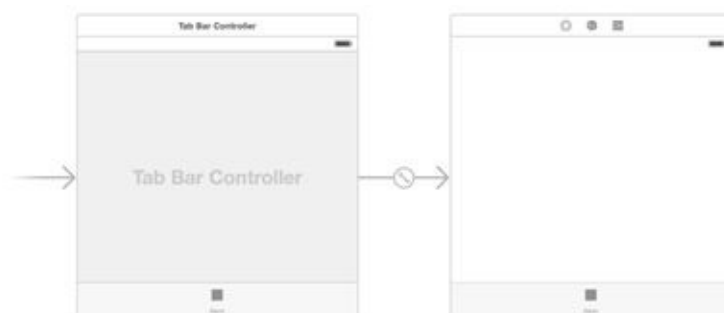


Figure 13-38: Initial view embedded in a Tab Bar Controller

The views managed by the Tab Bar Controller are connected with a Segue called *View Controllers*. If we want to add another view, we have to drag the View Controller option from the Object Library and create a View Controllers Segue from the Tab Bar Controller to the new view.

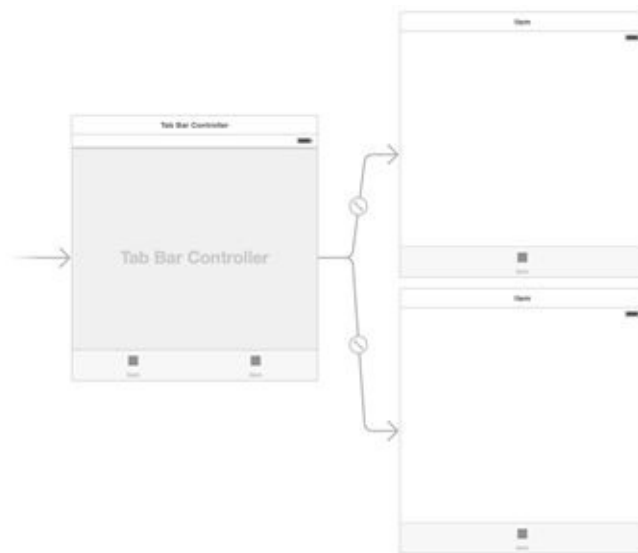


Figure 13-39: Multiple views managed by the Tab Bar Controller

We can add all the views we want. When a view is added to the Tab Bar Controller, the system automatically includes the corresponding tab at the bottom of the view. Then the Tab Bar Controller gets the tabs from the view controllers and generates the bar. If there is not enough space on the bar to place all the tabs available, the controller adds a tab called **More** that lets the user select the tabs that are not visible.

Tabs

The bar is managed by the Tab Bar Controller, but the tabs are provided by the view controller they belong to. To change the values and appearance of the tabs we have to select them from the view by clicking on them and edit their values from the Attributes Inspector panel.

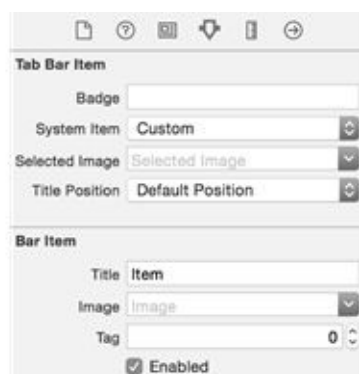


Figure 13-40: Options to configure the tab

Each tab has a title, selected image, unselected image, and a badge value to identify the view they represent. There are two alternatives to configure the tab: we can provide a custom name and images, or we can select a tab predefined by the system from the option called *System Item*. If we click on this field, a popup window lists all the tabs available. Figure 13-41, next, shows what our Tab Bar looks like after the style Favorites and Search were selected for the views of our interface.

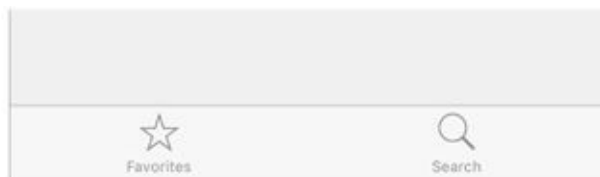


Figure 13-41: *Tab Bar with tabs defined by the system*

These options define the image and the name. If we want to provide our own name, we have to customize the image as well. The image must be provided with a size of 30 pixels by 30 pixels (60x60 for 2x scale and 90x90 for 3x scale) and a transparent background. No colors are necessary because the system uses the image to create a mask and present it with the colors defined by the bar. For our example we have created the images *iconweather.png* for the initial view and *iconsettings.png* for the second view. Figure 13-42 shows what the bar looks like after we modify each tab and give them the names “Weather” and “Settings”, respectively.

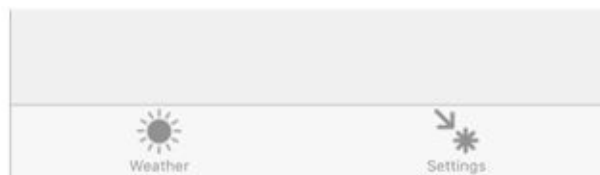


Figure 13-42: *Tab bar with custom tabs*

Do It Yourself: Create a Single View Application project and embed the initial view in a Tab Bar Controller (Editor/Embed In/Tab Bar Controller). Add a second view and create a Segue of type View Controllers from the Tab Bar Controller to the new view (Figure 13-39). Download from our website the files for the images *iconweather.png* and *iconsettings.png* and add them to the Assets Catalog. Click on the initial view’s tab (you can also select it from the Document Outline panel), go to the Bar Item section in the Attributes Inspector panel, and insert the name “Weather” and the image *iconweather.png*. Repeat the same process with the name “Settings” and the image *iconsettings.png* for the second view’s tab. The bar of the Tab Bar Controller should look like Figure 13-42.

Tabs are created from the `UITabBarItem` class. The class provides the following initializers to create a new tab.

`UITabBarItem(tabBarItemSystemItem: type, tag: tag)`—This initializer creates a tab of a type predefined by the system. The **`type`** attribute is an enumeration of type `UITabBarItemSystemItem` that determines the type of tab to create. The possible values are `More`, `Favorites`, `Featured`, `TopRated`, `Recents`, `Contacts`, `History`, `Bookmarks`, `Search`, `Downloads`, `MostRecent`, and `MostViewed`. The **`tag`** attribute is an integer to identify the tab.

`UITabBarItem(title: title, image: image, tag: tag)`—This initializer creates a custom tab with the title specified by the **`title`** attribute and the image specified by the **`image`** attribute. The image for the selected state is automatically generated by the system. The **`tag`** attribute is an integer to identify the tab.

`UITabBarItem(title: title, image: image, selectedImage: selected)`—This initializer creates a custom tab with the title specified by the **`title`** attribute and the images for the normal and selected states specified by the attributes **`image`** and **`selected`** (`UIImage` objects). The **`tag`** attribute is an integer to identify the tab.

The `UITabBarItem` is a subclass of the `UIBarButtonItem` class; a class also used to create the buttons for Navigation Controllers (see Listing 13-12). Together, these classes provide the following properties to configure the tab.

`title`—This property sets or returns the tab's title. It is an optional of type `String`.

`image`—This property sets or returns the image for the tab's normal state. It is an optional of type `UIImage`.

`selectedImage`—This property sets or returns the image for the tab's selected state. It is an optional of type `UIImage`.

`tag`—This property sets or returns the tab's tag value. It is of type `Int`.

`badgeValue`—This property sets or returns the value of the tab's badge (shown inside a red circle at the top-right corner of the icon). It is of type `String`.

The tabs are configured and replaced from the view controllers they belong to. The `UIViewController` class includes the `tabBarItem` property to access the view controller's tab. The next example adds an Action for a button to update the badge when the button is pressed.

Listing 13-25: Updating the badge from the view controller

```
import UIKit

class ViewController: UIViewController {
    var counter = 1
    @IBAction func updateBadge(sender: UIButton) {
        let item = tabBarItem
        item.badgeValue = String(counter)
        counter++
    }
}
```

The code for the **ViewController** presented in Listing 13-25 gets a reference to the tab from the **tabBarItem** property and then updates its **badgeValue** property with the value of the **counter** property. The counter is incremented every time the button is pressed to illustrate how the badge is modified by the system. Figure 13-43 shows what we see after we run the application and press the button 12 times.



Figure 13-43: *Tab with a badge*

Do It Yourself: Add a button to the initial view of the interface of Figure 13-39. Connect the button to an Action called **updateBadge()**. Complete the **ViewController** class with the code of Listing 13-25. Run the application and press the button. The badge should appear over the tab as shown in Figure 13-43.

UITabBarController

View controllers are in charge of their own tabs, but the bar and the list of view controllers available is managed by the Tab Bar Controller. The **UITabBarController** class includes the following properties to store the view controllers, keep a reference to the view controller currently selected, and access the bar.

tabBar—This property returns a reference to the **UITabBar** object that represents the bar.

viewControllers—This property sets or returns the view controllers managed by the Tab Bar Controller.

selectedViewController—This property returns a reference to the view controller of the view that is currently being shown on the screen.

selectedIndex—This property returns the index of the view controller of the view that is currently being shown on the screen. It is a value of type `Int` that represents an index of the `viewControllers` array.

The bar is created from the `UITabBar` class and offers the following properties for configuration.

barStyle—This property determines the style of the bar. It is an enumeration of type `UIBarStyle` with the values `Default` and `Black`.

barTintColor—This property sets or returns the color applied to the bar's background.

tintColor—This property sets or returns the color applied to the bar's items.

translucent—This property takes a Boolean value that indicates if the bar is translucent or not.

backgroundImage—This property sets the background image for the bar. Its value is an optional of type `UIImage`.

selectionIndicatorImage—This property sets the background image for the selected tab. Its value is an optional of type `UIImage`.

itemPositioning—This property sets the positioning scheme for the tabs. It is an enumeration of type `UITabBarItemPositioning` with the values `Automatic` (the tabs are positioned according to the device), `Fill` (the tabs are positioned to fill the width of the bar), and `Centered` (the tabs are positioned at the center of the bar).

itemSpacing—This property sets the space between tabs when the `backPositioning` property is set to `Centered`. It is a value of type `CGFloat`.

itemWidth—This property sets the width of the tabs when the `backPositioning` property is set to `Centered`. It is a value of type `CGFloat`.

Although the `UITabBar` class also adopts the `UIAppearance` protocol and implements its `appearance()` method to let us configure the bar by modifying the class, it is better to create a subclass of `UITabBarController` and manage the view controllers and the bar from it. For our example we have created a file called `TabBarController.swift` with a subclass of the

`UITabBarController` and assign it to the Tab Bar Controller of the interface of Figure 13-39.

Listing 13-26: *Configuring the bar from a subclass of `UITabBarController`*

```
import UIKit

class TabBarController: UITabBarController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let bar = tabBar
        bar.tintColor = UIColor.whiteColor()
        bar.translucent = true
        bar.backgroundImage = UIImage(named: "tabbar")
    }
}
```

The process is similar to those implemented to configure the bar for Navigation Controllers. In this case a reference to the bar is obtained from the `tabBar` property provided by the `UITabBarController` class and then the values of its properties are modified. The only difference is in how the bar presents the background image. A Tab Bar duplicates the image as many times as necessary to fill the bar's width, but it doesn't stretch it or shrink it, so we have to provide one with the right size. The image could be any width we want, but it has to be 49 pixels tall (98x98 pixels for the 2x scale and 147x147 pixels for the 3x scale). For this example we used an image of 49 x 49 points called `tabbar.png`.

Do It Yourself: Download the `tabbar.png` image from our website and add it to your Assets Catalog. Create a new file called `TabBarController.swift` with a subclass of the `UITabBarController` class and assign it to the Tab Bar Controller of Figure 13-39. Complete the subclass with the code of Listing 13-26. The application now shows the bar with a background image.

IMPORTANT: The `UIViewController` class includes a property called `tabBarController` to get a reference to the Tab Bar Controller from its view controllers. It is not recommendable to configure the bar from any of the view controllers because there is no guarantee that the code is going to be processed as soon as the app is executed, but the property may be useful sometimes to get information from the Tab Bar Controller.

There are more tasks we can perform from the `UITabBarController` subclass other than configuring the bar. Important things like establishing which view will be shown first or

setting the badges for each tab can be done from this subclass. The following example modifies the `selectedIndex` property to declare the second view as the initial view and adds a badge to the Weather's tab.

Listing 13-27: Initializing the Tab Bar Controller

```
import UIKit

class TabBarController: UITabBarController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let bar = tabBar
        bar.tintColor = UIColor.whiteColor()
        bar.translucent = true
        bar.backgroundImage = UIImage(named: "tabbar")
        selectedIndex = 1
        let list = viewControllers!
        let controller = list[0] as! ViewController
        let tab = controller.tabBarItem
        tab.badgeValue = String(20)
    }
}
```

The view controllers managed by the Tab Bar Controller are stored in an array called `viewControllers` in the order they were created on the Storyboard. In our example, the view controller for the Weather view was stored at index 0 and the view controller for the Settings view was stored at index 1. This is why we have to assign a value 1 to the `selectedIndex` property to declare the Settings view as the initial view. On the other hand, the view controller of the Weather view is at index 0, and therefore to add a badge to that view's tab we have to get the first element of the `viewControllers` array, cast it as `ViewController` (the array contains elements of type `UIViewController`), and access its properties as we did before in Listing 13-26.

The Basics: Because Tab Bar Controllers are often implemented as the starting point of the application, Xcode offers a template called *Tabbed Application* to create a project that initializes the Storyboard with a Tab Bar Controller and two views.

Delegate

The Tab Bar Controller can use a delegate object to report when something happened or is about to happen with its tabs. The UIKit framework includes the `UITabBarControllerDelegate` protocol for this purpose. The following are some of the methods available in this protocol.

`tabBarController(tabBarController, didSelectViewController: viewController)`—This method is called by the `UITabBarController` object when a tab is selected. The **`tabBarController`** attribute is a reference to the `UITabBarController` object, and the **`viewController`** attribute is a reference to the view controller of the selected tab.

`tabBarController(tabBarController, shouldSelectViewController: viewController)` —This method is called by the `UITabBarController` object to know whether or not to let the user select a tab. The method returns a Boolean value that indicates the decision made by the code. The **`tabBarController`** attribute is a reference to the `UITabBarController` object, and the **`viewController`** attribute is a reference to the view controller of the selected tab.

The delegate object may be an external object or the `UITabBarController` subclass that is controlling our Tab Bar Controller. The following example implements the last option to perform a task when a tab is selected.

Listing 13-28: Defining the Tab Bar Controller’s delegate

```
import UIKit
```

```
class TabBarController: UITabBarController, UITabBarControllerDelegate {  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        delegate = self  
    }  
  
    func tabBarController(tabBarController: UITabBarController, didSelectViewController viewController: UIViewController) {  
        let list = viewControllers!  
        let controller = list[1]  
        if viewController === controller {  
            print("It's Settings")  
        }  
    }  
}
```

To turn the `TabBarController` class of our example into its own delegate, we have to

declare that the class conforms to the `UITabBarControllerDelegate` protocol first, and then assign `self` to its `delegate` property. From that moment on, the `TabBarController` object will call the delegate's methods on itself. In Listing 13-28 we declare the method that is called when the user selects a tab. To check which tab was selected we compare the value on its `viewController` attribute against the reference stored at index 1 in the `viewControllers` array of the `TabBarController` object. This reference corresponds to the `SecondViewController` object and therefore the task is performed when the Settings view is shown on the screen.

There are other ways to detect the type of the view controller selected. For example, we could try to convert it to one of our `UIViewController` subclasses and compare the value returned by the conversion against `nil`, as in `viewController as? SecondViewController != nil`. If the view controller received by the method is of type `SecondViewController`, the condition is true and we can perform the task as we did in Listing 13-28. But we could also get the object from the conversion and read its properties, as shown in the following example.

Listing 13-29: Allowing the selection of a tab

import UIKit

```
class TabBarController: UITabBarController, UITabBarControllerDelegate {
    override func viewDidLoad() {
        super.viewDidLoad()
        delegate = self
    }

    func tabBarController(tabBarController: UITabBarController, shouldSelectViewController viewController:
    UIViewController) -> Bool {
        if let controller = viewController as? SecondViewController {
            let control = controller.myproperty
            if control != 0 {
                return false
            }
        }
        return true
    }
}
```

This time the protocol method implemented is called when a tab is selected but before its view is shown on the screen. We have to return a Boolean value from the method to indicate whether or not the action is valid. To know what value to return, the code of Listing 13-29 detects the tab the user is trying to open by comparing the value received from the `viewController` attribute against the class corresponding to the view controller of the Settings view (`SecondViewController`). In case of success, it stores the result in the `controller` constant and uses this reference to access a property called `myproperty`. If the value of this

property is different than 0, the code returns **false** and doesn't let the user select the tab.

Do It Yourself: Copy the example you want to try in the `TabBarController.swift` file created before and run the application. The code of Listing 13-29 assumes that you have already declared a property called `myproperty` in the `SecondViewController` class and assign an integer value to it. If the value is different than 0, the delegate method in this example returns **false** and therefore you won't be able to select the Settings tab.

Real-Life Application

For a better understanding of Tab Bar Controllers and the kind of applications they are suitable for, we are going to finish the Weather application started in previous examples. We need the interface of Figure 13-39 and the tabs of Figure 13-42. The views have to be associated with their respective view controllers (`ViewController` class for the Weather view and `SecondViewController` class for the Settings view), and we will also need the `TabBarController` class for the Tab Bar Controller.

The app shows the information about the weather in the initial view (Weather) and lets the user select the city and the unit of measurement in the second view (Settings). To create the interface for the Weather view, we have to add two big labels at the top for the name of the city and the value of the temperature, and two empty views with a gray background, as shown in the first picture of Figure 13-44 below. The elements have to be pinned to the main view with Space constraints and then the rest of the labels are added inside the gray views. We have to create two labels per value, one for the name of the value and another for the value itself. The values we are going to present inside the gray views are Precipitation, Humidity, Wind, Pressure, Visibility, and Feels Like.



Figure 13-44: Interface design for the Weather view

To make the app look more professional, we added an Image View as background (the file for the image is called `clouds.jpg` and is available on our website). We can add this image first and then the rest of the interface. To create the constraints, we can use the

Document Outline panel, as explained in Chapter 6, Figure 6-56.

The interface for the Settings view is similar. We have to add two gray views, pin them to the sides of the main view, and then incorporate a label and a Segmented Control with two buttons to the top view, and a picker to the bottom view (center of Figure 13-45). The buttons of the Segmented Control were called Celsius and Fahrenheit to let the user select the unit of measurement. This view was also improved with a background image. The file for the image is called lake.png and is available on our website.



Figure 13-45: Interface design for the Settings view

The next step is to create the model that contains the data that is going to be processed by the app. Because the data is significant, we store it in a separate file, as we did for previous examples.

Listing 13-30: Preparing the data for the app

```
struct WeatherData {
    var temperature: Int
    var precipitation: Double
    var humidity: Int
    var wind: Double
    var pressure: Double
    var visibility: Double
    var feels: Int
}

struct ApplicationData {
    var citiesName: [String] = []
    var citiesData: [WeatherData] = []
    var selectedCity: Int
    var celsius: Bool

    init() {
        var cities: [String: [Double]] = [:]
        cities["Toronto"] = [21, 0.0, 83, 0.0, 1.016, 14.5, 24]
        cities["New York"] = [18, 3.0, 95, 12.4, 1.020, 8.5, 15]
```

```

cities["Paris"] = [24, 8.7, 90, 5.4, 1.055, 10.5, 25]
for (city, data) in cities {
    citiesName.append(city)

    let newData = WeatherData(temperature: Int(data[0]), precipitation: data[1], humidity: Int(data[2]), wind: data[3],
    pressure: data[4], visibility: data[5], feels: Int(data[6]))
    citiesData.append(newData)
}
selectedCity = 0
celsius = true
}
}

var AppData = ApplicationData()

```

Two structures were defined in Listing 13-30. The **WeatherData** structure stores the data for every city and the **ApplicationData** structure manages the data and the settings for the app. There are four properties in the **ApplicationData** structure: **citiesName**, **citiesData**, **selectedCity**, and **celsius**. The **citiesName** property is an array of strings to store the names of the cities, the **citiesData** property is an array of **WeatherData** structures to store the data for every city, the **selectedCity** property is used to store the index of the currently selected city, and the **celsius** property is a Boolean value that indicates whether the temperature is going to be shown in Celsius or Fahrenheit. When the **ApplicationData** structure is initialized, the information for every city is stored inside a dictionary and then the dictionary is used to initialize the properties. This dictionary has a **String** value for key and an array of **Double** for values, as the dictionaries studied in Chapter 2 (see Listing 2-57 and 2-58).

The Basics: In a real application this information would be loaded from a server or a file. We will learn how to incorporate data to the app from external sources in further chapters.

Now we have to show the information on the Weather view. For this purpose, the **ViewController** class have to include Outlets for every label.

Listing 13-31: View controller for the Weather view

```

import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var cityLabel: UILabel!
    @IBOutlet weak var temperatureLabel: UILabel!
    @IBOutlet weak var precipitationLabel: UILabel!
    @IBOutlet weak var humidityLabel: UILabel!
}

```



```

@IBOutlet weak var windLabel: UILabel!
@IBOutlet weak var pressureLabel: UILabel!
@IBOutlet weak var visibilityLabel: UILabel!
@IBOutlet weak var feelsLabel: UILabel!
override func viewWillAppear(animated: Bool) {
    let selected = AppData.selectedCity
    let cityData = AppData.citiesData[selected]
    var temperature = cityData.temperature
    var feels = cityData.feels
    if !AppData.celsius {
        temperature = temperature * 9/5 + 32
        feels = feels * 9/5 + 32
    }
    temperatureLabel.text = String(temperature)
    feelsLabel.text = "\\(feels) degrees"
    cityLabel.text = AppData.citiesName[selected]
    precipitationLabel.text = "\\(cityData.precipitation) cm"
    humidityLabel.text = "\\(cityData.humidity) %"
    windLabel.text = "\\(cityData.wind) Km/h"
    pressureLabel.text = "\\(cityData.pressure) mb"
    visibilityLabel.text = "\\(cityData.visibility) Km"
}
}

```

The labels connected to the view controllers are those that present the actual data (not those that describe the data). Using these references, we get the data from the **AppData** structure and present their values on the screen as soon as the view is loaded.

IMPORTANT: Tab Bar Controllers load the views once and then keep them in memory. This means that the **viewDidLoad()** method will be executed only once, and any change performed by the user in another view won't be reflected on the screen later. That is why this time we used the **viewWillAppear()** method instead. The **viewDidLoad()** method is executed once after the view is loaded in memory, but the **viewWillAppear()** method is executed every time the view is about to be shown on the screen.

The Basics: The code of Listing 13-31 calculates the temperature in Fahrenheit from Celsius using a conversion formula. The formula establishes that Fahrenheit is equal to $C \times 9/5 + 32$, and Celsius is equal to $(F - 32) \times 5/9$.

The Settings view requires a little work but it is also a simple code. We need an Action for the Segmented Control to change the unit of measurement when a button is pressed and the delegate and data source methods to configure the picker and get the value selected.

Listing 13-32: View controller for the Settings view

```
import UIKit

class SecondViewController: UIViewController, UIPickerViewDelegate, UIPickerViewDataSource {
    @IBOutlet weak var selectTemperature: UISegmentedControl!
    @IBOutlet weak var picker: UIPickerView!

    override func viewDidLoad() {
        super.viewDidLoad()
        picker.delegate = self
        picker.dataSource = self
        picker.selectRow(AppData.selectedCity, inComponent: 0, animated: false)
        if AppData.celsius {
            selectTemperature.selectedSegmentIndex = 0
        } else {
            selectTemperature.selectedSegmentIndex = 1
        }
    }

    @IBAction func changeTemperature(sender: UISegmentedControl) {
        if sender.selectedSegmentIndex == 0 {
            AppData.celsius = true
        } else {
            AppData.celsius = false
        }
    }

    func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int {
        return 1
    }

    func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        return AppData.citiesName.count
    }

    func pickerView(pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {
        return AppData.citiesName[row]
    }

    func pickerView(pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) {
        AppData.selectedCity = row
    }
}
```

}

The code of Listing 13-32 sets the initial conditions when the view is loaded and then changes those conditions in **AppData** responding to the user interaction. Because the data is stored in a global structure and we used the **viewWillAppear()** method to update the information on the Weather view, the changes performed here are immediately reflected on the screen.

IMPORTANT: Navigation Controllers and Tab Bar Controllers are powerful container controllers but are not the only ones available. In further chapters we will study more specific container controllers and how to combine every tool studied so far to create complex interfaces for iPhones and iPads.

Chapter 14

Table Views

14.1 Tables

One of the basic characteristics of computers is their capacity to process sequential data. Due to their elemental structure, conformed by a sequence of switches on and off, computers are experts at organizing information as lists of values. It didn't take long for computer systems to adopt the concepts of tables to present this type of information. A table is an organizational system that proposes the arrangement of data in rows and columns. Apple has always provided the tools to create tables on its systems, but because of the small screens in mobile devices, it was compelled to introduce a new version of tables composed by only one column. Despite their simplicity, tables for the iOS system are very powerful. They provide tools to manage the data, organize it in single lists or multiple sections, and are built inside a Scroll View, allowing content to be shown with no limitations whatsoever. Figure 14-1 shows the two basic configurations. In the table on the left, the items are presented in a single list. On the right, the items are split in sections.

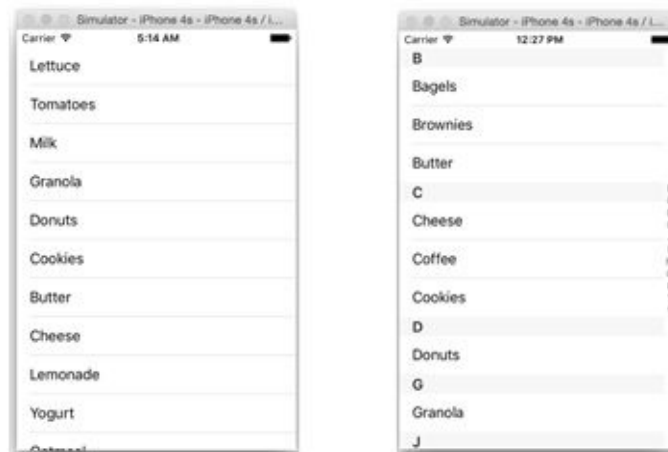


Figure 14-1: Table Views with Plain style

iOS tables may also adopt two styles: Plain and Grouped. There is not much difference between one style and another except for their graphic design and the kind of information they are meant to present. Plain tables were developed to display single lists with information of the same type, while the purpose of Grouped tables is to group information of different types. The tables in Figure 14-1 are both Plain, one with a single list and the other with sections to organize the items alphabetically. Figure 14-2, next, shows the same tables but this time with a Grouped style (notice that the version with sections separates the items in categories instead of alphabetically).

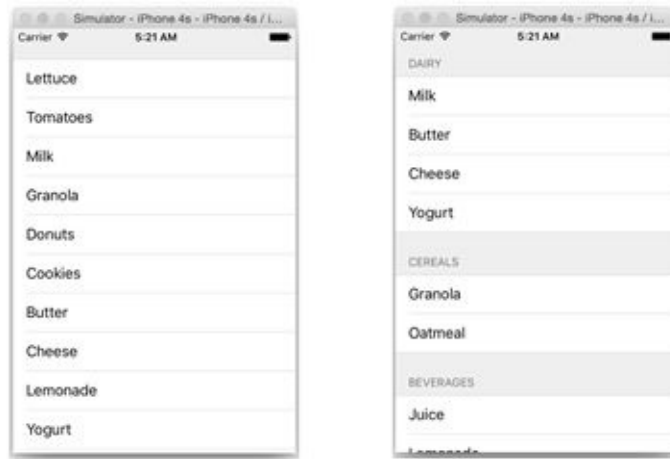


Figure 14-2: Table Views with Grouped style

UITableView

Tables are created from the `UITableView` class. The class includes the following properties and methods to create and configure the table.

allowsSelection—This property sets or returns a Boolean value that determines if the user can select a cell or not.

allowsMultipleSelection—This property sets or returns a Boolean value that determines if the user can select multiple cells or not.

editing—This property sets or returns a Boolean value to indicate whether the table is in edition mode or not. The edition mode is used to insert, delete, or move rows.

rowHeight—This property sets or returns a `CGFloat` value that determines the cells' height.

estimatedRowHeight—This property sets or returns a `CGFloat` value that determines the approximate size of the cell. It has the purpose to improve performance and it is required when working with variable cell heights.

separatorStyle—This property sets or returns the style for the cells' separators. It is an enumeration of type `UITableViewCellStyle` with the values `None`, `SingleLine`, and `SingleLineEtched`.

separatorColor—This property sets or returns the color for the cells' separators. It is an implicitly unwrapped optional of type `UIColor`.

separatorInset—This property sets or returns the padding for the cells' separators. It is a structure of type `UIEdgeInsets` with the properties `top`, `left`, `bottom`, and `right` (only the left and right values are considered).

tableHeaderView—This property sets or returns the view that represents the table’s header. The value by default is **nil**.

tableFooterView—This property sets or returns the view that represents the table’s footer. The value by default is **nil**.

sectionIndexColor—This property sets or returns the color for the Table View’s index text. It is an optional of type **UIColor**.

sectionIndexBackgroundColor—This property sets or returns the color for the background of the Table View’s index when it is not being touched. It is an optional of type **UIColor**.

sectionIndexTrackingBackgroundColor—This property sets or returns the color for the background of the Table View’s index when the user touches it. It is an optional of type **UIColor**.

indexPathForSelectedRow—This property returns an **NSIndexPath** object with the location of the currently selected cell.

dequeueReusableCellWithIdentifier(identifier, forIndexPath: index)—This method returns a reusable cell with the specified identifier. The **identifier** attribute is the string that identifies the cell and the **index** attribute is an **NSIndexPath** object with the location of the cell.

cellForRowAtIndexPath(index)—This method returns a reference to the **UITableViewCell** object of the cell at the index path specified by the **index** attribute. The attribute is an **NSIndexPath** object.

indexPathForCell(cell)—This method returns an **NSIndexPath** object with the location of a cell. The attribute is a reference to the **UITableViewCell** object that represents the cell.

selectRowAtIndexPath(index, animated: animated, scrollPosition: position)—This method selects the cell at the location specified by the **index** attribute. The **index** attribute is an **NSIndexPath** object with the information of the cell’s location. The **animated** attribute is a Boolean value that indicates whether the process is going to be animated or not. The **position** attribute indicates if the table will scroll to the position of the cell and how. It is an enumeration of type **UITableViewScrollPosition** with the values **None**, **Top**, **Middle**, and **Bottom**.

deselectRowAtIndexPath(index, animated: animated)—This method deselects the cell at the location specified by the **index** attribute. The **index** attribute is an **NSIndexPath** object with the location of the cell, and the **animated** attribute is a Boolean value that indicates whether the process is going to be animated or not.

scrollToRowAtIndexPath(index, atScrollPosition: position, animated:

animated)—This method scrolls the Table View to show on the screen the cell at the position indicated by the **index** attribute. The **index** attribute is an `NSIndexPath` object with the location of the cell. The **position** attribute indicates if the table will scroll to the position of the cell and how. It is an enumeration of type `UITableViewScrollPosition` with the values `None`, `Top`, `Middle`, and `Bottom`. The **animated** attribute is a Boolean value that indicates whether the process is going to be animated or not.

setEditing(value, animated: animated)—This method sets the value of the editing property. The **value** attribute is the Boolean value we want to assign to the property and the **animated** attribute is a Boolean value that determines if the change will be animated.

reloadData()—This method forces the Table View to reload the cells and sections.

beginUpdates()—This method indicates that an updating process has begun. It is used to animate the insertion or deletion of multiple cells.

endUpdates()—This method indicates that an updating process has ended. It is used to animate the insertion or deletion of multiple cells.

insertRowsAtIndexPaths(paths, withRowAnimation: animation)—This method inserts new cells at the location indicated by the **paths** attribute. The **paths** attribute is an array of `NSIndexPath` objects with the locations of every new cell. The **animation** attribute indicates how the insertion is going to be animated. It is an enumeration of type `UITableViewRowAnimation` with the values `Fade`, `Right`, `Left`, `Top`, `Bottom`, `None`, `Middle`, and `Automatic`.

deleteRowAtIndexPaths(paths, withRowAnimation: animation)—This method deletes the cells at the location indicated by the **paths** attribute. The **paths** attribute is an array of `NSIndexPath` objects with the locations of every cell. The **animation** attribute indicates how the deletion is going to be animated. It is an enumeration of type `UITableViewRowAnimation` with the values `Fade`, `Right`, `Left`, `Top`, `Bottom`, `None`, `Middle`, and `Automatic`.

insertSections(indexes, withRowAnimation: animation)—This method inserts new sections at the location indicated by the **indexes** attribute. The **indexes** attribute is an object of the `NSIndexSet` class that defines the set of indexes of the sections to be inserted. The **animation** attribute indicates how the insertion is going to be animated. It is an enumeration of type `UITableViewRowAnimation` with the values `Fade`, `Right`, `Left`, `Top`, `Bottom`, `None`, `Middle`, and `Automatic`.

deleteSections(indexes, withRowAnimation: animation)—This method deletes the sections at the location indicated by the **indexes** attribute. The **indexes** attribute is an object of the `NSIndexSet` class that defines the set of indexes of the sections to be deleted. The **animation** attribute indicates how the insertion is going to be animated. It is an enumeration of type `UITableViewRowAnimation` with the values `Fade`, `Right`, `Left`, `Top`, `Bottom`, `None`, `Middle`, and `Automatic`.

UITableViewCell

Tables present the information in cells, one per row, with each cell in charge of displaying a single piece of data. Cells are created from the `UITableViewCell` class. The class offers the following properties and methods to create and configure each cell.

textLabel—This property returns the `UILabel` object used to present the primary text of standard cells. The text is modified assigning a new value to its `text` property.

detailTextLabel—This property returns the `UILabel` object used to present the secondary text of standard cells. The text is modified assigning a new value to its `text` property.

imageView—This property returns the `UIImageView` object used to present the image of standard cells. The image is modified assigning a new `UIImage` object to its `image` property.

accessoryType—This property sets or returns the type of the accessory view (a view, usually an icon, displayed on the right side of standard cells). It is an enumeration of type `UITableViewCellAccessoryType` with the values `None`, `DisclosureIndicator`, `DetailDisclosureButton`, `Checkmark`, and `DetailButton`.

selectionStyle—This property sets or returns the style of the cell when it is selected. It is an enumeration of type `UITableViewCellSelectionStyle` with the values `None`, `Blue`, `Gray`, and `Default`.

contentView—This property returns a reference to the view of the cell. From this reference we can access the rest of the elements in the cell or add new ones.

backgroundView—This property sets or returns the background view of the cell.

Table View Protocols

The `UITableView` class defines two properties to assign delegate objects to the table: `delegate` and `dataSource`. The delegates will be in charge of providing the data to display and additional information to configure the table. The protocols defined for these delegates are `UITableViewDelegate` and `UITableViewDataSource`. The following are the most frequently implemented methods of the `UITableViewDelegate` protocol.

tableView(tableView, heightForRowAtIndexPath: indexPath)—This method is called by the table to get the height of the cell at the location indicated by the **indexPath** attribute. The method must return a **CGFloat** value with the height of the cell. The **indexPath** attribute is an **NSIndexPath** object that we can read from inside the method to get information of the cell and determine what value to return. When the cells are all of the same height, it is recommendable to use the Table View’s **rowHeight** property instead.

tableView(tableView, heightForHeaderInSection: section)—This method is called by the table to get the height of the header for the section at the index indicated by the **section** attribute. The method must return a **CGFloat** value with the height of the header. The **section** attribute is an integer value that we can read from inside the method to get information of the section and determine what value to return.

tableView(tableView, heightForFooterInSection: section)—This method is called by the table to get the height of the footer for the section at the index indicated by the **section** attribute. The method must return a **CGFloat** value with the height of the footer. The **section** attribute is an integer value that we can read from inside the method to get information of the section and determine what value to return.

tableView(tableView, willDisplayCell: cell, forRowAtIndexPath: indexPath)—This method is called by the table when the cell indicated by the **cell** attribute is about to be presented on the screen. The **cell** attribute is a reference to the cell and the **indexPath** attribute is an **NSIndexPath** object with its location.

tableView(tableView, willDisplayHeaderView: view, forSection: section)—This method is called by the table when the header for the section indicated by the **section** attribute is about to be presented on the screen. The **view** attribute is a reference to the header view and the **section** attribute is the index of its section.

tableView(tableView, willDisplayFooterView: view, forSection: section)—This method is called by the table when the footer for the section indicated by the **section** attribute is about to be presented on the screen. The **view** attribute is a reference to the footer view and the **section** attribute is the index of its section.

tableView(tableView, didSelectRowAtIndexPath: indexPath)—This method is called by the table after the user selects a cell. The **indexPath** attribute is an **NSIndexPath** object with the location of the cell.

tableView(tableView, didDeselectRowAtIndexPath: indexPath)—This method is called by the table when the selected cell is deselected. The **indexPath** attribute is an **NSIndexPath** object with the location of the cell.

And the following are the most frequently implemented methods of the **UITableViewDataSource** protocol.

tableView(tableView, numberOfRowsInSection: section)—This method is called by the table to get the number of cells that have to be created in the section indicated by the **section** attribute. The method must return an integer value with the number of cells in the section. The **section** attribute is an integer value with the index of the section.

numberOfSectionsInTableView(tableView)—This method is called by the table to get the total number of sections that have to be created. The method must return an integer value with the number of sections.

tableView(tableView, cellForRowAtIndexPath: indexPath)—This method is called by the table to get the cell for the location indicated by the **indexPath** attribute. The method must return the `UITableViewCell` object that represents the cell. The **indexPath** attribute is an `NSIndexPath` object with the location of the required cell.

tableView(tableView, titleForHeaderInSection: section)—This method is called by the table to get the title for the header of the section indicated by the **section** attribute. The method must return a `String` value with the header's title. The **section** attribute is an integer value with the index of the section.

tableView(tableView, titleForFooterInSection: section)—This method is called by the table to get the title for the footer of the section indicated by the **section** attribute. The method must return a `String` value with the footer's title. The **section** attribute is an integer value with the index of the section.

tableView(tableView, commitEditingStyle: editingStyle, forRowAtIndexPath: indexPath)—This method is called by the table when the user inserts or deletes a row. It is implemented along with edition tools provided by Table Views. The **editingStyle** attribute is an enumeration of type `UITableViewCellEditingStyle` that indicates the type of operation performed. The possible values are `None`, `Delete`, and `Insert`.

sectionIndexTitlesForTableView(tableView)—This method is called by the table to get the strings that represent the sections in the table's index. The method must return an array of strings with the values for the indexes.

tableView(tableView, sectionForSectionIndexTitle: title, atIndex: index)—This method is called by the table to get the section corresponding to an index title. The method must return an integer value with the section's index. The **title** attribute is a string with the index's title, and the **index** attribute is the position where the title is located on the table's index.

tableView(tableView, moveRowAtIndexPath: sourceIndexPath, toIndexPath: destinationIndexPath)—This method is called when the user moves a cell to a different position on the table. It is implemented along with edition tools provided by Table Views. The **sourceIndexPath** attribute is the path in which the cell is currently located and the **destinationIndexPath** attribute is the path where the cell is

going to be placed.

tableView(tableView, canMoveRowAtIndexPath: indexPath)—This method is called when the user tries to move a cell to a different position on the table. The method must return a Boolean value to indicate whether the action is allowed or not. The **indexPath** attribute is an **NSIndexPath** object with the cell’s current path.

NSIndexPath

Tables identify their sections and rows with consecutive index values starting from 0. To report these values, the table uses objects of the **NSIndexPath** class. The class provides the following properties.

section—This property returns the index of the section where the row is located.

row—This property returns the index of the row.

item—This property returns the index of the item in a Collection View (we will study Collection Views in Chapter 15).

Implementing Table Views

The first step to include a table in our interface is to add the Table View to the main view. The **UITableView** class includes an initializer to create Table Views from code.

UITableView(frame: frame, style: style)—This initializer creates a **UITableView** object with the frame specified by the **frame** attribute. The **style** attribute is an enumeration of type **UITableViewStyle** with the values **Plain** and **Grouped**.

To work with Table Views on the Storyboard, the Object Library includes the Table View option.

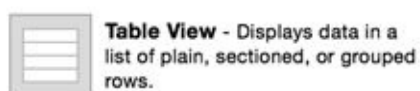


Figure 14-3: Table View option in the Object Library

Table Views are normal views and therefore they are positioned and sized inside the main view with constraints. Although we can give them any size we want, due to the type of information they managed, they usually occupy the whole screen.

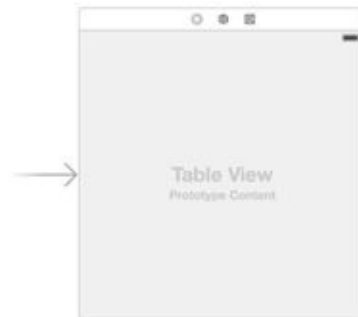


Figure 14-4: Table View pinned to the edges of the initial view

The `UITableViewCell` class also includes an initializer to create cells.

`UITableViewCell(style: style, reuseIdentifier: identifier)`—This initializer creates a `UITableViewCell` object for a cell with the style specified by the **style** attribute. The **style** attribute is an enumeration of type `UITableViewCellStyle` with the values **Default**, **Value1**, **Value2**, and **Subtitle**, and the **identifier** attribute is a string that identifies the cell (to use as a prototype for other cells).

The Object Library provides an option to add a cell to the Table View.

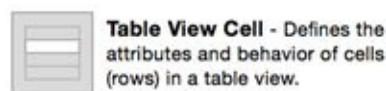


Figure 14-5: Table View Cell option in the Object Library

Dragging the option of Figure 14-5 to the Table View creates the cell.

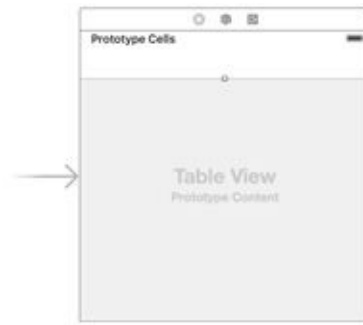


Figure 14-6: Table View with a prototype cell

Cells added to the Table View from the Object Library are prototype cells. When the app is executed, the view controller creates all the cells it needs to show the information on the screen from this prototype. Prototype cells in the Storyboard are identified from code using a string called *Identifier* set from the Attributes Inspector panel (Figure 14-7, number 2).

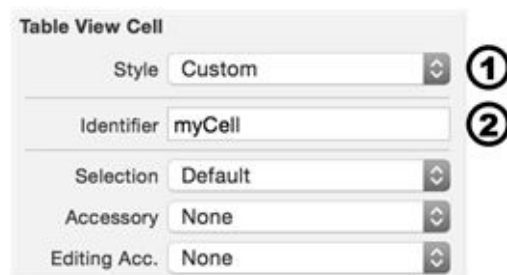


Figure 14-7: Configuration of a prototype cell

From this panel we can also select the style for the cell (Figure 14-7, number 1). There are five options available.

- **Custom** creates a cell with no style and custom content (the cell is empty, no elements are defined by default).
- **Basic (Default)** creates a cell with a single label for the title on the left and an optional Image View.
- **Right Detail (Value1)** creates a cell with a black label for the title on the left and a blue label for the description on the right.
- **Left Detail (Value2)** creates a cell with a blue label for the title on the left and a black label for the description on the right.
- **Subtitle** creates a cell with a black label for the title on top and a gray label for the description below.

The configuration of the table by default is enough to get it working, but the view controller has to adopt the protocols and implement their methods to provide the information the table is going to present. Most of the protocol methods are optional, but there are two of the `UITableViewDataSource` protocols that the table needs to call to know the amount of rows it has to present and the cells to include in each row. The view controller in the following example adopts the `UITableViewDataSource` protocol, assigns itself as the Table View's `dataSource` delegate, and then implements the methods required.

Listing 14-1: Feeding the table with data

```
import UIKit

class ViewController: UIViewController, UITableViewDataSource {
    @IBOutlet weak var myTable: UITableView!
    var items: [String]!

    override func viewDidLoad() {
        super.viewDidLoad()

        items = ["Lettuce", "Tomatoes", "Milk", "Granola", "Donuts", "Cookies", "Butter", "Cheese", "Lemonade",
        "Yogurt", "Oatmeal", "Juice", "Tea", "Coffee", "Bagels", "Brownies", "Potatoes", "Onions"]

        myTable.dataSource = self
    }

    func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return items.count
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell
    {
        let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)
        cell.textLabel?.text = items[indexPath.row]
        return cell
    }
}
```

The view controller needs an Outlet to access the Table View, but the cell is referenced from its identifier. The code of Listing 14-1 connects the table with an Outlet called `mytable` first and then defines a property called `items` to store the array of strings that is going to be shown by the table. To feed the table with these values, we have to implement the protocol's methods. The first method required is the `tableView(tableView, numberOfRowsInSection: section)`. From this method we have to return the number of items we are going to present in every section of the table. By default, tables only have one section, so we just have to count the elements in the `items` array and return the result. The second method requires a little bit of work. The `tableView(tableView, cellForRowAtIndexPath: indexPath)`

method is called every time the table is going to draw a row on the screen and needs its cell. Using the value of the index path received by the method, we have to create the cell for the row and return it. Since we have added a prototype cell to the table in the Storyboard, all we have to do to create the cell is to call the table's `dequeueReusableCellWithIdentifier()` method with the identifier of the prototype cell ("myCell" in our example). The method returns a `UITableViewCell` object that we have to modify to configure the cell with the right values for the row. For a standard cell (not customized), there are only three properties we can work with: `titleLabel` for the title of the cell, `detailTextLabel` for the description, and `imageView` for the image. In our example we get the element of the `items` array corresponding to the row's index and assign it to the `text` property of the `UILabel` object referenced by the `titleLabel` property. As a result, every cell on the table has a title defined with a value of the `items` array.

Do It Yourself: Create a new Single View Application project. Add a Table View to the main view and pin it to the edges of the view. Add a Table View Cell to the table. Select the cell and assign the identifier "myCell" to it from the Attributes Inspector panel. Also from this panel, select the Basic style for the cell. Connect the table to an Outlet in the view controller called `myTable`. Complete the view controller with the code of Listing 14-1. Run the application. You should see something similar to the left picture of Figure 14-1.

IMPORTANT: Cells are created from a prototype to improve performance. Using prototypes, the system can create only the cells that are visible, and reuse the `UITableViewCell` objects to represent new cells when the user scrolls the table (hence the name *reuse identifier*). You can create as many prototype cells as you want, just remember to give them a different identifier to be able to select what prototype you want to use from the `tableView(tableView, cellForRowAtIndexPath: indexPath)` method.

We can take advantage of the rest of the elements in a standard cell to present more information. The following example uses a prototype cell with the style Subtitle and three associated arrays to include the title, description, picture, and an accessory for every item on the list.

Listing 14-2: Taking advantage of the elements in a standard cell

```
import UIKit

class ViewController: UIViewController, UITableViewDataSource {
    @IBOutlet weak var myTable: UITableView!
    var items: [String]!
    var descriptions: [String]!
```

```

var images: [String]!
var selected: [Bool]!

override func viewDidLoad() {
    super.viewDidLoad()

    items = ["Lettuce", "Tomatoes", "Milk", "Granola", "Donuts", "Cookies", "Butter", "Cheese", "Lemonade",
"Yogurt", "Oatmeal", "Juice", "Tea", "Coffee", "Bagels", "Brownies", "Potatoes", "Onions"]

    descriptions = ["1 lb.", "Sweet tomatoes", "2 lbs.", "12 bars", "A dozen", "Oreos", "2", "Lactose free", "2 lbs.",
"Strawberry yogurt", "1 box", "Orange juice", "Green tea", "1 bag of beans", "6", "Chocolate brownies", "2 lbs.", "1
lb."]

    images = ["lettuce", "tomato", "milk", "granola", "donuts", "cookies", "butter", "cheese", "lemonade", "yogurt",
"oatmeal", "juice", "tea", "coffee", "bagels", "brownies", "potato", "onions"]

    selected = [true, true, true, true, true, true, true, true, true, true, true, true, true, true, true, true, true]

    myTable.dataSource = self
}

func tableView(tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
    return items.count
}

func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)

    let row = indexPath.row
    cell.textLabel?.text = items[row]
    cell.detailTextLabel?.text = descriptions[row]
    cell.imageView?.image = UIImage(named: images[row])
    if selected[row] {
        cell.accessoryType = .Checkmark
    }

    return cell
}
}

```

The view controller of Listing 14-2 uses four properties to store the data for the table. These are associated arrays, where the elements of the same index are part of the same unit of information. The element at index 0 of the `items` array contains the name of the first item, the element at index 0 of the `descriptions` array contains the description of the first item, and so on. The `selected` property is an array of Boolean values. When the value of the element is `true`, we show an accessory in that cell. The initial values were defined as `true`, so the table will look like Figure 14-8.

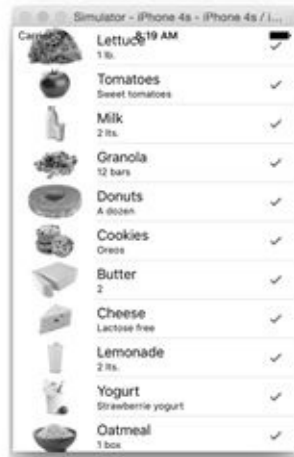


Figure 14-8: Subtitle cells in the simulator with Checkmark accessories on the right

IMPORTANT: The status bar is transparent. If you pin the Table View to the top of the main view, the first item is positioned behind the bar. How to solve this is a matter of design and what you want for your application. The `UITableView` is a subclass of `UIScrollView` and therefore you could create a padding at the top to move the items down modifying the `contentInset` property of the table (`myTable.contentInset = UIEdgeInsets(top: 20, left: 0, bottom: 0, right: 0)`). But, as we will see later, tables are usually embedded in Navigation Controllers, and they take care of these aspects for us.

The Basics: Arrays combined to store pieces of data that correspond to the same unit of information are called *Parallel Arrays*. Parallel arrays are practical when working with small applications, like the one created in Listing 14-2, but are not recommendable for larger applications. They are error prone and difficult to sort. If we try to order one array alphabetically, for example, the indexes won't match anymore. We will see better forms to organize the model in further examples.

Selection

Accessory types like the one implemented in the previous example are used to reflect or propose user interaction. For example, the Disclosure Indicator type is used to indicate to the user that there is more information available for that item. To take advantage of these indicators, we need to let the user interact with the table. The `UITableViewDelegate` protocol offers a few methods for this purpose. In the following example we implement the `tableView(tableView, didSelectRowAtIndexPath: indexPath)` method to modify the accessory type every time a cell is selected.

Listing 14-3: Selecting items

```
import UIKit
```

```
class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {  
    @IBOutlet weak var myTable: UITableView!  
    var items: [String]!  
    var descriptions: [String]!  
    var images: [String]!  
    var selected: [Bool]!  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        items = ["Lettuce", "Tomatoes", "Milk", "Granola", "Donuts", "Cookies", "Butter", "Cheese", "Lemonade",  
"Yogurt", "Oatmeal", "Juice", "Tea", "Coffee", "Bagels", "Brownies", "Potatoes", "Onions"]  
  
        descriptions = ["1 lb.", "Sweet tomatoes", "2 lts.", "12 bars", "A dozen", "Oreos", "2", "Lactose free", "2 lts.",  
"Strawberry yogurt", "1 box", "Orange juice", "Green tea", "1 bag of beans", "6", "Chocolate brownies", "2 lbs.", "1  
lb."]  
  
        images = ["lettuce", "tomato", "milk", "granola", "donuts", "cookies", "butter", "cheese", "lemonade", "yogurt",  
"oatmeal", "juice", "tea", "coffee", "bagels", "brownies", "potato", "onions"]  
  
        selected = [true, true, true, true, true, true, true, true, true, true, true, true, true, true, true, true]  
        myTable.dataSource = self  
        myTable.delegate = self  
    }  
    func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {  
        return items.count  
    }  
    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {  
        let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)  
        let row = indexPath.row  
        cell.textLabel?.text = items[row]  
        cell.detailTextLabel?.text = descriptions[row]  
        cell.imageView?.image = UIImage(named: images[row])  
  
        if selected[row] {  
            cell.accessoryType = .Checkmark  
        }  
        return cell  
    }  
    func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {  
        let cell = myTable.cellForRowAtIndexPath(indexPath)  
        let row = indexPath.row  
        if selected[row] {  
            cell?.accessoryType = .None  
            selected[row] = false  
        } else {
```

```

        cell?.accessoryType = .Checkmark
        selected[row] = true
    }
    myTable.deselectRowAtIndexPath(indexPath, animated: true)
}
}

```

The view controller of Listing 14-3 adopts the `UITableViewDelegate` protocol to be able to implement the `tableView(tableView, didSelectRowAtIndexPath: indexPath)` method. This method is called by the Table View every time the user selects a cell. The information for the cell is sent to the method through the `indexPath` attribute. We first take this data to get a reference to the selected cell with the table's `cellForRowAtIndexPath()` method and then use this reference to modify the `accessoryType` property of that cell. If the cell already had a checkmark, we remove it or put it back otherwise.

Every time a cell is selected, the table highlights the cell with a gray background. The background remains as long as the cell keeps its selected state. This state doesn't change until the user selects another cell, and therefore it is our responsibility to change it if it is no longer necessary. To deselect the cell we have to call the table's `deselectRowAtIndexPath()` method, as we did in Listing 14-3. Although it is recommended to deselect the cell every time the selection is no longer necessary, we also have the alternative to change the cell's selection style. For example, if we don't want the cell to show a gray background when selected, we can assign the value `None` to the cell's `selectionStyle` property and the selection won't be visible anymore.

Do It Yourself: Select the prototype cell created in the previous example and change its style to Subtitle. Download the thumbnail images from our website and add them to the Assets Catalog. Replace the code in the view controller with the code of Listing 14-3. Run the application. You should see something similar to Figure 14-8. Tap on the cells to select or deselect an item.

Sections and Indexes

Tables may organize the information in sections. By default, a table only has one section and all the rows belong to the same section, but we may declare more from the `UITableViewDataSource` protocol's methods. The `numberOfSectionsInTableView(tableView)` method returns the number of sections available and the `tableView(tableView, titleForHeaderInSection: section)` method provides the title for each section.

Listing 14-4: Defining sections

```
import UIKit
```

```
class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
    @IBOutlet weak var myTable: UITableView!

    var categories: [String]!
    var items: [[String]]!

    override func viewDidLoad() {
        super.viewDidLoad()

        categories = ["B", "C", "D", "G", "J", "L", "M", "O", "P", "T", "Y"]
        items = [["Bagels", "Brownies", "Butter"], ["Cheese", "Coffee", "Cookies"], ["Donuts"], ["Granola"], ["Juice"],
["Lemonade", "Lettuce"], ["Milk"], ["Oatmeal", "Onions"], ["Potatoes"], ["Tea", "Tomatoes"], ["Yogurt"]]

        myTable.delegate = self
        myTable.dataSource = self
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)

        let row = indexPath.row
        let section = indexPath.section
        cell.textLabel?.text = items[section][row]

        return cell
    }

    func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return items[section].count
    }

    func numberOfSectionsInTableView(tableView: UITableView) -> Int {
        return categories.count
    }

    func tableView(tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
        let title = categories[section]
        return title
    }
}
```

Like rows, sections are identified with a consecutive index starting from 0, but when working with sections we have to contemplate that the sequence of indexes for rows start from 0 in every section. The data for the table has to observe this pattern, and this is why in the example of Listing 14-4 we use a multidimensional array to store the values (see Chapter 2, Listing 2-48). The first element of the `items` array is an array with the items of section 0, the second element is an array with the items of section 1, and so on. This provides a data structure that is easy to associate with the structure of the table. For example, to return the number of rows in a section from the `tableView(tableView, numberOfRowsInSection: section)` method, all we have to do is count the elements in the array

corresponding to the section (`items[section].count`). The same happens when we prepare the cell in the `tableView(tableView, cellForRowAtIndexPath: indexPath)` method. The value for the title of the cell is obtained from the `items` array using the values of the `section` and `row` properties as indexes.

For the titles of every section we use a separate array called `categories`. Each element in this array corresponds to a section in the `items` array, and therefore we use it to count the sections available in the `numberOfSectionsInTableView(tableView)` method. The values of the `categories` array are returned from the `tableView(tableView, titleForHeaderInSection: section)` method to represent the titles for each section. If we don't implement this method, the sections' headers are not shown on the screen.

If we set the style of the table for the previous example as Plain, we will see something similar to the right picture of Figure 14-1, where the headers of each section adopt a size relative to the size of their content. Implementing the `tableView(tableView, heightForHeaderInSection: section)` method of the `UITableViewDelegate` protocol we can specify a custom height.

Listing 14-5: Assigning a specific height to the headers

```
func tableView(tableView: UITableView, heightForHeaderInSection section: Int) -> CGFloat {  
    return 50  
}
```

The Basics: By checking the value of the `section` attribute we can return a specific height for each section. The same may be achieved for the footers and cells implementing the `tableView(tableView, heightForFooterInSection: section)` and `tableView(tableView, heightForRowAtIndexPath: indexPath)` methods, respectively.

There are also methods available in the `UITableViewDelegate` protocol to customize the views of headers, footers and cells before they are drawn on the screen. They are implemented to perform last-minute settings. For example, we may implement the `tableView(tableView, willDisplayCell: cell, forRowAtIndexPath: indexPath)` to change aspects of the cells, such as the background color. In the following example, we calculate if the index for the row of the current cell is odd or even and assign a background color accordingly (see Listing 2-68).

Listing 14-6: Changing the background color of the cells

```
func tableView(tableView: UITableView, willDisplayCell cell: UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) {  
    if indexPath.row % 2 == 0 {  
        cell.backgroundColor = UIColor(red: 245.0/255.0, green: 245.0/255.0, blue: 245.0/255.0, alpha: 1)  
    } else {
```

```

cell.backgroundColor = UIColor.whiteColor()
}
}

```

Figure 14-9, below, shows some examples. The picture on the left illustrates what the table created by Listing 14-3 looks like after the method of Listing 14-6 is added to the view controller, and the picture on the right illustrates what the table created by Listing 14-4 looks like after the height of the headers is incremented with the method of Listing 14-5.

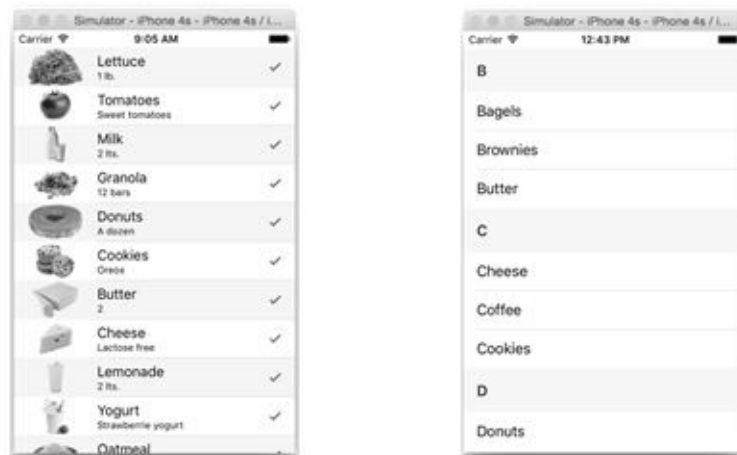


Figure 14-9: *Different designs for cells and sections*

Do It Yourself: Add the method in Listing 14-5 to the view controller of the example of Listing 14-4 and the method in Listing 14-6 to the view controller of the example of Listing 14-3. These examples should look in the simulator like the pictures of Figure 14-9.

When working with sections and Plain tables, we can also provide an index on the right to help the user navigate through the items (see right picture in Figure 14-1). The `UITableViewDataSource` protocol includes a method to return an array with the titles for the indexes. Although the index titles may be anything from a single letter to a sentence, it is frequently used to organize the table alphabetically. For example, we can add the following method to the view controller of Listing 14-4 to create an index with the letters assigned to the titles of the headers.

Listing 14-7: *Creating the index*

```

func sectionIndexTitlesForTableView(tableView: UITableView) -> [String]? {
    return categories
}

```


Do It Yourself: Add the method in Listing 14-7 to the example of Listing 14-4. Run the application. You should see something similar to the right picture of Figure 14-1. Tap the letters in the index at the right side of the screen. The table will scroll to the selected section.

The previous index contains one item per section, but sometimes what we want is to show a standard index that includes all the possible options, like a consecutive list of numbers or the entire alphabet, for example. In this case, we have to provide a separate array for the index and then tell the table what section corresponds to the selected item in the index with the `tableView(tableView, sectionForSectionIndexTitle: title, atIndex: index)` method.

Listing 14-8: Creating a general index

```
import UIKit

class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
    @IBOutlet weak var myTable: UITableView!
    var indexLetters: [String]!
    var categories: [String]!
    var items: [[String]]!
    override func viewDidLoad() {
        super.viewDidLoad()

        indexLetters = ["#", "A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q",
            "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
        categories = ["B", "C", "D", "G", "J", "L", "M", "O", "P", "T", "Y"]
        items = [["Bagels", "Brownies", "Butter"], ["Cheese", "Coffee", "Cookies"], ["Donuts"], ["Granola"], ["Juice"],
            ["Lemonade", "Lettuce"], ["Milk"], ["Oatmeal", "Onions"], ["Potatoes"], ["Tea", "Tomatoes"], ["Yogurt"]]
        myTable.delegate = self
        myTable.dataSource = self
        myTable.contentInset = UIEdgeInsets(top: 20, left: 0, bottom: 0, right: 0)
    }
    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)
        let row = indexPath.row
        let section = indexPath.section
        cell.textLabel?.text = items[section][row]
        return cell
    }
    func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return items[section].count
    }
}
```

```

}
func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return categories.count
}
func tableView(tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
    let title = categories[section]
    return title
}
func sectionIndexTitlesForTableView(tableView: UITableView) -> [String]? {
    return indexLetters
}
func tableView(tableView: UITableView, sectionForSectionIndexTitle title: String, atIndex index: Int) -> Int {
    if let index = categories.indexOf(title) {
        return index
    }
    return 0
}
}

```

The example of Listing 14-8 creates a new array with the letters of the alphabet to represent every item of the index and implements the `tableView(tableView, sectionForSectionIndexTitle: title, atIndex: index)` method to let the table know the relation between these items and the sections. The method is called every time the user taps on the index. The letter tapped is assigned to the method's `title` attribute. For example, when the user taps on the letter C, the value of `title` is "C" and therefore the `indexOf()` method returns the value 1 (the index of the letter C in the `categories` array). The value is returned and the table scrolls to the section C. If the title does not correspond to any section, the value return is 0 (the table scrolls to the first section).

The value returned when no section matches the index title depends on what we want for our application. The method could simply return the index corresponding to the first section (as we did in our example), the last section, or it could try to find the nearest section and return its index. The following example presents a method that looks for a section that matches the title and then, if no match is found, keeps comparing previous titles until it finds a match. The method only returns 0 when no match has been found. In consequence, the table always scrolls to the section selected by the user or the previous one if the section selected doesn't exist.

Listing 14-9: Finding the nearest section

```

func tableView(tableView: UITableView, sectionForSectionIndexTitle title: String, atIndex index: Int) -> Int {
    let start = indexLetters.indexOf(title)
    for var counter = start!; counter >= 0; counter— {

```

```

let letter = indexLetters[counter]
if let index = categories.indexOf(letter) {
    return index
}
}
return 0
}

```

Do It Yourself: Replace the view controller of Listing 14-4 with the view controller of Listing 14-8. Run the application. Tap on the letter in the index. The table scrolls to the section if it exists, or to the section with the index 0 otherwise. Replace the `tableView(tableView, sectionForSectionIndexTitle: title, atIndex: index)` method of Listing 14-8 by the method of Listing 14-9. Run the application again. Tap on the index items that don't match with any section to see how the table scrolls to the nearest one.

Custom Cells

Sometimes the labels and the Image View included in a standard cell may not satisfy the requirements of our application. In cases like this, cells may be customized with the elements we need. Designing the cell is easy. We just have to set the style for the cell as Custom, drag the elements we want to include from the Object Library to the prototype cell, and set the constraints as we do with any other views. The Size Inspector panel even offers an option to extend the height of the cell when the standard height is not enough for our design.

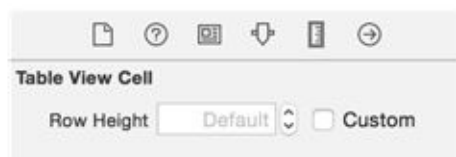


Figure 14-10: Custom height for prototype cells

The Basics: You can also set the height of the cell in the Storyboard by dragging down the little square at the bottom of the cell. The size set in the Storyboard does not determine the real size of the cell. This final size is determined by setting the table's `rowHeight` property or implementing the `tableView(tableView, heightForRowAtIndexPath: indexPath)` method of the `UITableViewDelegate` protocol, as we will see soon.

For the following example we set the height for the cell as 120 and added an Image

View on the right and a label on the left. The Image View has a height of 100 points, leaving a 10-points margin at the top and bottom.

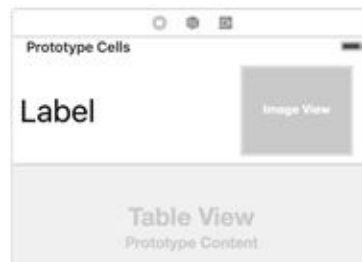


Figure 14-11: Custom cell with a label and an Image View

To have access to the elements of a custom cell we have to create a subclass of `UITableViewCell` and assign it to the cell. The file for the subclass is created as we did before for view controllers (Figures 13-3 and 13-4), only this time we have to select the `UITableViewCell` class as the superclass. As always, there are no requirements for the name, but it is better to give it a name related to the cell (for our example we just called it `FoodCell`). Once the file with the subclass is created, it has to be assigned to the cell from the Identity Inspector panel (Figures 13-6 and 13-7).

The current template generated by Xcode for subclasses of `UITableViewCell` include the methods `awakeFromNib()` and `setSelected()`. The `awakeFromNib()` method is like the `viewDidLoad()` method for the view controller. It is called as soon as the view is loaded from the Storyboard and it is used to perform any custom initialization we need. The `setSelected()` method, on the other hand, is called when the cell is selected, allowing us to perform custom tasks and manage the selection from inside the cell. These methods are good if we need to perform additional tasks when the cell is created or selected, as we will see in Chapter 15.

The Basics: The `awakeFromNib()` method is part of a protocol adopted by objects created from the Storyboard. The method is called as soon as the object is loaded from the file to report its existence. One of the classes that implement this method is the `UIViewController` class. In this class, the method is called before the rest of the methods implemented by the class, such as `viewDidLoad()` or `viewWillAppear()`. You may implement it along with these methods or by itself to initialize the view controller.

Once the `UITableViewCell` subclass is created and assigned to the cell, we have to create the Outlets we need to access the elements added to the prototype cell in the Storyboard. Xcode doesn't show the file in the assistant editor when we select the cell, we have to click on the Automatic option at the top (Figures 6-99 and 6-100), select Manual, and then navigate through the menu to find the file with the `UITableViewCell` subclass we just created (`FoodCell` in our example). With the file opened in the Assistant Editor we can now control-

drag the lines from the elements on the cell to the subclass to create the Outlets. The following example shows what the subclass looks like when the process is over (we have deleted the methods included by the template and created two Outlets called `cellTitle` and `cellImage`).

Listing 14-10: Creating a subclass of `UITableViewCell` for a custom cell

```
import UIKit

class FoodCell: UITableViewCell {
    @IBOutlet weak var cellTitle: UILabel!
    @IBOutlet weak var cellImage: UIImageView!
}
```

Do It Yourself: Create a new Single View Application project. Add a Table View to the main view. Add a Table View Cell to the table. Select the cell and open the Size Inspector panel (Figure 14-10). Click on Custom and insert the value 120 in the Row Height field. Add a label and an Image View to the prototype cell, as shown in Figure 14-11. Add the necessary constraints between the elements and the cell. Create a new file with a subclass of `UITableViewCell` and call it `FoodCell`. Select the cell, open the Identity Inspector panel and assign the subclass to the cell (Figures 13-6 and 13-7). Open the Assistant Editor, click on Automatic, select Manual, and search for the `FoodCell.swift` file. When opened, erase the code inside the class included by the template and create Outlets for the cell’s label and Image View called `cellTitle` and `cellImage`, respectively (Listing 14-10).

With the prototype cell ready, it is time to create the view controller. The main difference when working with custom cells is that now we don’t have access to the standard elements anymore, we have to modify the custom elements we added in the Storyboard from the Outlets created in the `UITableViewCell` subclass (`cellTitle` and `cellImage`).

Listing 14-11: Working with custom cells from the view controller

```
import UIKit

class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
    @IBOutlet weak var myTable: UITableView!
    var items: [String]!
    var images: [String: String]!
    override func viewDidLoad() {
        super.viewDidLoad()
        items = ["Lettuce", "Tomatoes", "Milk", "Granola", "Donuts", "Cookies", "Butter", "Cheese", "Lemonade",
        "Yogurt", "Oatmeal", "Juice", "Tea", "Coffee", "Bagels", "Brownies", "Potatoes", "Onions"]
    }
```

```
images = ["Lettuce": "lettuce", "Tomatoes": "tomato", "Milk": "milk", "Granola": "granola", "Donuts": "donuts",
"Cookies": "cookies", "Butter": "butter", "Cheese": "cheese", "Lemonade": "lemonade", "Yogurt": "yogurt",
"Oatmeal": "oatmeal", "Juice": "juice", "Tea": "tea", "Coffee": "coffee", "Bagels": "bagels", "Brownies": "brownies",
"Potatoes": "potato", "Onions": "onions"]
```

```
items.sortInPlace({ (value1, value2) in value1 < value2 })

myTable.dataSource = self
myTable.delegate = self
}

func tableView(tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
    return items.count
}

func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath) as! FoodCell
    let row = indexPath.row
    let title = items[row]
    cell.cellTitle.text = title

    if let image = images[title] {
        cell.cellImage.image = UIImage(named: image)
    }
    return cell
}

func tableView(tableView: UITableView, heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {
    return 120
}
}
```

The `dequeueReusableCellWithIdentifier(identifier, forIndexPath: path)` method returns a `UITableViewCell` object. This is fine when our prototype cell is defined with a standard style, but for custom cells we need the object to be of the type of our subclass. In Listing 14-11 we cast the object as `FoodCell` and then assign the values from the `items` array and the `images` dictionary to the properties `cellTitle` and `cellImage`, respectively.

This time we organized the data for the table combining an array and a dictionary to be able to sort the items in alphabetical order (see the `sortInPlace()` method). When we use parallel arrays, as we did in previous examples, we can't order the items because once one array is sorted the indexes don't match anymore. Combining an array with a dictionary allows us to order the elements of the `items` array in any way we want and still find the right image in the `images` dictionary using the name of the item as key.

Notice that for the cell to have the right height, we also have to declare the `tableView(tableView, heightForRowAtIndexPath: indexPath)` method. The height defined in the Size Inspector panel is only considered in the Storyboard. The result is shown in Figure 14-12.

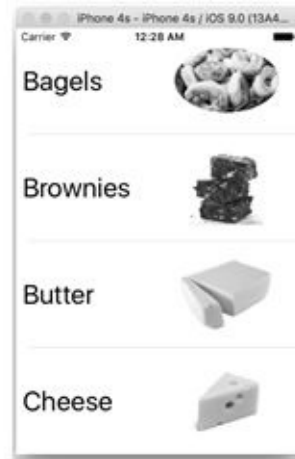


Figure 14-12: Table with custom cells

Self-Sizing Cells

Custom cells have a fixed height. They are either of the size determined as default by the system or of the size we set from the Table View's `rowHeight` property or the `tableView(tableView, heightForRowAtIndexPath: indexPath)` method (see Listing 14-11). The `UITableView` class defines a special constant called `UITableViewAutomaticDimension` we can assign to the Table View's `rowHeight` property to force the system to calculate the size of the cell according to its content. The process requires us to establish a few conditions. Besides assigning the `UITableViewAutomaticDimension` constant to the `rowHeight` property, we also have to assign an estimated height to the table's `estimatedRowHeight` property, define constraints for the elements of our cell to pin them to every side of the cell, including the bottom, and declare the labels as multiple line labels, assigning the value 0 to their `numberOfLines` property. Figure 14-12 shows an interface prepared to present variable height cells. The interface includes a Table View with a Custom prototype cell and a label inside the cell, the size of which has been determined by four Space constraints, one for each side.

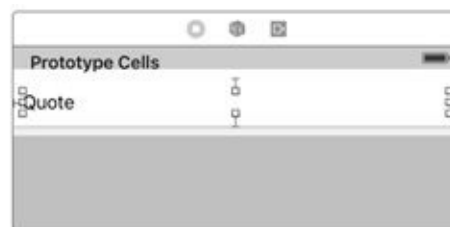


Figure 14-13: Cell ready to be presented with variable height

For this example we assigned to the cell a `UITableViewCell` subclass called `QuotesCell` and created an Outlet for the label called `quotesLabel`.

Listing 14-12: Defining the subclass

```
import UIKit

class QuotesCell: UITableViewCell {
    @IBOutlet weak var quotesLabel: UILabel!
}
```

The only thing left is to define the code for the view controller and set the values for the **estimatedRowHeight** and **rowHeight** properties.

Listing 14-13: Creating cells of variable height

```
import UIKit

class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
    @IBOutlet weak var myTable: UITableView!
    var items: [String]!
    override func viewDidLoad() {
        super.viewDidLoad()
        items = ["Don't walk behind me; I may not lead. Don't walk in front of me; I may not follow. Just walk beside me and be my friend.",
            "Don't cry because it's over, smile because it happened.",
            "They push the human race forward. And while some may see them as the crazy ones, we see genius. Because the people who are crazy enough to think they can change the world, are the ones who do.",
            "The only thing standing between you and your goal is the bullshit story you keep telling yourself as to why you can't achieve it.",
            "To be yourself in a world that is constantly trying to make you something else is the greatest accomplishment."]
        myTable.dataSource = self
        myTable.delegate = self
        myTable.estimatedRowHeight = 44.0
        myTable.rowHeight = UITableViewAutomaticDimension
        myTable.contentInset = UIEdgeInsets(top: 20, left: 0, bottom: 0, right: 0)
    }
    func tableView(tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
        return items.count
    }
    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath) as! QuotesCell
        let row = indexPath.row
        cell.quotesLabel.text = items[row]
        return cell
    }
}
```


Except for the definition of the `estimatedRowHeight` and `rowHeight` properties, there is nothing new in the view controller of Listing 14-13. The value for the `estimatedRowHeight` property is an approximate height that helps the system improve performance. We usually set this value to the value in the Storyboard (44 by default). Figure 14-14 shows the application in the simulator. Every cell adapts its size to show the entire quote for each row.

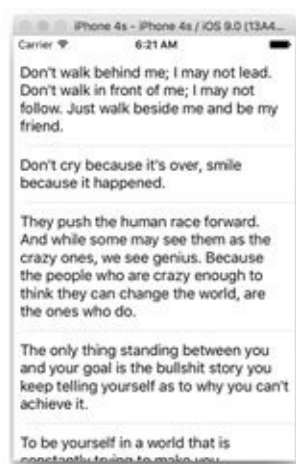


Figure 14-14: Cells with variable height

Do It Yourself: Create a new Single View Application project. Add a Table View to the main view and connect it to the view controller with an Outlet called `myTable`. Add a cell to the table and give it the identifier “myCell”. Create a `UITableViewCell` subclass called `QuotesCell` and assign it to the cell. Add a label to the cell and pin it to every side with Space constraints. Connect the label to an Outlet called `quotesLabel` (Listing 14-12). Complete the view controller with the code of Listing 14-13. Run the application. You should see something similar to Figure 14-14.

IMPORTANT: Self-Sizing cells adapt their size to the size of their content. This, combined with dynamic type fonts, provides an excellent experience for the user. To take advantage of these features, select the cell’s label and change its style to Body from the Attributes Inspector panel. Now not only the size of the cell will adapt to its content but also the size of the text will change to comply with the device’s settings.

14.2 Table Views and Navigation Controllers

Tables in iOS are meant to work along with other main views to provide more functionality. The addition of new rows, for example, requires a separate main view to provide the fields and buttons the user needs to insert the information. This is the reason why we frequently see applications with tables embedded in Navigation Controllers. Navigation Controllers present views with a transition that feels more natural to the user and helps establish the relationship between the other views and the table. Figure 14-15 shows what a view with a table looks like after it is embedded in a Navigation Controller.

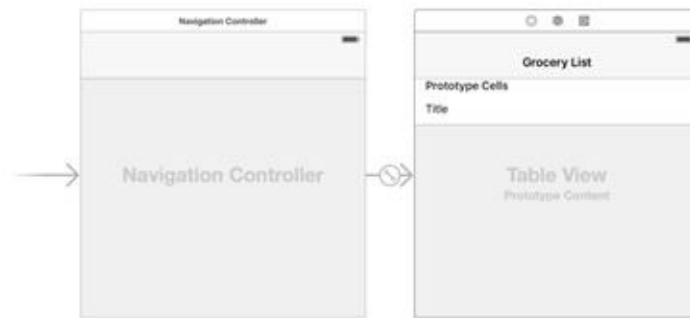


Figure 14-15: Table embedded in a Navigation Controller

The Basics: Apple recognized this symbiosis between Navigation Controllers and tables a long time ago and in recent systems integrated both elements to make them look like they were built together. When we embed a table in a Navigation Controller, the system inserts padding at the top of the table to move the items below the Navigation Bar, but it gives the bar a translucent background by default to let the user see the items when the table is scrolled up.

A common function provided by tables is the possibility to select a row and see more information related to the item. For example, considering the applications created before, we may add a main view that shows the nutritional values associated with an item every time the user taps on its cell.

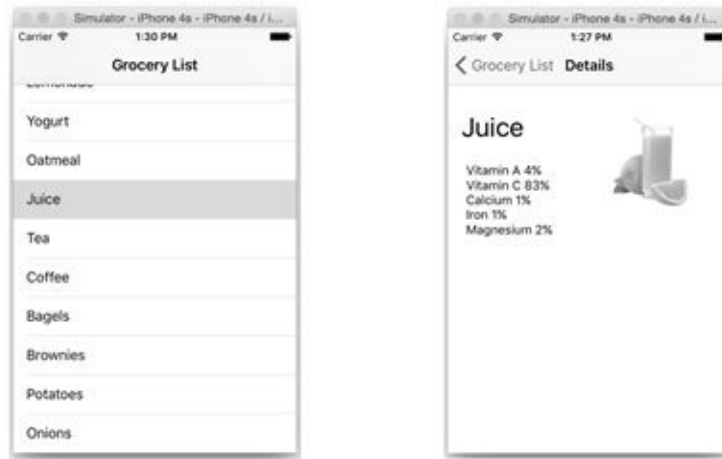


Figure 14-16: Table with a detail view

The view presenting additional information is usually called *Detail View*, because it shows details about the selected item. To create a detail view we have to add a new main view to the Storyboard and connect it with a Show Segue to the view with the table (Figure 13-18). For our example, we have added a label, an Image View, and a Text View to the detail view to present the information.

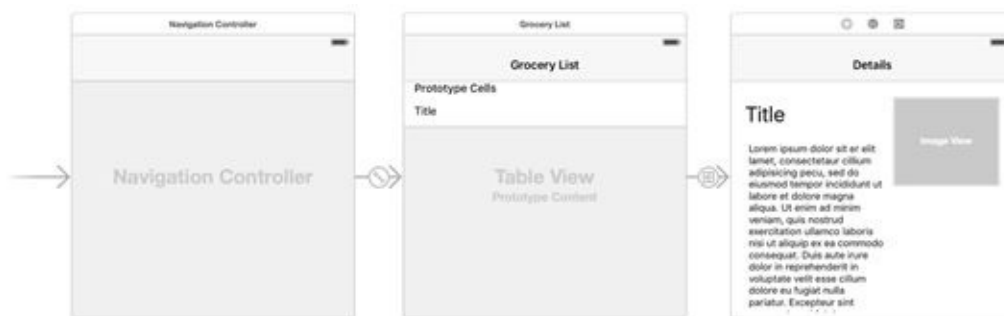


Figure 14-17: Detail view in the Storyboard

Detail views present additional information of every item on the table. Because the amount of data is now significant, it is better to manage this information from an external model, as those created in Chapter 13. The following code recreates that model to build the structure we need for the application of Figure 14-16. The **ApplicationData** structure for this example contains two properties: an array called **items** with the names of the items and a dictionary called **details** with the nutritional values of every item. The dictionary has a string as key and an array as value. The dictionary's keys are the values from the **items** array and the dictionary's values are arrays with two elements that represent the name of the image and the nutritional information of each item.

Listing 14-14: Building a model with an array for the rows and a dictionary for the details

```
struct ApplicationData {
    var items: [String] {
```

```

didSet {
    items.sortInPlace({ (value1, value2) in value1 < value2 })
}
}

var details: [String: [String]]
init() {
    items = ["Lettuce", "Tomatoes", "Milk", "Granola", "Donuts", "Cookies", "Butter"]
    details = [
        "Lettuce": ["lettuce", "Vitamin A 148%\r\nVitamin C 15%\r\nCalcium 3%\r\nIron 4%\r\nVitamin B-6 5%\r\nMagnesium 3%"],
        "Tomatoes": ["tomato", "Vitamin A 16%\r\nVitamin C 22%\r\nCalcium 1%\r\nIron 1%\r\nVitamin B-6 5%\r\nMagnesium 2%"],
        "Milk": ["milk", "Calcium 12%\r\nVitamin B-12 8%\r\nMagnesium 2%"],
        "Granola": ["granola", "Vitamin C 1%\r\nCalcium 6%\r\nIron 16%\r\nVitamin B-6 5%\r\nMagnesium 24%"],
        "Donuts": ["donuts", "Vitamin C 2%\r\nCalcium 2%\r\nIron 22%\r\nVitamin B-12 1%\r\nMagnesium 7%"],
        "Cookies": ["cookies", "Vitamin A 1%\r\nCalcium 3%\r\nIron 15%\r\nVitamin D 1%\r\nVitamin B-6 5%\r\nVitamin B-12 1%"],
        "Butter": ["butter", "Vitamin A 49%\r\nCalcium 2%\r\nVitamin D 15%\r\nVitamin B-12 3%"]
    ]
    items.sortInPlace({ (value1, value2) in value1 < value2 })
}
}

var AppData = ApplicationData()

```

A great advantage of using an external model to declare the data for the app is that we can manage the data and select what and how it is going to be available. In the example of Listing 14-14, we take advantage of the property's `didSet()` method to sort the elements of the `items` array in alphabetical order every time a new item is introduced. This alleviates the work of the view controllers that now don't have to worry about how to present the data, they just have to read it from the model and send it to the views.

The Basics: The nutritional values in the example of Listing 14-14 are separated by the characters `\r\n`. This is a traditional sequence of characters used to indicate a line break. The characters after the sequence are printed on a new line. It is a simple way to create new lines in multiple line elements like labels and Text Views.

A table inside a Navigation Controller is managed the same way we did before, but because this time we are using a more complex model, instead of reading the `items` property from the view controller we have to read it from the `AppData` structure. Letting the user select a cell and open its detail view also combines techniques implemented before. We have to detect when a cell is selected with the `tableView(tableView, didSelectRowAtIndexPath: indexPath)` method and trigger the Segue with the `performSegueWithIdentifier()` method (see Listings 13-6 and 14-15).

Listing 14-15: Opening the detail view when a cell is selected

```
import UIKit

class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
    @IBOutlet weak var myTable: UITableView!

    override func viewDidLoad() {
        super.viewDidLoad()
        myTable.delegate = self
        myTable.dataSource = self
    }

    func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        let data = AppData.items
        return data.count
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let data = AppData.items
        let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)
        cell.textLabel?.text = data[indexPath.row]
        return cell
    }

    func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
        performSegueWithIdentifier("showDetails", sender: self)
    }

    override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        if segue.identifier == "showDetails" {
            let controller = segue.destinationViewController as! DetailViewController
            if let path = myTable.indexPathForSelectedRow {
                controller.selected = path.row
            }
        }
    }
}
```

The view controller of Listing 14-15 also implements the `prepareForSegue()` method to tell the view controller of the detail view what was the row selected by the user (see Listing 13-4). When the user selects a row, the table calls the `tableView(tableView, didSelectRowAtIndexPath: indexPath)` method. From this method we execute the `performSegueWithIdentifier()` method to trigger the Segue connected to the detail view (identified in our example with the name “showDetails”). When a Segue is triggered, the system calls the `prepareForSegue()` method. In this method we get a reference to the view

controller of the detail view and update the value of its `selected` property with the index path of the selected row (obtained from the Table View's property `indexPathForSelectedRow`).

The view controller for the detail view needs a property called `selected` to store the index of the selected row, and also Outlets for each of the elements on the interface to show the information on the screen. We called this view controller **DetailViewController**.

Listing 14-16: Showing the details for the selected row

```
import UIKit

class DetailViewController: UIViewController {
    @IBOutlet weak var titleItem: UILabel!
    @IBOutlet weak var imageItem: UIImageView!
    @IBOutlet weak var nutritionItem: UITextView!

    var selected: Int!

    override func viewDidLoad() {
        super.viewDidLoad()

        if selected != nil {
            let item = AppData.items[selected]
            if let data = AppData.details[item] {
                titleItem.text = item
                imageItem.image = UIImage(named: data[0])
                nutritionItem.text = data[1]
            }
        }
    }
}
```

The **DetailViewController** class checks whether the `selected` property contains a value or not and then gets the information corresponding to the selected row from the `details` dictionary in the **AppData** structure to show the values on the screen.

Do It Yourself: Create a new Single View Application project. Add a table and a cell to the main view. Identify the cell with the name “myCell” (Figure 14-7) and connect the table to the view controller with an Outlet called **myTable**. Embed the main view in a Navigation Controller. You should see something similar to Figure 14-15. Add another main view to the Storyboard. Create a Show Segue from the main view with the table to the second view (Figure 13-18) and identify the Segue with the string “showDetails”. Add a label, an Image View and a Text View to the second view to get an interface similar to Figure 14-17. Create a view controller file for the second view called **DetailViewController** (a subclass of **UIViewController**). Create the Outlets for the

elements in the view with the names used in Listing 14-16. Create a Swift file called `ApplicationData.swift` for the `AppData` structure of Listing 14-14. Complete the `ViewController` class with the code of Listing 14-15 and the `DetailViewController` class with the code of Listing 14-16. Run the application and select a row. You should see something similar to Figure 14-16.

Once selected, the rows remain in that state until the user selects another row. This means when the user comes back from the detail view, the selected row is still selected. As we have seen before, to deselect a row we have to call the Table View's `deselectRowAtIndexPath()` method (see Listing 14-3). Where we call this method depends on what we want to achieve with our app. The only thing we have to remember is that once the row is deselected, the `indexPathForSelectedRow` property returns `nil` (because there is no selected row anymore), so we always have to deselect the cell after the value of its row is no longer needed. In our example, the best place to call the method would be after the selected row was assigned to the `selected` property of the detail view (`controller.selected = path.row`). A nice effect is achieved by calling the method from the `viewWillAppear()` method. The `viewWillAppear()` method is executed by the system every time the main view is going to be shown on the screen, and therefore we can use it to perform tasks when the user comes back from the detail view. The following code shows an implementation of this method that checks if there is a selected row and deselects it.

Listing 14-17: Deselecting the selected row

```
override func viewWillAppear(animated: Bool) {  
    super.viewWillAppear(animated)  
    if let path = myTable.indexPathForSelectedRow {  
        myTable.deselectRowAtIndexPath(path, animated: true)  
    }  
}
```

Do It Yourself: Add the method of Listing 14-17 to the view controller of Listing 14-15. Run the application and select a row. Go back to the Table View. The selected row should be deselected with an animation.

There is a Segue we may create that simplifies our example a little bit. Instead of connecting the main view with the table to the detail view, we connect the prototype cell with the detail view, as shown in Figure 14-18 below. The type of Segue is still Show, but the advantage is that the table triggers the Segue as soon as a cell is selected. Because of this we don't have to call the `performSegueWithIdentifier()` method anymore. We can still implement the `tableView(tableView, didSelectRowAtIndexPath: indexPath)` method to perform any task related to the selection of the cell, but we don't have to trigger the Segue from code

anymore.



Figure 14-18: Show Segue connecting the prototype cell with the detail view

Do It Yourself: Delete the Segue created before between the initial view and the detail view and create a new Show Segue from the prototype cell to the detail view (Figure 14-18). Delete the method `tableView(tableView, didSelectRowAtIndexPath: indexPath)` in the `ViewController` class. Run the application. Everything should work like before.

Adding Rows

Navigation controllers not only provide a natural transition between views but also a bar where we can put buttons to let the user manage the table and the information it contains. A very common feature added to the table from this bar is a button that opens a view with a form to add new items to the list. There is even a predefined style for Bar Button Items called Add that creates a button with a plus sign (Figure 13-31). Figure 14-19 shows the plus button added to the bar on the right and a new main view with an input field to add items. The views are connected with a Show Segue from the plus button to the second view.

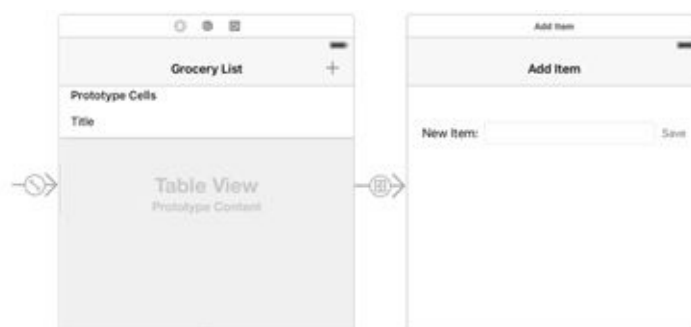


Figure 14-19: Plus button

The addition of new items is very simple. We just have to create a form for the user to insert the data and then store this information in the applications model (the `AppData` structure in our example). For our example we have created a view with a Text Field and a

save button, as shown in Figure 14-19. The view controller class for the view was called **AddItemViewController**. To keep it simple, we have included only a Text Field in the interface to let the user insert the name of the item. The details for that item are filled with placeholders (“noimage” and “Not Defined”). The following is the code for the **AddItemViewController** class.

Listing 14-18: Adding items to the AppData structure

```
import UIKit

class AddItemViewController: UIViewController {
    @IBOutlet weak var newItem: UITextField!

    @IBAction func saveItem(sender: UIButton) {
        var text = newItem.text!
        text = text.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
        if text != "" {
            let lower = text.lowercaseString
            let final = lower.capitalizedString
            AppData.items.append(final)
            AppData.details[final] = ["noimage", "Not Defined"]
            self.navigationController?.pushViewControllerAnimated(true)
        }
    }
}
```

The class includes an Outlet for the Text Field called **newItem** and an Action for the button called **saveItem()**. When the button is pressed, the **saveItem()** method trims the text from the Text Field with the **stringByTrimmingCharactersInSet()** method (see Listing 7-2), and then compares it with an empty string. If the user’s input is not empty, the code capitalizes the string, adds it to the **items** array with the **append()** method, and uses it as a key for the **details** dictionary.

The process adds the item to the model, but the Table View still doesn’t know that there are more items available. One way to tell the table to update its content is calling the **reloadData()** method. In our example, the best moment to call this method is when the table is about to be shown on the screen. The following example updates the **viewWillAppear()** method of Listing 14-17 to reload the data every time the user comes back to the table.

Listing 14-19: Updating the table

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    if let path = myTable.indexPathForSelectedRow {
```

```

        myTable.deselectRowAtIndexPath(path, animated: true)
    }
    myTable.reloadData()
}

```

Do It Yourself: Add a Bar Button Item to the Navigation Bar over the Table View of our previous example. Add a new main view to the Storyboard. Connect the bar button to the new view with a Show Segue. Add a label, a Text Field, and a button to the main view, as shown in Figure 14-19. Create the **AddItemViewController** subclass and assign it to the new view. Connect the Text Field to the **AddItemViewController** class with an Outlet called **newItem**. Complete this view controller with the code of Listing 14-18. Add the method of Listing 14-19 to the **ViewController** class (or update the one you added before). Run the application, press the plus button, insert a text in the Text Field, and press the Save button. The new item should be added to the table.

Another alternative to keep the table up to date is to manage the addition of the new item from the table's view controller. This requires us to assign the **ViewController** object of our example as the delegate of the **AddItemViewController** object, and implement a method in the **ViewController** class to save the new item and update the table. The advantage of this procedure over updating the table from the **viewWillAppear()** method is that now the table won't reload its data every time it is shown on the screen but only when new items are available. The following is the new code for the **ViewController** class.

Listing 14-20: *Storing new items and updating the table from the **ViewController** class*

```

import UIKit

class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {
    @IBOutlet weak var myTable: UITableView!

    override func viewDidLoad() {
        super.viewDidLoad()
        myTable.delegate = self
        myTable.dataSource = self
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        if let path = myTable.indexPathForSelectedRow {
            myTable.deselectRowAtIndexPath(path, animated: true)
        }
        myTable.reloadData()
    }
}

```

```

func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    let data = AppData.items
    return data.count
}

func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let data = AppData.items
    let cell = myTable.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)
    cell.textLabel?.text = data[indexPath.row]
    return cell
}

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showDetails" {
        let controller = segue.destinationViewController as! DetailViewController
        if let path = myTable.indexPathForSelectedRow {
            controller.selected = path.row
        }
    } else if segue.identifier == "showAddItem" {
let controller = segue.destinationViewController as! AddItemViewController
controller.delegate = self
    }
}

func saveItem(title: String) {
    let lower = title.lowercaseString
    let final = lower.capitalizedString
    AppData.items.append(final)
    AppData.details[final] = ["noimage", "Not Defined"]
    navigationController?.popViewControllerAnimated(true)
}
}

```

There are two things we have to do in this controller to manage the updates. We have to declare the **ViewController** class as delegate of the **AddItemViewController** class, and implement a method to save the item and update the table. The delegate is assigned from the **prepareForSegue()** method. In this method we check when the Segue of the plus button is triggered and then assign **self** (the **ViewController** object) to a property of the **AddItemViewController** class we called **delegate** (to be able to recognize the Segue we assigned the identifier "showAddItem" to it from the Attributes Inspector panel).

The method to store the new item was called **saveItem()**. This method receives the string inserted by the user and follows the same procedure as before to store the item and remove the view from the navigation stack, but this time it also calls the **reloadData()** method to update the table.

In the `AddItemViewController` class we have to declare the `delegate` property and call the delegate's `saveItem()` method when the button is pressed. Notice that now in the Action we only trim the text and check whether or not it is empty, but the rest of the process is performed in the `saveItem()` method of the `ViewController` object.

Listing 14-21: *Calling the `saveItem()` method from the `AddItemViewController` class*

```
import UIKit

class AddItemViewController: UIViewController {
    @IBOutlet weak var newItem: UITextField!
    var delegate: ViewController!

    @IBAction func saveItem(sender: UIButton) {
        var text = newItem.text!
        text = text.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
        if text != "" {
            delegate.saveItem(text)
        }
    }
}
```

Do It Yourself: Assign the identifier “showAddItem” to the Segue connected to the plus button. Copy the code of Listing 14-20 inside the `ViewController` class and the code of Listing 14-21 inside the `AddItemViewController` class. Run the application. Everything should work exactly the same, although this time the table is updated from its own view controller. The `ViewController` class of the example of Listing 14-20 includes all the code generated before in Listing 14-15, including the instructions necessary to open the detail view and show the details of every item, but it doesn't call the `performSegueWithIdentifier()` method anymore because it assumes that you have followed previous instructions to create a Show Segue from the prototype cell to the detail view.

The Basics: The use of a delegate object usually requires the creation of a protocol that tells you what properties and methods you have to implement. In cases like this, where the connection is established between only two objects, a protocol is not really necessary.

Deleting Rows

The same way we can add items we can also delete them. The process is as easy as deleting the item from the model and updating the table, but there are infinite possibilities on how to let the user do it. An alternative is to add a button in the Detail view that the user can press to delete the item. In this view we know what item was selected because we got the index of its row from the table, and the user also knows what item is going to be deleted because it is seeing its data on the screen. Figure 14-20 shows the Detail view of our example modified to include a Delete button.

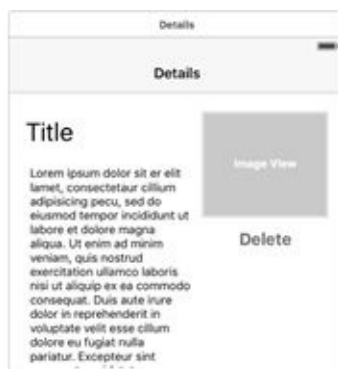


Figure 14-20: Delete button

The Delete button has to be connected to an Action in the view controller of the detail view to remove the item from the model and close the view. The following is the Action we need to add to the **DetailViewController** class to delete the item when the button is pressed.

Listing 14-22: Removing the selected item from the `items` array and the `details` dictionary

```
@IBAction func deleteItem(sender: UIButton) {
    let title = AppData.items[selected]
    AppData.details.removeValueForKey(title)
    AppData.items.removeAtIndex(selected)
    navigationController?.popViewControllerAnimated(true)
}
```

The method of Listing 14-22 reads the value of the selected item, uses this value as a key to remove the corresponding element from the `details` dictionary, and finally deletes the item from the `items` array. When the `popViewControllerAnimated()` method is executed to remove the view from the navigation stack, the view with the table is pushed back to the top of the stack, its `viewWillAppear()` method is executed, and the table is updated with the `reloadData()` method added in Listing 14-19.

Do It Yourself: Add a button called Delete to the detail view of the previous example (Figure 14-20). Connect the button to an Action in the **DetailViewController** called

`deleteItem()`. Complete the method with the code of Listing 14-22. Run the application. Select a row and press the Delete button. The item should be removed from the table.

The Basics: The user should be warned every time the code is going to delete data from the model, especially when the action is performed in response to user interaction. UIKit offers the `UIAlertController` class to create views for this purpose. We will study how to create Alert Views in Chapter 17.

Besides any custom technique we may develop to let the user remove items from the table, Table Views also provide built-in functionality to do it. The table includes an editing mode that when activated shows buttons in every row to delete them and a second button for confirmation.

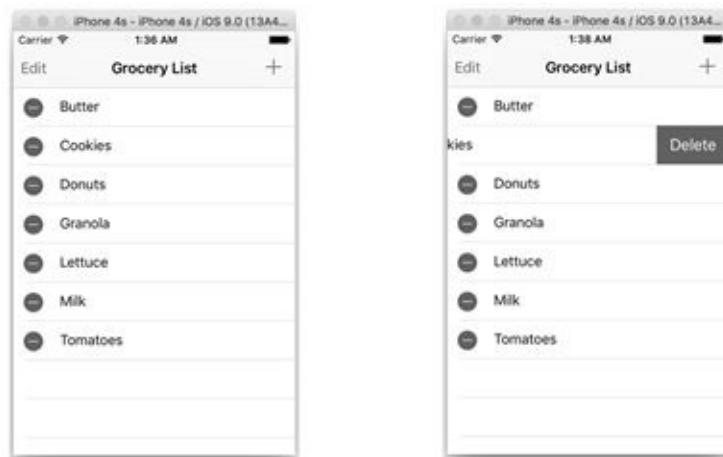


Figure 14-21: Table in editing mode

The process to activate and deactivate this mode involves the `editing` property, the `setEditing()` method (both from the `UITableView` class), and the `tableView(tableView, commitEditingStyle: editingStyle, forRowAtIndexPath: indexPath)` method defined in the `UITableViewDataSource` protocol. We first have to provide a way for the user to decide when to activate or deactivate the mode and then process the feedback. For our example, we have added a Bar Button Item to the Navigation Bar called Edit.

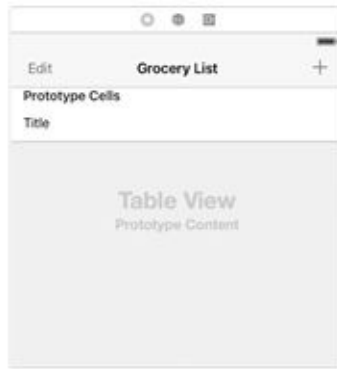


Figure 14-22: Edit button

The button has to activate or deactivate the edition mode depending on its current state. To determine the state we have to read the `editing` property, and to set a new state we have to call the `setEditing()` method.

Listing 14-23: *Activating and deactivating the editing mode*

```
@IBAction func editItems(sender: UIBarButtonItem) {
    if myTable.editing {
        myTable.setEditing(false, animated: true)
    } else {
        myTable.setEditing(true, animated: true)
    }
}
```

Listing 14-23 shows the Action we need to add to the `ViewController` class of our example to take advantage of the table's editing mode. When the user taps on the Edit button of the interface in Figure 14-22, the `editItems()` method checks the value of the `editing` property and sets the mode to `false` if the property is `true` or to `true` if the property is `false`. If the `editing` value is set to `true`, the table activates the editing mode and shows the edition buttons on the rows (Figure 14-21, left). The buttons don't perform any action but displace the row to the left to uncover a Delete button when pressed. This is the button that actually asks the table to delete the row. When the Delete button is pressed, the table calls the `tableView(tableView, commitEditingStyle: editingStyle, forRowAtIndexPath: indexPath)` method to indicate to its delegate that the user wants to delete the row. From this method we have to remove the item from the model and call the Table View's `deleteRowsAtIndexPaths(paths, withRowAnimation: animation)` method to delete the row. The following is the implementation for our example.

Listing 14-24: *Responding to the Delete button*

```
func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle,
forRowAtIndexPath indexPath: NSIndexPath) {
```

```

if editingStyle == UITableViewCellEditingStyle.Delete {
    let row = indexPath.row
    let item = AppData.items[row]
    AppData.details.removeValueForKey(item)
    AppData.items.removeAtIndex(row)

    myTable.deleteRowsAtIndexPaths([indexPath], withRowAnimation: UITableViewRowAnimation.Automatic)
}
}

```

The method of Listing 14-24 removes the item from the model first and then deletes the row. We always have to update the model first or the table won't have enough rows to display the data and will throw an error. Notice that the `deleteRowsAtIndexPaths(paths, withRowAnimation: animation)` method takes an array of `NSIndexPath` objects instead of a single one. This is because the method is capable of deleting several rows at a time.

Do It Yourself: Add a Bar Button Item called Edit to the Navigation Bar at the top of the table's view (Figure 14-22). Connect the button to an Action in the `ViewController` class of our previous example called `editItems()` and complete the method with the code of Listing 14-23. Add the method of Listing 14-24 to the `ViewController` class of Listing 14-20. Run the application and press the Edit button. You should see the edition buttons appear on the left, as shown in Figure 14-21. Press one of these buttons. The row should displace to the left and reveal the Delete button. Press the button to delete the row.

IMPORTANT: If you need to remove or add several rows at a time, the `UITableView` class includes the methods `beginUpdates()` and `endUpdates()` to animate the changes simultaneously. All you have to do is to call the `beginUpdates()` method before the changes are performed and the `endUpdates()` method after everything is done. An example is available in Chapter 22, Listing 22-25.

The Basics: Tables include a hidden feature that simplifies the deletion of rows. Once the `tableView(tableView, commitEditingStyle: editingStyle, forRowAtIndexPath: indexPath)` method has been implemented, the feature is activated and the user can uncover the Delete button by swiping the row to the left. If you think that this feature is enough for your application, you don't have to add an Edit button to activate or deactivate the editing mode, all you have to do is implement the delegate's method.

Moving Rows

The edition mode also allows users to move cells to a different position on the table. This option is presented as an icon on the right side of the cells. To activate it, we have to implement the `tableView(tableView, moveRowAtIndexPath: sourceIndexPath, toIndexPath: destinationIndexPath)` method of the `UITableViewDataSource` protocol. This method is called when a cell is moved to a different position on the table.

The cells are automatically placed in their new position every time the user performs the action, but we have to make sure that the new order of the cells is reflected in the model, so the next time the table loads its content, it shows the cells in the same order the user determined before. The following example shows a possible implementation of the delegate method.

Listing 14-25: Reorganizing the model when a cell is moved

```
func tableView(tableView: UITableView, moveRowAtIndexPath sourceIndexPath: NSIndexPath, toIndexPath
destinationIndexPath: NSIndexPath) {
    let item = AppData.items[sourceIndexPath.row]
    AppData.items.removeAtIndex(sourceIndexPath.row)
    AppData.items.insert(item, atIndex: destinationIndexPath.row)
}
```

In the method of Listing 14-25 we modify the `items` array of our model to change the index of the item that was moved by the user. We first get a reference to the string for the row that is being moved and then remove the item corresponding to that row and add it again in the new position. Figure 14-23 illustrates what the table looks like after the editing mode is activated.

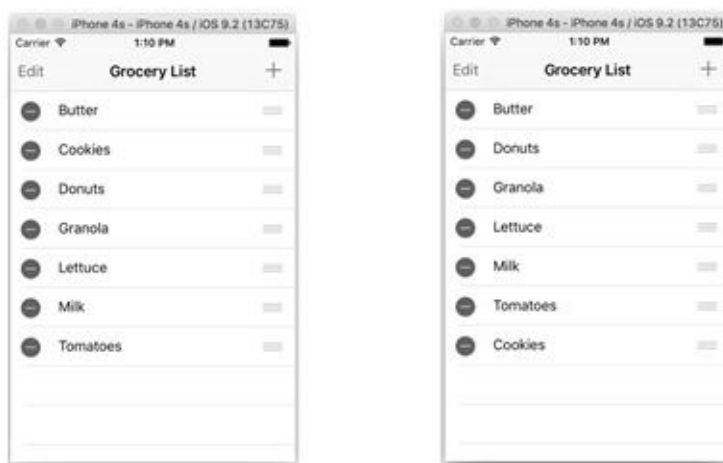


Figure 14-23: Editing mode to move cells

To move a cell, the user has to click on the icon and drag the cell to the new position. In

the example of Figure 14-23 we moved the Cookies cell to the end of the table. When the change is performed, the Table View calls the delegate method of Listing 14-25 and we move the item corresponding to the cell to an index that reflects its new position on the table.

Do It Yourself: Add the method of Listing 14-25 to the `ViewController` class of Listing 14-20. Run the application and press the Edit button. You should see the icons to move the cells as shown in Figure 14-23. Drag the cell to a new position from one of the icons. The table should reorder the cells, as shown in Figure 14-23.

IMPORTANT: The model defined in this chapter automatically organizes the items in alphabetical order (see Listing 14-14). If we want the user to decide the order of the items, we have to use a model that allows it. To test the method of Listing 14-25 you have to delete the `didSet()` method of the `items` property. This makes sure that the model delivers the items in the order they are stored in the array.

14.3 UITableViewController

Table Views that occupy the whole screen are very common in mobile applications. With the intention of simplifying the creation of these tables, the UIKit framework provides a subclass of the `UIViewController` class that includes a full screen table. This view controller presents two advantages. First, the class adopts the Table View protocols by default, so we don't have to declare the adoption on our subclass and define the values of the table's `delegate` and `dataSource` properties. And second, the class offers another type of cells called *Static Cells* that allow us to build static tables (tables that always present the same cells defined in the Storyboard). The Object Library includes the Table View Controller option to add the view controller to the interface.

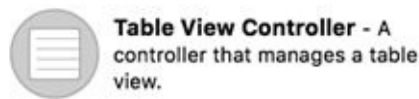


Figure 14-24: Table View Controller option in the Object Library

A Table View Controller looks exactly like a main view with a full screen table inside. The difference is that the Table View is integrated to the main view and we can't change its size.

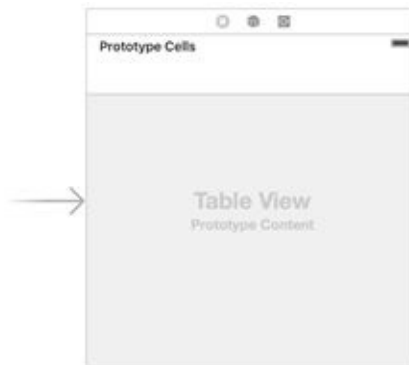


Figure 14-25: Table View Controller in the Storyboard

To manage the Table View we have to create a subclass of the `UITableViewController` class and assign it to the view in the Storyboard. The file is created as any other file with the difference that now we have to select the `UITableViewController` class as the superclass (Figures 13-3 and 13-4). The template created by Xcode offers most of the delegate methods required for a Table View, but the class works like a normal view controller class. We can delete and add the properties and methods we need. The following example shows a subclass of `UITableViewController` called `MyTableViewController` that we have created to manage the interface of Figure 14-25.

Listing 14-26: Managing the table from a subclass of the `UITableViewController` class

```
import UIKit

class MyTableViewController: UITableViewController {
    var items: [String]!

    override func viewDidLoad() {
        super.viewDidLoad()

        items = ["Lettuce", "Tomatoes", "Milk", "Granola", "Donuts", "Cookies", "Butter", "Cheese", "Lemonade",
        "Yogurt", "Oatmeal", "Juice", "Tea", "Coffee", "Bagels", "Coffee Filters", "Brownies", "Potatoes", "Onions"]
    }

    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return items.count
    }

    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
    UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)

        let row = indexPath.row
        cell.textLabel?.text = items[row]

        return cell
    }
}
```

The `UITableViewController` class adopts the delegate protocols by default, but it provides an implementation of the delegate methods, so we have to override those that we want to implement in our subclass. This just means that we have to add the **override** keyword in front of every delegate method we use. The class also includes a property called **tableView** with a reference to the Table View. This saves us from creating the Outlet for the table. The rest of the configuration and codes work exactly the same way as before.

Do It Yourself: Create a new Single View Application project. Delete the main view and add a Table View Controller from the Object Library (Figure 14-24). Select the Scene, open the Attributes Inspector panel and activate the option Is Initial View Controller to designate this view as the initial view. Select the cell, assign the “myCell” identifier to it and change its style to Basic. Create a new file with a subclass of `UITableViewController` called `MyTableViewController`. Complete the class with the code of Listing 14-26. Run the application. You should see a simple table like those created at the beginning of this chapter.

Static Tables

Probably the most interesting feature of Table View Controllers is the possibility to create static tables. Static tables present a fixed number of cells and therefore are particularly useful when we have a limited number of options or pieces of data to display. To turn the table of a Table View Controller into a static table we have to select it and change the Content option in the Attributes Inspector panel to Static Cells, as shown in Figure 14-26.

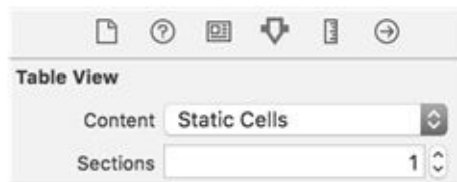


Figure 14-26: Option to change the type of cells

Once the option is changed, the table replaces the prototype cells by three custom empty cells.

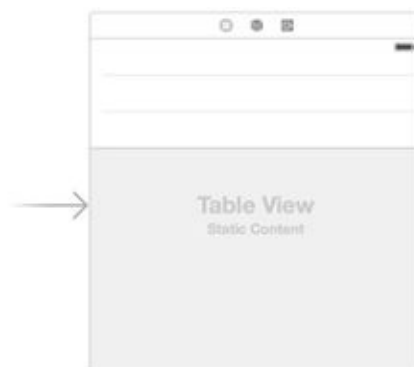


Figure 14-27: Empty static cells

From this point on, the cells and their content are designed as we did before for prototype cells. We may add new cells, delete them, change their size, etc. We can also add more sections modifying the value of the Section option in the Attributes Inspector panel (Figure 14-26). If the main view on the Storyboard is not high enough to fit all the cells we need, we can set a different size from the Simulated Size option in the Size Inspector panel, as we did for Scroll Views in Chapter 11 (Figure 11-13). Figure 14-28 illustrates a possible design. For this example we changed the Table View’s style to Grouped, assign different styles to the cells, change the text on the labels, and added one section with a single cell and an Image View inside.



Figure 14-28: Content of static cells

The Basics: A static table may include standard cells of any style. To create standard cells you have to select the cell, modify the value of the Style option in the Attributes Inspector panel, and double click on the labels to change the text.

The custom cells for a static table don't need their own `UITableViewCell` subclass; we just have to connect the elements directly to Outlets in the view controller. The following is the code for the `UITableViewController` subclass to control the interface of Figure 14-28.

Listing 14-27: Defining the content of static cells

```
import UIKit

class MyTableViewController: UITableViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var placeLabel: UILabel!
    @IBOutlet weak var dateLabel: UILabel!
    @IBOutlet weak var picture: UIImageView!
    override func viewDidLoad() {
        super.viewDidLoad()
        titleLabel.text = "Toronto's Waterfront"
        placeLabel.text = "Toronto"
        let formatter = NSDateFormatter()
        formatter.dateFormat = NSDateFormatterStyle.MediumStyle
        dateLabel.text = formatter.stringFromDate(NSDate())
        picture.image = UIImage(named: "Toronto")
    }
}
```

In the view controller for a static table we don't have to declare the number of rows or define the content for each cell because everything has been defined on the Storyboard, but we can assign new values to the elements on each cell. In Listing 14-27, we assign

some strings to the labels, a date, and an image to the Image View included in the interface of Figure 14-28.

Despite that almost everything is defined in the Storyboard, there are a few delegate methods that are still called by the table to set things from code. For example, we have to implement the `tableView(tableView, titleForHeaderInSection: section)` method if we want to assign titles to the sections' headers. The following code implements this method to assign the title "Description" and "Picture" to the headers of our table. The result is shown in Figure 14-29 below.

Listing 14-28: Changing the titles of the headers

```
import UIKit

class MyTableViewController: UITableViewController {
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var placeLabel: UILabel!
    @IBOutlet weak var dateLabel: UILabel!
    @IBOutlet weak var picture: UIImageView!
    override func viewDidLoad() {
        super.viewDidLoad()
        titleLabel.text = "Toronto's Waterfront"
        placeLabel.text = "Toronto"
        let formatter = NSDateFormatter()
        formatter.dateStyle = NSDateFormatterStyle.MediumStyle
        dateLabel.text = formatter.stringFromDate(NSDate())
        picture.image = UIImage(named: "Toronto")
    }
    override func tableView(tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
        var title = ""
        switch section {
        case 0:
            title = "Description"
        case 1:
            title = "Picture"
        default:
            break
        }
        return title
    }
}
```

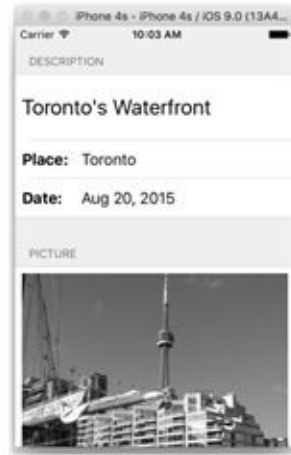


Figure 14-29: *Application with a static table*

Do It Yourself: Create a new Single View Application project. Download the image Toronto.jpg from our website and add it to the Assets Catalog. Delete the main view and add a Table View Controller from the Object Library (Figure 14-24). Select the Scene, open the Attributes Inspector panel and activate the option Is Initial View Controller. Select the table and change the value of the Content option to Static Cells and the value of the Style option to Grouped. Assign the value 2 to Sections to add another section to the table. Modify the sizes of the cells to fit the elements and add labels to the cells of the first section and an Image View to the cell of the second section, as shown in Figure 14-28. Create a subclass of `UITableViewController` called `MyTableViewController` and assign it to the view. Complete the class with the code of Listing 14-28 and connect the elements on the cells to the Outlets in the code. Run the application. You should see something similar to Figure 14-29.

14.4 Search

Table Views are capable of managing thousands of rows and sometimes it can be difficult for users to find what they want. But creating a Text Field with all the functionality necessary to let users search for specific items could be a daunting task. To give us a hand, the UIKit framework includes the `UISearchBar` class. The class includes all the properties and methods necessary to create and manage a Search Bar.

UISearchBar

A Search Bar doesn't perform any actual search, but provides intuitive tools that make it easy for users to search for a value. The Object Library includes an option to add the Search Bar to the Storyboard.

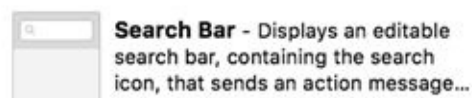


Figure 14-30: Search Bar option in the Object Library

Search Bars may be placed anywhere on the interface, but because of the strong relationship between these bars and Table Views, they are usually included at the top or bottom of tables. We can position the bar manually with constraints when working with Table Views or add them to a Table View Controller and let the class accommodate the bar for us. To add the bar to a Table View Controller we have to drag the option to the controller and drop it at the position we want. Figure 14-31 shows a Table View Controller embedded in a Navigation Controller with a Search Bar at the top.

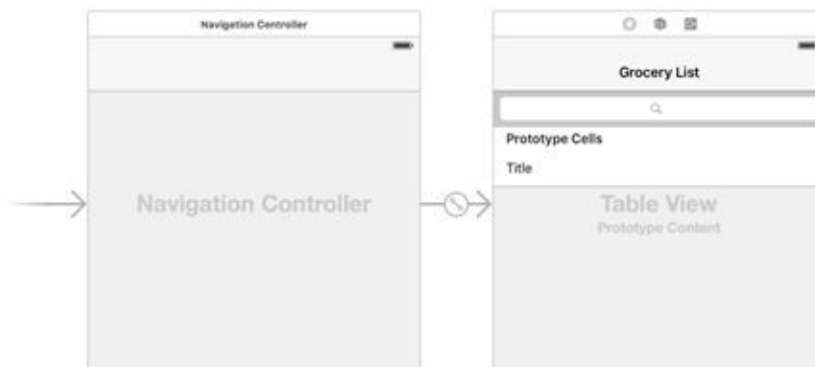


Figure 14-31: Search Bar in a Table View Controller

Once the bar is added to the interface, we can connect it with an Outlet to the Storyboard and access its properties and methods to change its appearance and manage its values.

text—This property sets or returns the text in the field. It is an optional of type `String`.

placeholder—This property sets or returns the field’s placeholder. It is an optional of type `String`.

prompt—This property sets or returns a text that is displayed over the bar. It is an optional of type `String`.

searchBarStyle—This property sets or returns the style of the bar. It is an enumeration of type `UISearchBarStyle` with the values **Default**, **Prominent** (translucent background with opaque search field), and **Minimal** (no background with translucent search field).

barTintColor—This property sets or returns the color of the bar’s background. It is an optional of type `UIColor`.

tintColor—This property sets or returns the color of the bar’s elements. It is an optional of type `UIColor`.

translucent—This property is a Boolean value that determines if the bar is translucent or not.

backgroundImage—This property sets or returns a background image for the bar. It is an optional of type `UIImage`.

showsCancelButton—This property is a Boolean value that determines whether or not the Cancel button is displayed.

showsScopeBar—This property is a Boolean value that determines whether or not the scope bar is displayed. When the value is set to **true**, an additional bar is shown at the bottom of the Search Bar with buttons to select the scope of the search.

scopeButtonTitles—This property sets or returns an array of strings with the title for the buttons of the scope bar.

selectedScopeButtonIndex—This property sets or returns the index of the selected button in the scope bar. It is of type `Int`.

setShowsCancelButton(value, animated: animated)—This method sets the value of the `showsCancelButton` property. The **value** attribute is a Boolean value that determines whether or not the Cancel button is displayed, and the **animated** attribute is another Boolean value that determines if the process is going to be animated.

IMPORTANT: The `UISearchBar` class also adopts the `UITextInputTraits` protocol and implements its properties to configure the keyboard and some features for the Text Field. See `UITextField` in Chapter 7 for more information.

The `UISearchBar` class can designate a delegate object to report the feedback received from the user. The delegate must adopt the `UISearchBarDelegate` protocol and implement its methods.

`searchBar(searchBar, textDidChange: searchText)`—This method is called by the bar on its delegate when the text inside the bar field changes. The **`searchText`** attribute is a `String` value with the current text in the Text Field.

`searchBarTextDidBeginEditing(searchBar)`—This method is called by the bar on its delegate when the edition begins (the user taps on the bar to perform a search).

`searchBarTextDidEndEditing(searchBar)`—This method is called by the bar on its delegate when the edition ends.

`searchBarCancelButtonClicked(searchBar)`—This method is called by the bar on its delegate when the Cancel button is pressed.

`searchBar(searchBar, selectedScopeButtonIndexDidChange: selectedScope)`—This method is called by the bar on its delegate when a button in the scope bar is pressed. The **`selectedScope`** attribute is an integer with the index of the button.

The Search Bar provides the tools for the user to perform the search, but it is up to us to take this input and select the items in our model that match the term. There are different ways to do this. When working with single arrays, the most simple is to use the `filter()` method studied in Chapter 3. The following example creates a model with a method to filter the data and an intermediate property to store the items currently selected.

Listing 14-29: Filtering the data

```
struct ApplicationData {  
    var items: [String]  
    var filteredItems: [String] = []  
    init() {  
        items = ["Lettuce", "Tomatoes", "Milk", "Granola", "Donuts", "Cookies", "Butter", "Cheese", "Lemonade",  
"Yogurt", "Oatmeal", "Juice", "Tea", "Coffee", "Bagels", "Brownies", "Potatoes", "Onions"]  
        filterData("")  
    }  
}
```

```

}

mutating func filterData(search: String) {
    if search == "" {
        filteredItems = items
    } else {
        filteredItems = items.filter({ (item) in
            let value1 = item.lowercaseString
            let value2 = search.lowercaseString
            let valid = value1.hasPrefix(value2)
            return valid
        })
    }

    filteredItems.sortInPlace({ (value1, value2) in value1 < value2 })
}

}

var AppData = ApplicationData()

```

This model is similar to previous examples, but this time we include a method to filter the items called `filterData()` and a second property called `filteredItems` to store the items that match the search. The `filterData()` method receives a string with the value the user is looking for. If the value is an empty string, the `items` property is assigned to the `filteredItems` property to show all the original values, otherwise the `items` array is filtered and only the values that match the search are assigned to `filteredItems`. Because the `filterData()` method is going to be called every time the user inserts or deletes a character, we can't compare entire words, we have to look for values in the `items` array the first letters of which match the characters already inserted by the user. The `hasPrefix()` method introduced in Chapter 3 is perfect for this situation. The method compares two strings and returns `true` if the first string is found at the beginning of the second string (the method distinguishes between lower and uppercase letters, so we have to lowercase both strings before comparing).

A view controller for an example like this is very simple. The way the view controller manages the Table View doesn't change, but now instead of getting the values for the cell from the original array we have to read the property that contains the values selected by the user. The view controller also must adopt the `UISearchBarDelegate` protocol to be able to manage the bar.

Listing 14-30: Processing input from the Search Bar

```

import UIKit

class MyTableViewController: UITableViewController, UISearchBarDelegate {

    @IBOutlet weak var searchItems: UISearchBar!

    override func viewDidLoad() {

```

```

super.viewDidLoad()
searchItems.delegate = self
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return AppData.filteredItems.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)

    let data = AppData.filteredItems[indexPath.row]
    cell.textLabel?.text = data
    return cell
}

func searchBar(searchBar: UISearchBar, textDidChange searchText: String) {
    if let text = searchItems.text {
        let search = text.stringByTrimmingCharactersInSet(.whitespaceAndNewlineCharacterSet())
        AppData.filterData(search)
        tableView.reloadData()
    }
}
}

```

When the user inserts a character in the bar's Text Field, the **searchBar(searchBar, textDidChange: searchText)** method is executed. In this method we get the current text, call the **filterData()** method in the model to search for the value, and update the table. Because **filterData()** stores the results in the **filteredItems** property and the values for the cells are taken from that same property, the results of the search are shown in the table as soon as the user types or deletes a character in the field.



Figure 14-32: Items are filtered with every character inserted in the Text Field

Do It Yourself: Create a new Single View Application project. Remove the main view and add a Table View Controller. Embed the controller in a Navigation Controller. Add a Search Bar at the top of the table (Xcode shows you a blue line between the Navigation Bar and the prototype cell where you can drop the bar). Your interface should look like Figure 14-31. Create the subclass for the Table View Controller and call it **MyTableViewController**. Complete the class with the code of Listing 14-30. Connect the Search Bar with the `searchItems` Outlet. Create a Swift file for the model and copy the code of Listing 14-29 inside. Run the application and perform a search. You should see something similar to Figure 14-32.

At this point, the application lets the user perform a search but it doesn't provide a way to cancel the search or hide the keyboard. We have to show the Cancel button when the user starts typing and hide the keyboard when the button is pressed. The following are the protocol methods we have to add to our view controller for this purpose.

Listing 14-31: Showing and hiding the Cancel button

```
func searchBarTextDidBeginEditing(searchBar: UISearchBar) {
    searchItems.setShowsCancelButton(true, animated: true)
}

func searchBarCancelButtonClicked(searchBar: UISearchBar) {
    searchItems.text = ""
    searchItems.setShowsCancelButton(false, animated: true)
    searchItems.resignFirstResponder()

    AppData.filterData("")
    tableView.reloadData()
}
```

The first method, `searchBarTextDidBeginEditing()`, is executed when the user taps on the bar to perform a search. From this method we have to show the Cancel button to give the user the chance to cancel the operation. The second method, `searchBarCancelButtonClicked()`, is executed when the button is pressed. Here we have to clean the search field, hide the Cancel button, resign first responder to close the keyboard (see Chapter 7, Listing 7-13), restore the original items by calling the `filterData()` method with an empty string, and update the table.

A nice effect, widely used these days, involves hiding the Navigation Bar when the edition starts and showing it again when the operation is cancelled. This may be achieved by executing the `setNavigationBarHidden()` method of the `UINavigationController` object.

Listing 14-32: Hiding the Navigation Bar

```

func searchBarTextDidBeginEditing(searchBar: UISearchBar) {
    searchItems.setShowsCancelButton(true, animated: true)
    navigationController?.setNavigationBarHidden(true, animated: true)
}

func searchBarCancelButtonClicked(searchBar: UISearchBar) {
    searchItems.text = ""
    searchItems.setShowsCancelButton(false, animated: true)
    searchItems.resignFirstResponder()
    AppData.filterData("")
    tableView.reloadData()
    navigationController?.setNavigationBarHidden(false, animated: true)
}

```

Do It Yourself: Add the methods in Listing 14-32 to the view controller of Listing 14-30. Run the application. You should see the Cancel button in the bar. Press the button. The Search Bar should be cleaned and all the elements go back to normal.

A Search Bar may include an additional bar at the bottom with selectable buttons called *Scope Bar*. The buttons work in a similar way as a Segmented Control (Figure 4-11), when a button is selected the other is automatically deselected. The bar assigns an index to every button and then, every time a button is selected, it calls the `searchBar(searchBar, selectedScopeButtonIndexDidChange: selectedScope)` method with the value of its index. This value may be used to refine the search. For example, we could look for the value in the items' titles or descriptions, depending on the button selected. To add a Scope Bar we have to select the Search Bar and activate the option Shows Scope Bar in the Attributes Inspector panel.

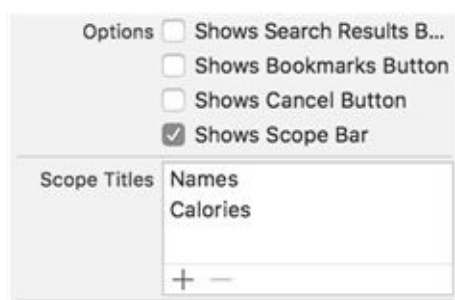


Figure 14-33: Configuration of the Scope Bar

The buttons are added from the box called Scope Titles. We have to press the plus button to add a new title and then press Enter to change its text. For our example we have created two buttons: Names and Calories. Figure 14-34 shows how the buttons look in the simulator.

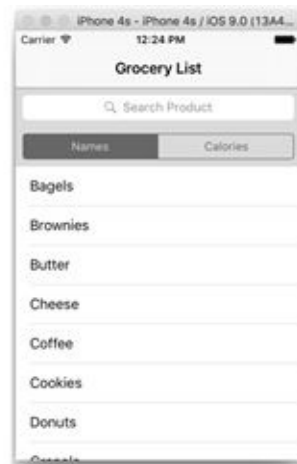


Figure 14-34: Scope Bar in the simulator

The following is an update to the previous model that includes a dictionary with the calories for every item.

Listing 14-33: *Expanding the model*

```
struct ApplicationData {
    var items: [String]
    var calories: [String: Int]
    var selectedButton: Int
    var filteredItems: [String] = []
    init() {
        items = ["Lettuce", "Tomatoes", "Milk", "Granola", "Donuts", "Cookies", "Butter", "Cheese", "Lemonade",
        "Yogurt", "Oatmeal", "Juice", "Tea", "Coffee", "Bagels", "Brownies", "Potatoes", "Onions"]
        calories = ["Lettuce": 15, "Tomatoes": 18, "Milk": 42, "Granola": 471, "Donuts": 452, "Cookies": 502, "Butter":
        717, "Cheese": 402, "Lemonade": 40, "Yogurt": 59, "Oatmeal": 68, "Juice": 23, "Tea": 1, "Coffee": 0, "Bagels": 250,
        "Brownies": 466, "Potatoes": 77, "Onions": 40]
        selectedButton = 0
        filterData("")
    }
    mutating func filterData(search: String) {
        if search == "" {
            filteredItems = items
        } else {
            filteredItems = items.filter({ (item) in
                var valid = false
                if selectedButton == 0 {
                    let value1 = item.lowercaseString
                    let value2 = search.lowercaseString
                    valid = value1.hasPrefix(value2)
                }
            })
        }
    }
}
```



```

    } else if let maximum = Int(search) {
    let caloriesItem = calories[item]
    if caloriesItem < maximum {
    valid = true
    }
    }
    return valid
    })
    }
    filteredItems.sortInPlace({ (value1, value2) in value1 < value2 })
    }
    }
    var AppData = ApplicationData()

```

There are two new properties in this model: **calories** and **selectedButton**. The **calories** property is a dictionary with the names of the items as keys and their calories as values, and the **selectedButton** property keeps track of the button selected at the moment. The **filter()** method filters the **items** array according to the value of this property. If the button with the index 0 is selected, **filter()** searches for the string in the items' names, otherwise it turns the string into an integer and looks for the items in the dictionary with a number of calories lower than the current value inserted by the user.

The view controller has to implement the **searchBar(searchBar, selectedScopeButtonIndexDidChange: selectedScope)** method to tell the model what is the button currently selected.

Listing 14-34: Reporting to the model the button selected

```

func searchBar(searchBar: UISearchBar, selectedScopeButtonIndexDidChange selectedScope: Int) {
    AppData.selectedButton = selectedScope
    if selectedScope == 0 {
        searchItems.placeholder = "Search Product"
    } else {
        searchItems.placeholder = "Maximum Calories"
    }
    searchItems.text = ""
    AppData.filterData("")
    tableView.reloadData()
}

```

The method is called every time the user presses a button. All we have to do to let the model know what button is currently selected is to assign the value received from the

`selectedScope` attribute to our `selectedButton` property. From this moment on, the `filter()` method looks for the right values in the model, but we take advantage of the delegate method to also change the placeholder text to give a hint to the user of what kind of value the Search Bar is expecting, clean the search field, invalidate the previous search by calling the `filterData()` method with an empty string, and update the table to take the interface back to its initial state.

IMPORTANT: UIKit includes the `UISearchController` class to create a view controller that manages the Search Bar and a separate view where the search is performed. The topic is beyond the scope of this book. For more information visit our website and follow the links for this chapter.

Chapter 15

Collection Views

15.1 A Collection of Views

With the introduction of the iPad and its larger screen it became evident for Apple that the single column layout provided by Table Views was not enough to create good user interfaces. But adding more columns to the table wouldn't satisfy these new demands either. Developers needed a flexible way to organize data on the screen and that is how Collection Views were born. Collection Views take a list of views (cells) and present them on the screen using a predefined layout. They come with a grid-like layout by default, but we can customize it to present the views any way we want.



Figure 15-1: Collection Views

UICollectionView

Collection Views are created the same way as tables. There is a class to create the view, a class for cells, and delegates to provide the data and adjust the configuration. The class for the view is called `UICollectionView` and the following are its most frequently used properties and methods.

allowsSelection—This property is a Boolean value that determines whether the selection of cells is allowed or not.

allowsMultipleSelection—This property is a Boolean value that determines whether multiple selection of cells is allowed or not.

backgroundView—This property sets or returns the view for the background of the cell. It is an optional value of type `UIView`.

collectionViewLayout—This property sets or returns the layout object in charge of

setting the layout for the cells. It is an object of a subclass of the `UICollectionViewLayout` class.

`dequeueReusableCellWithReuseIdentifier(identifier, forIndexPath: path)`—This method returns a `UICollectionViewCell` object to represent a cell. The object is created from the prototype with the identifier defined by the **`identifier`** attribute.

`dequeueReusableSupplementaryViewOfKind(kind, withReuseIdentifier: identifier, forIndexPath: path)`—This method returns a `UICollectionViewReusableView` object to represent additional views like headers and footers. The object is created from the prototype view with the identifier defined by the **`identifier`** attribute. The **`kind`** attribute is a constant that represents the kind of views allowed by the layout object. The object provided by default for the Collection View includes two constants: `UICollectionViewElementKindSectionHeader` and `UICollectionViewElementKindSectionFooter`.

`reloadData()`—This methods forces the Collection View to update the cells.

`insertItemsAtIndexPaths(path)`—This method inserts new cells in the positions indicated by the **`path`** attribute. The attribute is an array of `NSIndexPath` objects.

`deleteItemsAtIndexPaths(path)`—This method deletes the cells in the positions indicated by the **`path`** attribute. The attribute is an array of `NSIndexPath` objects.

`selectItemAtIndexPath(path, animated: animated, scrollPosition: position)`—This method selects the cell at the position indicated by the **`path`** attribute. The **`animated`** attribute is a Boolean value that indicates if the process is going to be animated, and the **`position`** attribute is a constant of the `UICollectionViewScrollPosition` structure to indicate how the Collection View is going to scroll to show the cell on the screen. The constants available are `None`, `Top`, `Bottom`, `Left`, `Right`, `CenteredVertically`, and `CenteredHorizontally`.

`deselectItemAtIndexPath(path, animated: animated)`—This method deselects the cell at the position indicated by the **`path`** attribute. The **`animated`** attribute is a Boolean value that indicates if the process is going to be animated.

`scrollToItemAtIndexPath(path, atScrollPosition: position, animated: animated)`—This method scrolls the Collection View to show on the screen the cell at the position indicated by the **`path`** attribute. The **`animated`** attribute is a Boolean value that indicates if the process is going to be animated, and the **`position`** attribute is a constant of the `UICollectionViewScrollPosition` structure to indicate how the Collection View is going to scroll. The constants available are `None`, `Top`, `Bottom`, `Left`, `Right`, `CenteredVertically`, and `CenteredHorizontally`.

`indexPathForCell(cell)`—This method returns an `NSIndexPath` object indicating the position of the cell referenced by the **`cell`** attribute. The attribute is a `UICollectionViewCell` object.

cellForItemAtIndexPath(path)—This method returns a reference to the `UICollectionViewCell` object that represents the cell at the position indicated by the **path** attribute. The attribute is an `NSIndexPath` object.

numberOfSections()—This method returns an integer with the number of sections available.

numberOfItemsInSection(section)—This method returns an integer with the number of cells in the section specified by the **section** attribute. The attribute is an `Int` value with the index of the section.

UICollectionViewCell

Collection View's cells don't offer standard styles and therefore don't have predefined elements to display the data. They are empty cells that we have to customize as we did before for custom table cells. The cells are created from the `UICollectionViewCell` class and contain three empty views to manage their content: a view for the content, a view for the background, and a second background view that is shown when the cell is selected. The following are the properties to access these views.

contentView—This property returns the content view of the cell. All the elements added to customize the cell are incorporated as subviews of this view.

backgroundView—This property sets or returns the view for the background of the cell. It is an optional of type `UIView`.

selectedBackgroundView—This property sets or returns the view for the background when the cell is selected. It is an implicitly unwrapped optional of type `UIView`.

UICollectionViewFlowLayout

The big difference between Table Views and Collection Views is that the positions of the cells in a Collection View are not determined by the view, they are set by a layout object that works along with the Collection View to present the cells on the screen. This object is created from a subclass of the `UICollectionViewLayout` class. Although we can create our own subclass to provide any layout we want, Collection Views include by default a subclass called `UICollectionViewFlowLayout` that provides a very customizable grid-like layout that is usually more than enough for our projects. The layout is called *Flow*, and it includes the following properties for configuration.

scrollDirection—This property sets or returns the direction of the scroll. It is an enumeration of type `UICollectionViewScrollDirection` with the values `Vertical` and `Horizontal` (with `Vertical` set by default).

minimumInteritemSpacing—This property sets or returns a `CGFloat` value that determines the space between cells. In vertical scrolling the value determines the space between cells in the same row, but in horizontal scrolling it determines the space between items in the same column.

minimumLineSpacing—This property sets or returns a `CGFloat` value that determines the space between cells. In vertical scrolling the value determines the space between rows, but in horizontal scrolling it determines the space between columns.

sectionInset—This property sets or returns the margins for the sections. Its value is a structure of type `UIEdgeInsets`.

itemSize—This property sets or returns the size for the cells. It is a value of type `CGSize`.

estimatedItemSize—This property sets or returns the estimated size for the cells. It is a value of type `CGSize` used to improve performance.

Collection View Protocols

As Table Views, Collection Views designate delegate objects to get the information they have to display and respond to user interaction. UIKit defines two protocols for Collection Views: `UICollectionViewDelegate` and `UICollectionViewDataSource`. The following are the most frequently implemented methods of the `UICollectionViewDelegate` protocol.

collectionView(collectionView, didSelectItemAtIndexPath: indexPath)—This method is called by the Collection View on its delegate when a cell is selected. The **indexPath** attribute is an `NSIndexPath` object with the position of the cell.

collectionView(collectionView, didDeselectItemAtIndexPath: indexPath)—This method is called by the Collection View on its delegate when a cell is deselected. The **indexPath** attribute is an `NSIndexPath` object with the position of the cell.

collectionView(collectionView, willDisplayCell: cell, forItemAtIndexPath: indexPath)—This method is called by the Collection View on its delegate when a cell is going to be displayed. The **cell** attribute is a reference to the `UICollectionViewCell` object that represents the cell, and the **indexPath** attribute is an `NSIndexPath` object with its position.

collectionView(collectionView, willDisplaySupplementaryView: view, forElementKind: elementKind, atIndexPath: indexPath)—This method is called by the Collection View on its delegate when a supplementary view is going to be displayed. Supplementary views are views added to the Collection View to present additional information, such as the views used to create headers and footers. The **view** attribute is a reference to the `UICollectionViewReusableView` object that represents the view, the **elementKind** attribute is a constant that determines the type of the view (`UICollectionViewElementKindSectionHeader` and `UICollectionViewElementKindSectionFooter` for the flow layout included by default), and the **indexPath** attribute is an `NSIndexPath` object with its position.

The `UICollectionViewDataSource` protocol defines the following methods.

collectionView(collectionView, numberOfItemsInSection: section)—This method is called by the Collection View on its delegate when it needs to know the number of cells in a section. It returns an integer with the number of cells. The **section** attribute is an integer with the index of the section.

numberOfSectionsInCollectionView(collectionView)—This method is called by the Collection View on its delegate when it needs to know the number of sections in the Collection View. It returns an integer with the number of sections.

collectionView(collectionView, cellForItemAtIndexPath: indexPath)—This method is called by the Collection View on its delegate when it needs the `UICollectionViewCell` object for a cell. The **indexPath** attribute is an `NSIndexPath` object with the location of the cell.

collectionView(collectionView, viewForSupplementaryElementOfKind: kind, atIndexPath: indexPath)—This method is called by the Collection View on its delegate when it needs the `UICollectionViewReusableView` object for a supplementary view. The **kind** attribute is a constant that determines the type of view (`UICollectionViewElementKindSectionHeader` and `UICollectionViewElementKindSectionFooter` for the flow layout included by default), and the **indexPath** attribute is an `NSIndexPath` object with the location of the view.

Layout Protocol

Flow layout can also designate a delegate to get specific values for individual cells. The methods are defined in the `UICollectionViewDelegateFlowLayout` protocol. The following are the most frequently used.

collectionView(collectionView, layout: collectionViewLayout, sizeForItemAtIndexPath: indexPath)—This method is called by the layout object to get the size of a cell. It must return a **CGSize** value with the size of the cell at the location indicated by the **indexPath** attribute.

collectionView(collectionView, layout: collectionViewLayout, insetForSectionAtIndex: section)—This method is called by the layout object to get the **UIEdgeInsets** structure with the inset values (the margins) for the section. The **section** attribute is an integer with the section's index.

collectionView(collectionView, layout: collectionViewLayout, minimumLineSpacingForSectionAtIndex: section)—This method is called by the layout object to get the **CGFloat** value that determines the minimum space between lines in a section. The **section** attribute is an integer with the section's index.

collectionView(collectionView, layout: collectionViewLayout, minimumInteritemSpacingForSectionAtIndex: section)—This method is called by the layout object to get the **CGFloat** value that determines the minimum space between cells in a section. The **section** attribute is an integer with the section's index.

Implementing Collection Views

Collection Views are like any other view; they may be created from code or added to the main view in the Storyboard from the option in the Object Library.

UICollectionView(frame: frame, layout: layout)—This initializer creates a **UICollectionView** object with the frame specified by the **frame** attribute and the layout object provided by the **layout** attribute. The object for the layout is created from a subclass of **UICollectionViewLayout**. We may specify our own layout object or assign the object by default (**UICollectionViewFlowLayout()**).

The Object Library includes the Collection View option to add a Collection View to the Storyboard.

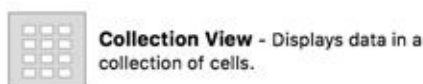


Figure 15-2: Collection View option in the Object Library

Cells may also be created from code or added to the Storyboard from the option in the Object Library, but this is usually not necessary because Collection Views added to the Storyboard already include a prototype cell. Figure 15-3 shows a Collection View pinned to the edges of the main view (the square in the top-left corner is the cell).

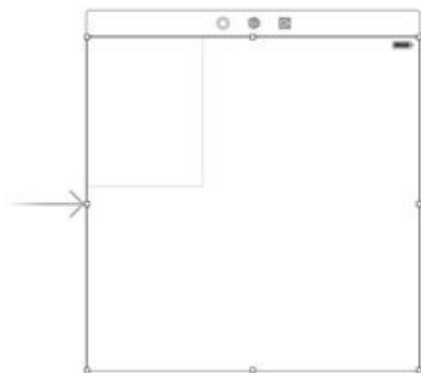


Figure 15-3: *Collection View with a prototype cell*

Collection Views don't include any standard cell, so we have to customize the prototype cell to be able to show content. The first step for customization is to give the cell the size we want. The size defined for the cell on the Storyboard is the size used by the Collection View to present the cell unless we change it from the layout object. To define it we can select the cell and drag the little squares around it to the position we want or specify the precise values from the Size Inspector panel. When we select the Collection View, the Size Inspector panel shows a list of options to define the size and aspects of several elements inside the view, including the header, the footer, and the cell. Figure 15-4 shows the top of the panel with some of the options available.

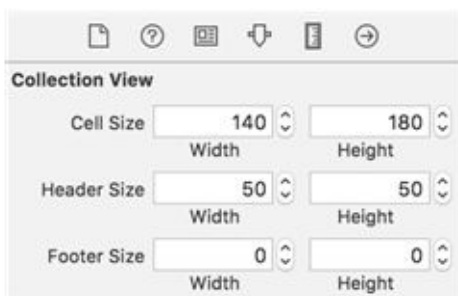


Figure 15-4: *Option to configure the elements in the Collection View*

Once we set the size we want, we have to start adding elements to the cell. Figure 15-5 shows a cell with an Image View pinned to its margins.

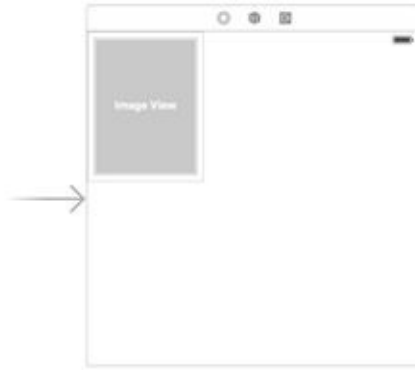


Figure 15-5: Custom cell with an Image View

Because Collection View’s cells are custom cells, we have to define our own subclass of `UICollectionViewCell` to manage their content. The following code defines a subclass called `BookCell` with an Outlet called `bookCover` to manage the cell of Figure 15-5.

Listing 15-1: *Creating a subclass for a Collection View cell*

```
import UIKit

class BookCell: UICollectionViewCell {
    @IBOutlet weak var bookCover: UIImageView!
}
```

As always, the model depends on the characteristics of our application. As an example, we are going to use a model that includes a list of books and the images of their covers. We use an array called `items` for the names of the files and a dictionary called `itemsData` for the titles and authors.

Listing 15-2: *Creating the model for a Collection View*

```
struct ApplicationData {
    var items: [String]
    var itemsData: [String: [String]]
    init() {
        items = ["book1", "book2", "book3", "book4", "book5", "book6", "book7", "book8", "book9", "book10",
"book11", "book12"]
        itemsData = ["book1": ["Steve Jobs", "Walter Isaacson"],
"book2": ["HTML5 for Masterminds", "J.D Gauchat"],
"book3": ["The Road Ahead", "Bill Gates"],
"book4": ["The C Programming Language", "Brian W. Kernighan"],
"book5": ["Being Digital", "Nicholas Negroponte"],
"book6": ["Only the Paranoid Survive", "Andrew S. Grove"],
"book7": ["Accidental Empires", "Robert X. Cringely"],
```

```

        "book8": ["Bobby Fischer Teaches Chess", "Bobby Fischer"],
        "book9": ["New Guide to Science", "Isaac Asimov"],
        "book10": ["Christine", "Stephen King"],
        "book11": ["IT", "Stephen King"],
        "book12": ["Ending Aging", "Aubrey de Grey"]]
    }
}

var AppData = ApplicationData()

```

The view controller that manages the Collection View has to adopt at least the **UICollectionViewDataSource** protocol and implement two of its methods to report the number of cells to display and provide the **UICollectionViewCell** objects for every cell.

Listing 15-3: Managing a Collection View

```

import UIKit

class ViewController: UIViewController, UICollectionViewDataSource {
    @IBOutlet weak var collectionView: UICollectionView!
    override func viewDidLoad() {
        super.viewDidLoad()
        collectionView.dataSource = self
        collectionView.backgroundColor = UIColor.whiteColor()
    }
    func collectionView(collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return AppData.items.count
    }
    func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) ->
    UICollectionViewCell {
        let cell = collectionView.dequeueReusableCellWithReuseIdentifier("myCell", forIndexPath: indexPath) as!
        BookCell
        let file = AppData.items[indexPath.item]
        cell.bookCover.image = UIImage(named: file)
        return cell
    }
}

```

The process to configure a Collection View is very similar to the one we used for tables. We have to implement the delegate methods to report the number of cells in the section (only one by default), and provide the cells. We also have to get the prototype cell calling the **dequeueReusableCellWithReuseIdentifier(identifier, forIndexPath: path)** with the identifier assigned to the cell from the Attributes Inspector panel ("myCell" in this example), cast it

as our subclass (**BookCell** in this example), and then configure the elements in the cell. There is only one little detail that differs from tables. The **NSIndexPath** object includes three properties: **section**, **row**, and **item**. We used the **section** and **row** properties before to identify the sections and each cell on a Table View, but Collection Views consider a cell to be an item, not an entire row, and therefore they use the **section** and **item** properties instead.

When we add a Collection View to the Storyboard, the system creates a **UICollectionViewFlowLayout** object and assigns it to the **collectionViewLayout** property of the Collection View's object. This layout object is configured by default to position the cells in a grid. The result is shown in Figure 15-6.

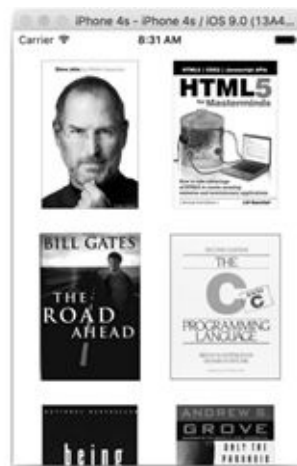


Figure 15-6: Standard layout

Do It Yourself: Create a new Single View Application project. Add a Collection View to the main view and pin it to the sides of the view (Figure 15-3). Connect the Collection View to the view controller with an Outlet called **collectionItems**. Select the Collection View and open the Size Inspector panel. Change the size of the cell to 140 x 180 (Figure 15-4). Select the cell and give it the identifier “myCell”. Drag an Image View to the cell, pin it to its margins with Space constraints, and select its aspect mode as Aspect Fit. Create a subclass of **UICollectionViewCell** with the name **BookCell**. Open the Identity Inspector panel and assign the class to the cell (Figures 13-6 and 13-7). Connect the Image View to the **BookCell** class with an Outlet called **bookCover** (Listing 15-1). Create a Swift file for the model of Listing 15-2. Complete the **ViewController** class with the code of Listing 15-3. Run the application. You should see something similar to Figure 15-6.

IMPORTANT: The same way it does for Table Views, the UIKit framework also includes a subclass of the **UIViewController** class to manage a full screen Collection View. The subclass is called **UICollectionViewController** and is represented in the Storyboard by a main view created from an option on the Object Library called *Collection View Controller*.

The Basics: Sections in Collection Views are managed the same way as in tables. You have to provide the data for every section and implement the protocol methods as we did in Listing 14-4.

Scroll Direction

Flow layout (the layout object assigned by default to the Collection View) offers two directions for the scrolling: Horizontal and Vertical. When the Horizontal direction is selected, cells are placed from left to right according to the space available. The first cell is positioned on the left, then the next on its right and so on. When the next cell doesn't fit anymore in the space available in that row, the object creates a new one and starts adding cells from the left side again. The same process is performed for the Vertical scroll, although the flow goes from top to bottom. The Attributes Inspector panel includes an option called *Scroll Direction* to select the direction for the layout (Figure 15-7, number 1).

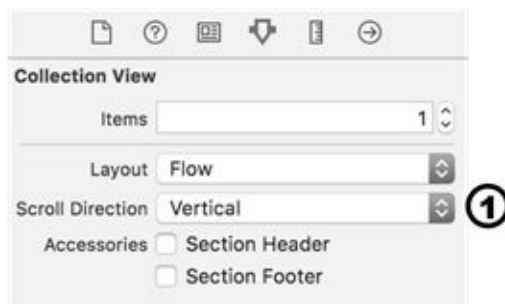


Figure 15-7: Flow layout options

The amount of columns and rows generated by the layout always depends on the space available, the size of the cells, and the scrolling direction selected. For example, with a Vertical scroll, our example will only be able to fit one row of cells on some devices in landscape mode.



Figure 15-8: Horizontal scrolling in landscape mode

Selection

Collection View's cells show their selected background view when they are selected. The property containing this view is set to `nil` by default, so it is our responsibility to provide the view and let the user see it. In the following example we assign a new `UIView` object to the `selectedBackgroundView` property of each cell as soon as the cell is created.

Listing 15-4: Adding the selected background view to the cell

```
import UIKit

class BookCell: UICollectionViewCell {
    @IBOutlet weak var bookCover: UIImageView!
    override func awakeFromNib() {
        super.awakeFromNib()

        selectedBackgroundView = UIView()
        selectedBackgroundView?.backgroundColor = UIColor(red: 0.9, green: 0.9, blue: 0.9, alpha: 1)
    }
}
```

The `awakeFromNib()` method is called on the `UICollectionViewCell` subclass after the prototype cell is loaded from the Storyboard. This is a good place to prepare and configure the cell. In Listing 15-4 we assign a new `UIView` object to the cell's `selectedBackgroundView` property and give it a gray color. Notice that we don't need to declare the frame for the `UIView` object. The views added to background views are automatically expanded to occupy the space available.

Background views are positioned behind the content, so we have to make sure that the content leaves some room for the user to see them. This is why we pinned the Image View of our example to the margins of the cell and not to the edges (Figure 15-5). When a cell is selected, the view is visible around the book cover, as shown In Figure 15-9.



Figure 15-9: *HTML5 for Masterminds is selected*

As well as Table Views, Collection Views call a method on the delegate to let our code know that a cell was selected. The method is called `collectionView(collectionView, didSelectItemAtIndexPath: indexPath)`. But we don't need to implement this method if we want to trigger a Segue. As mentioned in the previous chapter, if the Segue is created from the cell to the view, it is triggered when the cell is selected. Of course, to take advantage of this we have to create a full interface with all the necessary views and navigation tools. Figure 15-10 illustrates a possible interface for our example. The Collection View was embedded in a Navigation Controller and the prototype cell was connected to a detail view with a Show Segue.

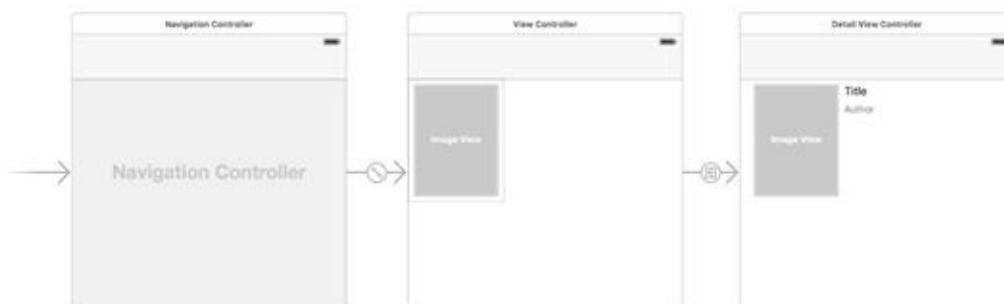


Figure 15-10: *Collection View in a Navigation Controller*

The view controller for the Collection View has to include the `prepareForSegue()` method to send the index of the selected cell to the detail view. The difference with Table Views is that the only method provided by Collection Views to get the path for the selected cell returns an array with all the cells selected. Because we are only working with single selection, we have to retrieve the first element of this array and read its `item` property.

Listing 15-5: *Sending the index of the selected cell to the detail view*

```

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?){
    if segue.identifier == "showDetails" {
        let controller = segue.destinationViewController as! DetailViewController
        if let paths = collectionItems.indexPathsForSelectedItems() {
            let selectedCell = paths[0].item
        }
    }
}
  
```

```

        controller.selected = selectedCell
    }
}
}

```

The controller for the detail view receives the index of the selected cell and gets the information from the model to configure the elements on the screen. For this example we called the view controller **DetailViewController** and defined three Outlets to show the cover, the title, and the author of the book on the screen.

Listing 15-6: Getting the values from the model and setting the detail view

```

import UIKit

class DetailViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    var selected: Int!

    override func viewDidLoad() {
        super.viewDidLoad()

        let file = AppData.items[selected]
        bookCover.image = UIImage(named: file)
        if let data = AppData.itemsData[file] {
            bookTitle.text = data[0]
            bookAuthor.text = data[1]
        }
    }
}

```

When a cell is selected, the Segue is triggered and the view controller calls the **prepareForSegue()** method. This method gets the path of the selected cell and assigns its index to the **selected** property of the **DetailViewController** object. Using this index the view controller gets the information from the model and shows it on the screen.



Figure 15-11: Detail view for a Collection View

Do It Yourself: Embed the initial view of the previous example in a Navigation Controller. Add another view to represent the detail view. Connect the prototype cell to the detail view with a Show Segue (Figure 14-18). Assign the identifier “showDetails” to the Segue. Create a view controller for the new view called **DetailViewController**. Add an Image View and two labels to the detail view, as shown In Figure 15-10. Connect the elements to the Outlets shown In Listing 15-6. Complete the **DetailViewController** with the code of Listing 15-6. Add the **prepareForSegue()** method of Listing 15-5 to the **ViewController** class of Listing 15-3. Add the **awakeFromNib()** method of Listing 15-4 to the **BookCell** class. Run the application and select a book to open the detail view.

Again, we run into the problem that when we get back to the Collection View, the cell is still selected. As we did before, we can implement the **viewWillAppear()** method on the **ViewController** class and deselect the selected cell. Because the Collection View only provides a method that returns an array of cells, we have to iterate through the array to deselect each cell.

Listing 15-7: Deselecting cells

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    if let paths = collectionItems.indexPathsForSelectedItems() {
        for path in paths {
            collectionItems.deselectItemAtIndexPath(path, animated: true)
        }
    }
}
```

Do It Yourself: Add the method In Listing 15-7 to the **ViewController** class of Listing

15-3. Run the application, select a book, and go back to the Collection View. The book should be deselected.

Supplementary Views

Besides cells, Collection Views use other views to include additional content such as headers and footers. These views are called *Supplementary Views* and are managed by the layout object. Flow layout includes two supplementary views, one for the header and another for the footer of every section. The options to add these views to the Collection View In the Storyboard are available in the Attributes Inspector panel (Figure 15-7, below the Scroll Direction option). Once any of these options is selected, the corresponding view is added to the Collection View. The views appear at the top and bottom of the prototype cell when the scroll direction is set to Vertical, and on the left and right when the scroll direction is set to Horizontal.



Figure 15-12: Header and Footer in a vertical layout

Like cells, supplementary views added to the Storyboard are just prototypes. They are used as templates to create the headers and footers for every section. To set the size for these views we have to select the Collection View and modify their values from the Size Inspector panel (Figure 15-4). The height defined In this panel is considered when the scroll direction is Vertical, and the width is considered when the scroll direction is Horizontal. For our example, we added an Image View and a label to the header, and an Image View to the footer.

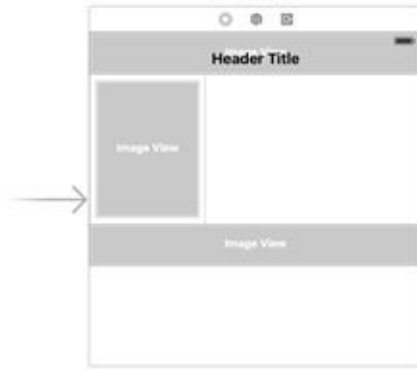


Figure 15-13: *Customs header and footer*

Supplementary views, as cells, are reusable. For this reason, they are not created from the `UIView` class but from a subclass called `UICollectionReusableView`. To connect the elements added to each view with our code, we have to create a subclass of the `UICollectionReusableView` class for every view, assign it to the view, and then create the Outlets. This is the same process as for cells, but instead of the `UICollectionViewController` class we use the `UICollectionReusableView` class as the superview. The following is the subclass we have created for the header of our example. It is called **HeaderView** and it contains an Outlet for the label called **headerTitle** and an Outlet for the Image View called **headerImage**.

Listing 15-8: *Creating a subclass for the header view*

```
import UIKit

class HeaderView: UICollectionReusableView {
    @IBOutlet weak var headerTitle: UILabel!
    @IBOutlet weak var headerImage: UIImageView!
}
```

The footer view also requires a subclass. This one was called **FooterView** and it contains an Outlet called **footerImage** for the Image View.

Listing 15-9: *Creating a subclass for the footer view*

```
import UIKit

class FooterView: UICollectionReusableView {
    @IBOutlet weak var footerImage: UIImageView!
}
```

When it is time to display the header or the footer of a section, the Collection View calls the `collectionView(collectionView, viewForSupplementaryElementOfKind: kind, atIndexPath: indexPath)`

method on its delegate. From this method we have to get the instance of the supplementary view for the section, assign the values to its elements, and return it.

Listing 15-10: Creating the supplementary views

```
func collectionView(collectionView: UICollectionView, viewForSupplementaryElementOfKind kind: String,
atIndexPath indexPath: NSIndexPath) -> UICollectionViewReusableView {
    switch kind {
        case UICollectionViewElementKindSectionHeader:
            let headerView =
collectionView.dequeueReusableSupplementaryViewOfKind(UICollectionViewElementKindSectionHeader,
withReuseIdentifier: “myHeader”, forIndexPath: indexPath) as! HeaderView
            headerView.headerTitle.text = “My Books”
            headerView.headerImage.image = UIImage(named: “gradientTop”)
            return headerView
        case UICollectionViewElementKindSectionFooter:
            let footerView =
collectionView.dequeueReusableSupplementaryViewOfKind(UICollectionViewElementKindSectionFooter,
withReuseIdentifier: “myFooter”, forIndexPath: indexPath) as! FooterView
            footerView.footerImage.image = UIImage(named: “gradientBottom”)
            return footerView
        default:
            assert(false, “Error”)
    }
}
```

The Collection View calls this method to get the view for the header and the footer, so the first step is to identify what kind of view is being requested. The Flow layout uses two constants to identify each view. When the method is called, the constant corresponding to the view is assigned to the `kind` attribute. Reading this attribute with a `switch` statement we are able to know the kind of view we have to return. When the value of `kind` is `UICollectionViewElementKindSectionHeader` we create and return a header view, and when the value is `UICollectionViewElementKindSectionFooter` we create and return a footer view.

Because supplementary views are reusable, the instance is obtained from the `dequeueReusableSupplementaryViewOfKind(kind, withReuseIdentifier: identifier, forIndexPath: path)` method. The method needs the identifier to create the instance from the prototype view in the Storyboard (for this example we called them “myHeader” and “myFooter”, respectively), and also the constant for the `kind` attribute. The rest of the code accesses the elements in the views through the Outlets and assigns the values, as we did for cells before. The result is shown in Figure 15-11.



Figure 15-14: Header and footer in a vertical layout

Do It Yourself: To test supplementary views, start with the example of Figure 15-5 and the view controller of Listing 15-3. Download the images `gradientTop.png` and `gradientBottom.png` from our website and add them to the Assets Catalog. Select the Collection View and activate the options Section Header and Section Footer in the Attributes Inspector panel (Figure 15-7). The header and footer views should be automatically added to the Collection View, as shown in Figure 15-12. Add an Image View to each view and pin them to the sides. Add a label over the Image View in the header and pin it to the bottom and center of the view (Figure 15-13). Create two files with subclasses of the `UICollectionViewReusableView` class called `HeaderView` and `FooterView`. Open the Identity Inspector panel and assign the subclasses to each view (Figures 13-6 and 13-7). From the Attributes Inspector panel assign the identifiers “myHeader” and “myFooter” to each view, respectively. Connect the elements of the views to Outlets, as shown in Listings 15-8 and 15-9. Add the method of Listing 15-10 to the view controller of Listing 15-3. Run the application. You should see something similar to Figure 15-14.

The Basics: The `switch` statement must be exhaustive. That means that every option must be contemplated. If we can’t create a `case` for every possible option, then we have to include a `default` statement. The problem in this case is that the method has to return an object, no matter what. If there is no value to return, then the execution of the application has to be interrupted. This is why in the `default` statement we call the `assert()` function with the value `false`. This method stops the execution of the application and prints a message on the console.

Flow Layout

As we already mentioned, Flow layout is the layout object assigned by default to the

Collection View. The object comes preconfigured to create a grid of cells. The cells are positioned in sequential order from left to right or top to bottom, depending on the scrolling direction. If there is space left in the row or column, the layout object puts the space in between the cells. This is why the cells of our examples are always positioned close to the sides of the screen. We can modify this configuration assigning new values to the layout object. The `UICollectionViewFlowLayout` class offers properties to designate spaces in between the cells, between the lines, or around each section. For example, we can modify the `sectionInset` property to create a margin around the cells.

Listing 15-11: Configuring the Flow layout

```
import UIKit
```

```
class ViewController: UIViewController, UICollectionViewDataSource {
    @IBOutlet weak var collectionItems: UICollectionView!

    override func viewDidLoad() {
        super.viewDidLoad()
        collectionItems.dataSource = self
        collectionItems.backgroundColor = UIColor.whiteColor()
        let layout = collectionItems.collectionViewLayout as! UICollectionViewFlowLayout
        layout.sectionInset = UIEdgeInsets(top: 25, left: 20, bottom: 25, right: 20)
    }

    func collectionView(collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return AppData.items.count
    }

    func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) ->
    UICollectionViewCell {
        let cell = collectionItems.dequeueReusableCellWithReuseIdentifier("myCell", forIndexPath: indexPath) as!
        BookCell
        let file = AppData.items[indexPath.item]
        cell.bookCover.image = UIImage(named: file)
        return cell
    }
}
```

The layout object is stored in the `collectionViewLayout` property of the Collection View. The property returns a generic object of type `UICollectionViewLayout` that we have to cast to the subclass or our layout object (in this case `UICollectionViewFlowLayout`). In the code of Listing 15-11 we get the layout object and modify the value of the `sectionInset` property to generate margins of 25 points at the top and bottom and 20 points on the sides. Figure 15-15 illustrates what the interface looks like now.



Figure 15-15: Margins around the cells in a section

By default, Flow layout organizes the cells considering the size set on the Storyboard. All the cells are displayed with the same size. In our example, the images are not all of the same size, but they are expanded or shrink to fit inside the Image View. If we want to give each cell the size of its image, we can adopt the **UICollectionViewDelegateFlowLayout** protocol and implement the **collectionView(collectionView, layout: collectionViewLayout, sizeForItemAtIndexPath: indexPath)** method to return a specific size for every cell.

Listing 15-12: Declaring a specific size for each cell

```
import UIKit

class ViewController: UIViewController, UICollectionViewDataSource, UICollectionViewDelegateFlowLayout {
    @IBOutlet weak var collectionItems: UICollectionView!

    override func viewDidLoad() {
        super.viewDidLoad()
        collectionItems.dataSource = self
        collectionItems.backgroundColor = UIColor.whiteColor()
        collectionItems.delegate = self

        let layout = collectionItems.collectionViewLayout as! UICollectionViewFlowLayout
        layout.sectionInset = UIEdgeInsets(top: 25, left: 20, bottom: 25, right: 20)
    }

    func collectionView(collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return AppData.items.count
    }

    func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) ->
    UICollectionViewCell {
        let cell = collectionItems.dequeueReusableCellWithReuseIdentifier("myCell", forIndexPath: indexPath) as!
        BookCell

        let file = AppData.items[indexPath.item]
        cell.bookCover.image = UIImage(named: file)
        return cell
    }
}
```

```

func collectionView(collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout,
sizeForItemAtIndexPath indexPath: NSIndexPath) -> CGSize {
    var width: CGFloat = 140
    var height: CGFloat = 180
    let file = AppData.items[indexPath.item]
    if let image = UIImage(named: file) {
        width = image.size.width
        height = image.size.height
    }
    return CGSize(width: width, height: height)
}
}

```

The layout object calls the protocol methods on the delegate assigned to the Collection View's `delegate` property, so we have to designate our view controller as the delegate (`collectionItems.delegate = self`). Flow layout calls the `collectionView(collectionView, layout: collectionViewLayout, sizeForItemAtIndexPath: indexPath)` method every time it needs to know the size of a cell to calculate the layout. In this example we decided to give the cell the size of the image it is going to display. We create a temporary `UIImage` object with the image file for the cell, get the value of its width and height from the `size` property, and return a `CGSize` structure with these values. Every cell in the Collection View will now have the size of its image.

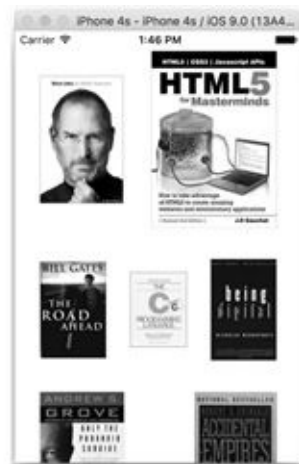


Figure 15-16: Cells of different size

Do It Yourself: Create a new Single View Application project. Add a Collection View and configure the cell as we did for the first example of this chapter. Add the book covers to the Assets Catalog and use the same model of Listing 15-2. Create the subclass for the cell and connect the elements with Outlets, as before. Complete the `ViewController` class with the code of Listing 15-12 and run the application. You should see something similar to Figure 15-16.

IMPORTANT: Calculating the size of a cell every time it is going to be displayed is a resource-consuming process. In a real application, the images are stored in files or databases along with their metadata (size, name, etc.), and when the layout requires this information it retrieves it from the model.

Custom Layout

With Flow layout we can do things like modifying the margins of the sections from the `sectionInset` property (as we did in Listing 15-11), specifying the minimum space between cells from the `minimumInteritemSpacing` property, or even declaring the size of each cell from the protocol methods, but the amount of cells displayed in a row or column is always determined by the layout according to the space available (Figure 15-16). Flow layout gets the dimensions of the Collection View, subtracts the margins and the space between cells, and then positions the cells that fit in the remaining space. If, for example, we set the `minimumInteritemSpacing` property to a value too big to fit two cells in the same row, Flow layout will show only one cell and move the second cell to a new row. But as soon as we rotate the device and the space available becomes wider, the layout positions both cells back in the same row. The only way to make sure we have only the rows we expect or the cells are organized the way we want is by creating our own layout object.

A layout object is built from a subclass of the `UICollectionViewLayout` class. The class includes several methods that we have to overwrite to provide the information required to organize the cells. Some of these methods are optional, but there are a few that are required for the object to have enough information to calculate the layout.

prepareLayout()—This method is executed first by the Collection View to give the layout object the chance to perform some calculations and get the values it needs to construct the layout. When the amount of cells in the Collection View is not significant, it is recommended to calculate their positions and sizes here as well.

collectionViewContentSize()—This method is executed to know the size of the Collection View's content. Collection Views are subclasses of Scroll Views and therefore they have a content area. The system needs to know how large the content area is to let the user scroll through it (see Chapter 11). The method must return a `CGSize` structure with the width and height of the area.

layoutAttributesForItemAtIndexPath(path)—This method is executed to get the attributes of every cell. The values must be returned as a `UICollectionViewLayoutAttributes` object. The **path** attribute is an `NSIndexPath` object with the location of the cell.

layoutAttributesForElementsInRect(rectangle)—This method is executed to get the attributes of all the cells located inside the rectangle determined by the **rectangle** attribute. The method must detect which cells are inside the rectangle and return an array of **UICollectionViewLayoutAttributes** objects with the values of their attributes. The **rectangle** attribute is a **CGRect** structure. The method is frequently used by the Collection View to get the attributes of the visible cells (the cells that are inside the rectangle of the screen).

shouldInvalidateLayoutForBoundsChange(bounds)—This method returns a Boolean value that indicates whether or not the layout must be recalculated when the size of the Collection View changes (usually after a device rotation). The **bounds** attribute is a **CGRect** structure with the values of the new Collection View's inside frame.

The attributes for every cell are managed by objects of the **UICollectionViewLayoutAttributes** class. The class includes the following initializer.

UICollectionViewLayoutAttributes(forCellWithIndexPath: path)—This initializer creates a **UICollectionViewLayoutAttributes** object for a cell in the location indicated by the **path** attribute. The attribute is an **NSIndexPath** object.

The **UICollectionViewLayoutAttributes** class includes the following properties to store and retrieve the values of the cell's attributes.

indexPath—This property sets or returns an **NSIndexPath** object with the position of the cell the attributes belong to.

frame—This property sets or returns the frame of the cell. It is a value of type **CGRect**. Setting the value of this property also sets the values of the **size** and **center** properties.

bounds—This property sets or returns the bounds of the cell. It is a value of type **CGRect**. Setting the value of this property also sets the value of the **size** property.

center—This property sets or returns the coordinates for the center of the cell. It is a value of type **CGPoint**. Setting the value of this property also sets the values for the position in the **frame** property.

size—This property sets or returns the size of the cell. It is a value of type **CGSize**. Setting the value of this property also sets the values for the size in the **frame** and **bounds** properties.

alpha—This property sets or returns the alpha value of the cell (transparency). It is a value of type **CGFloat**.

hidden—This property sets or returns a Boolean value that indicates if the cell should be hidden. The value by default is **false**.

zIndex—This property sets or returns an integer that determines the order of the cells when they overlap. Cells with a higher value will appear on top of cells with a lower value.

transform—This property sets or returns a structure of type **CGAffineTransform** with values that determine different characteristics of the cell. The system works with two coordinate systems, one for the device and another to produce what the user sees. A transformation specifies how points in one coordinate system correlate to the points in the other. Modifying that correlation we can perform operations on the object such as translation, scaling, or rotation, for example (we will study **CGAffineTransform** in Chapter 24).

transform3D—This property sets or returns a structure of type **CATransform3D** with values that determine different characteristics of the cell. It works like the **transform** property but it also processes values for the **z** axis, producing tridimensional effects (we will study **CATransform3D** in Chapter 24).

Subclasses of the **UICollectionViewLayout** class are created as any other subclass we have created before. We have to select the option to create a new file from the File menu, double click the icon Cocoa Touch Class, insert the name for our subclass (for our example we called it **MyCustomLayout**) and select the superclass (**UICollectionViewLayout**, in this case).

Custom layout objects have to be assigned to the **collectionViewLayout** property of the Collection View, but the option is also available in the Attributes Inspector panel. We have to select the Custom value from the Layout option and insert the name of our subclass in the Class field.

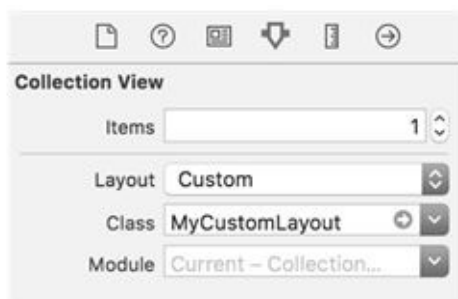


Figure 15-17: Custom layout

The Collection View calls the methods in the layout object to know where to display the cells and the rest of the views. Therefore, the task of the layout object is to calculate the coordinates of every element according to the space available inside the Collection

View and the layout we want to create. For our example we are going to organize the cells always in two columns, as shown in Figure 15-18.



Figure 15-18: Layout with only two columns

For a basic layout object we need to overwrite at least five methods of the `UICollectionViewLayout` class. That includes the `prepareLayout()` method to calculate the values for the layout, the `collectionViewContentSize()` method to return the size of the content area, the `shouldInvalidateLayoutForBoundsChange(bounds)` method to invalidate the layout when the device is rotated, and the methods `layoutAttributesForItemAtIndexPath(path)` and `layoutAttributesForElementsInRect(rectangle)` to return the attributes of the cells. The following is the subclass of the `UICollectionViewLayout` class we have developed to create the layout illustrated by Figure 15-18.

Listing 15-13: *Creating a custom layout object*

```
import UIKit

class MyCustomLayout: UICollectionViewLayout {
    let widthCell: CGFloat = 140
    let heightCell: CGFloat = 180
    let spaceCells: CGFloat = 10
    var listCells: [NSIndexPath: UICollectionViewLayoutAttributes] = [:]

    override func prepareLayout() {
        let collection = collectionView!
        let bounds = collection.bounds
        let columnLeft = bounds.midX - widthCell
        let columnRight = bounds.midX
        let totalCells = collection.numberOfItemsInSection(0)
        var posY: CGFloat = spaceCells
        for var index = 0; index < totalCells; index++ {
```

```

let path = NSIndexPath(forItem: index, inSection: 0)
let attributes = UICollectionViewLayoutAttributes(forCellWithIndexPath: path)
if index % 2 == 0 {
    attributes.frame = CGRect(x: columnLeft, y: posY, width: widthCell, height: heightCell)
} else {
    attributes.frame = CGRect(x: columnRight, y: posY, width: widthCell, height: heightCell)
    posY = posY + heightCell + spaceCells
}
listCells[path] = attributes
}
}

override func collectionViewContentSize() -> CGSize {
    let collection = collectionView!
    let totalCells = collection.numberOfItemsInSection(0)
    let totalColumn = ceil(CGFloat(totalCells) / 2)
    let width = collection.bounds.size.width
    let height = totalColumn * (heightCell + spaceCells)
    return CGSize(width: width, height: height)
}

override func layoutAttributesForItemAtIndex(indexPath: NSIndexPath) -> UICollectionViewLayoutAttributes? {
    return listCells[indexPath]
}

override func layoutAttributesForElementsInRect(rect: CGRect) -> [UICollectionViewLayoutAttributes]? {
    var listAttributes: [UICollectionViewLayoutAttributes] = []
    for attributes in listCells.values {
        let cellRect = attributes.frame
        if CGRectIntersectsRect(cellRect, rect) {
            listAttributes.append(attributes)
        }
    }
    return listAttributes
}

override func shouldInvalidateLayoutForBoundsChange(newBounds: CGRect) -> Bool {
    return true
}
}

```

The first method called by the Collection View is **prepareLayout()**. In this method we have to calculate the values we need to later determine the position of the cells, such as the coordinates of the Collection View, margins, etc. Because the attributes of the cells are used by at least two other methods, it is recommended to also calculate those values in this

method to improve performance. In the subclass of Listing 15-13 we adopt this approach. All the values to position the cells are calculated in the `prepareLayout()` method and stored in a dictionary called `listCells`. Because in our layout the cells have always the same size and space in between, we declare those values in the `widthCell`, `heightCell`, and `spaceCells` properties. Their positions, on the other hand, are calculated according to their place on the list. The first cell is positioned at the top of the left column, the second cell on its right, and then the next cell is moved down to generate a new row (Figure 15-18). We start by getting the `CGRect` value corresponding to the bounds of the Collection View and read its `midX` property to get the x coordinate of the center of the view (see Chapter 4). Using this value we calculate the horizontal position of the left and right columns. The vertical positions of every cell require more work. We first have to get the total amount of cells in the Collection View from the `numberOfItemsInSection()` method (we only consider one section), and then create a `for` loop with this value. Inside the loop, we need to create an `NSIndexPath` object to represent the cell and then use it to create the `UICollectionViewLayoutAttributes` object to hold its data. The vertical position depends on the row the cell is in, and the row depends on the number of cells we have already calculated. To determine where we are at any cycle of the loop, we get the remainder of the index divided by 2. If the result is equal to 0, it means the index is even and the cell belongs to the left column, otherwise the index is odd and therefore the cell belongs to the right column. Considering this result, we set the position of the cell to be in the left or right column, assign the values to the `frame` property, and finally store the `UICollectionViewLayoutAttributes` object in the `listCells` dictionary. Notice that when the cell is in the right column, we also calculate the vertical position for the next row (`posY = posY + heightCell + spaceCells`). After the loop is over, the dictionary contains a `UICollectionViewLayoutAttributes` object with the data for every cell.

The next method in our subclass is called `collectionViewContentSize()`. Its purpose is to return the size of the content area so the system knows how long it has to scroll to let the user see all the cells. In our layout, the width of the content area is always the same as the Collection View (there is no horizontal scroll), but the height is calculated from the total amount of cells in a column. To get this value we first get the total amount of cells from the `numberOfItemsInSection()` method, then divide it by two to get the cells in a single column, and finally calculate the height of the content area by multiplying the number of cells times their height and the space between them.

The Collection View calls two methods to get the attributes for the cells. The `layoutAttributesForItemAtIndexPath(path)` method is called when the Collection View needs the attributes of a single cell, and the `layoutAttributesForElementsInRect(rectangle)` method is called when the Collection View needs the attributes of all the cells inside a specific rectangle (usually called to get the attributes of the visible cells in the rectangle represented by the screen). The first method is simple, we just have to get the element of the `listCells` dictionary that corresponds to the cell's path (`listCells[indexPath]`), but the second method has to return an array of `UICollectionViewLayoutAttributes` objects corresponding to the cells inside the rectangle, so we have to read the values of every cell, compare their frames with the rectangle sent to the method, and add the attributes of those cells that intersect to the array.

At first sight the task looks daunting, but the `CGRectIntersectsRect()` function provided by the Core Graphics framework makes it easy. This function compares the `CGRect` value of its first attribute with the `CGRect` value of its second attribute and returns `true` if they intersect (see Chapter 4). All we have to do to know what attributes to return is to get the value of the `frame` property of each cell and compare it with the value of the `rect` attribute using this function. If they intersect, the `UICollectionViewLayoutAttributes` object is added to the array.

The last method in our subclass is `shouldInvalidateLayoutForBoundsChange(bounds)`. The purpose of this method is to indicate if we want the layout to be recalculated when the size of the Collection View changes. We return `true` to adapt the layout to every orientation.

Do It Yourself: Create a new Single View Application project with the interface of Figure 15-5, the `UICollectionViewCell` subclass of Listing 15-1, the model of Listing 15-2, and the view controller of Listing 15-3. The Flow layout is assigned to the Collection View by default, so you need to create a new file with a subclass of the `UICollectionViewLayout` class called `MyCustomLayout`, select the option Custom from the Attributes Inspector panel, and assign the `MyCustomLayout` subclass to the Collection View (Figure 15-17). Complete the `MyCustomLayout` subclass with the code of Listing 15-13 and run the application. You should see something similar to Figure 15-18.

IMPORTANT: Besides cells, the `UICollectionViewLayoutAttributes` class can also manage supplementary views (like the header and footer created by the Flow layout) and additional views called *Decoration Views*. The example of this chapter only works with cells, but the same logic applies to the rest of the elements. The topic is beyond the scope of this book. For more information visit our website and follow the links for this chapter.

Real-Life Application

Custom layouts are usually not necessary in real life. Flow layout provides enough flexibility to adapt the layout to the most common scenarios. For example, if we want to make sure that only one row is displayed at a time, independently of the space available on the screen, we can create a Collection View of the size of the cell. In the following example we are going to implement a Collection View with horizontal scroll and a specific height to always show only one column of items.



Figure 15-19: *Collection View with a specific size*

In this example, we give a sense of depth to the interface using an Image View under a Visual Effect View. First, we have to create a Single View Application project, add an Image View to the main view with the image we want to show in the background, and pin it to the sides with Space constraints (Figure 15-19, left). Next, we have to add a Visual Effect View over the Image View and also pin it to the sides of the main view with Space constraints (Figure 15-19, center). The Collection View has to be added to the Visual Effect View with Space constraints to the left and right sides, a Height constraint of 300 points, and a Vertical Alignment constraint to set its position at the center of the screen (Figure 15-19, right). Finally, we have to elect the Collection View and change the Scroll Direction option to Horizontal from the Attributes Inspector panel (Figure 15-7), and the size of the cell to 215 x 300 (Figure 15-4) from the Size Inspector panel.

For the content of the cell we need an Image View that will show the cover of the book and two labels for the title and the author. To delimit the area we can put the labels inside an empty view below the image, as shown in Figure 15-20.

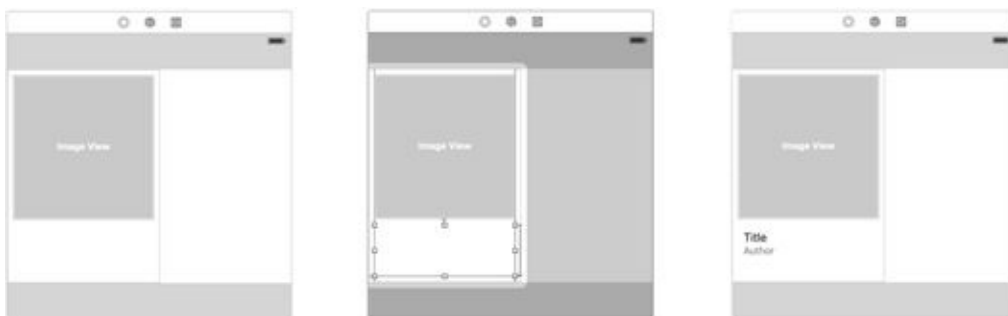


Figure 15-20: *Configuring the prototype cell*

The design always depends on what we want to achieve with our application. For this example we decided to include the labels inside an empty view. We have to add a view with a white background, pin it to the Image View on the top and to the cell's bottom, left and right margins, and give it a Height constraint to fix its height (Figure 15-20, center). The two labels are added next to the view. We have to pin the label at the top to the view's top, left, and right margins, and the label at the bottom with the label at the top and the view's left and right margins. The label at the top must be configured from the Attributes Inspector panel to have 2 lines (Figure 5-19). This is so the label at the top can display the

book's title in two lines if necessary.

To access the elements of the cell, we have to create a subclass of the `UICollectionViewCell` class called `BookCell`, assign it to the cell from the Identity Inspector panel (Figure 13-6), and connect the elements to the subclass with Outlets, as shown next.

Listing 15-14: Modifying the cell's background

import UIKit

```
class BookCell: UICollectionViewCell {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    @IBOutlet weak var titleView: UIView!
    override func awakeFromNib() {
        super.awakeFromNib()
        backgroundColor = UIColor(red: 0.7, green: 0.7, blue: 0.7, alpha: 0.3)
        titleView.backgroundColor = UIColor(red: 1.0, green: 1.0, blue: 1.0, alpha: 0.3)
    }
}
```

For this example, we want the cell and the white view containing the labels to be semitransparent. The transparency is usually achieved by modifying the alpha value of the element, but the problem in this case is that this also affects the elements inside. The best way to turn the views semitransparent without affecting the Image View and the labels is to assign `UIColor` objects with an alpha value different than 1.0 to the `backgroundColor` properties of each element. This is why in the subclass of Listing 15-14 we created an additional Outlet for the white view and modified the properties from the `awakeFromNib()` method (see also Listing 15-4).

Now that the cell is ready we have to create the view controller to manage the Collection View. The code is similar than previous examples, but we modified the Flow layout to improve the design, and also gave the Collection View a clear color so the background image can be seen through.

Listing 15-15: Modifying the design of the Collection View from the view controller

import UIKit

```
class ViewController: UIViewController, UICollectionViewDataSource {
    @IBOutlet weak var collectionItems: UICollectionView!
    override func viewDidLoad() {
        super.viewDidLoad()
```

```

collectionItems.dataSource = self
collectionItems.backgroundColor = UIColor.clearColor()
let layout = collectionItems.collectionViewLayout as! UICollectionViewFlowLayout
layout.sectionInset = UIEdgeInsets(top: 0, left: 40, bottom: 0, right: 40)
layout.minimumLineSpacing = 40
}

func collectionView(collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
    return AppData.items.count
}

func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) ->
UICollectionViewCell {
    let cell = collectionItems.dequeueReusableCellWithReuseIdentifier("myCell", forIndexPath: indexPath) as!
    BookCell

    let file = AppData.items[indexPath.item]
    cell.bookCover.image = UIImage(named: file)
    if let data = AppData.itemsData[file] {
        cell.bookTitle.text = data[0]
        cell.bookAuthor.text = data[1]
    }
    return cell
}
}

```

Now all that is left to do is to create the Swift file with the model of Listing 15-2 to provide the data. Remember to connect the Collection View to the **collectionItems** Outlet, assign the “myCell” identifier to the cell, and add to the Assets Catalog all the images with the covers and also the image for the Image View in the background (for our example we used an image called spot1.png available on our website).



Figure 15-21: One column Collection View

Chapter 16

Split View Controllers

16.1 Universal Container

The container view controllers studied so far are good for devices with small screens, such as iPhones and iPods, but the space available in devices like the iPads and iPhones 6 Plus demand a more elaborated design. UIKit includes a subclass of `UIViewController` called `UISplitViewController` that can present two main views at a time, one called the Master view and another called the Detail view. The Master view is used to present the main interface, with controls that let the user select options and content, while the purpose of the Detail view is to show the selected content.



Figure 16-1: Two main views presented by a Split View Controller

The views are presented on the screen depending on the space available. On iPads and iPhones 6 Plus in landscape mode, the Master and Detail views are shown together (Figure 16-1), but on the rest of the devices, and also on iPads and iPhones 6 Plus in portrait mode, only one view is shown at a time. When only one view is presented, the views replace one another. For example, if the Master view is embedded in a Navigation Controller (as it happens in the application of Figure 16-1), the Detail view is presented with a transition from right to left and a Back button is added to its Navigation Bar to let the user go back to the Master view, as shown in Figure 16-2.

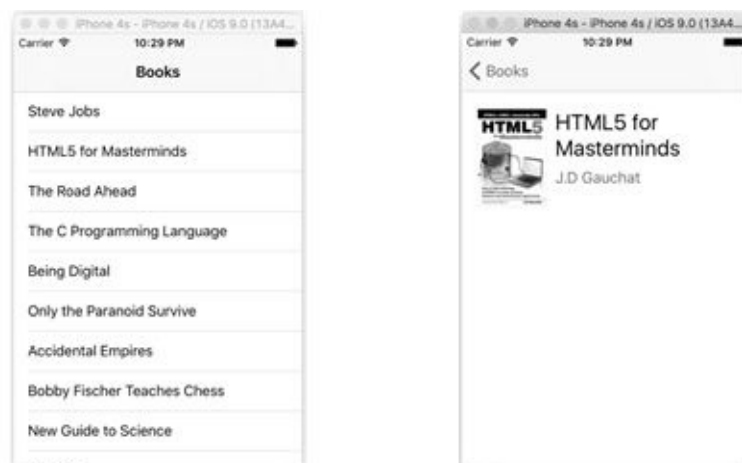


Figure 16-2: Split View Controller in portrait mode

Because of their capacity to adapt the interface to the space available, Split View Controllers are perfect to create universal applications that work on iPads, iPhones, and iPods. That is why Xcode offers a template called *Master-Detail application* that initializes the Storyboard with a Split View Controller and its Master and Detail views. Although this template includes everything we need to implement a Split View Controller, it also comes with preprogrammed code to produce a sample application that we usually have to erase to create our own. For this reason, it is better to start with a Single View Application project and then add the Split View Controller from the Object Library.



Figure 16-3: *Split View Controller option in the Object Library*

The option adds to the Storyboard a Split View Controller with Master and Detail views, as shown in Figure 16-4.

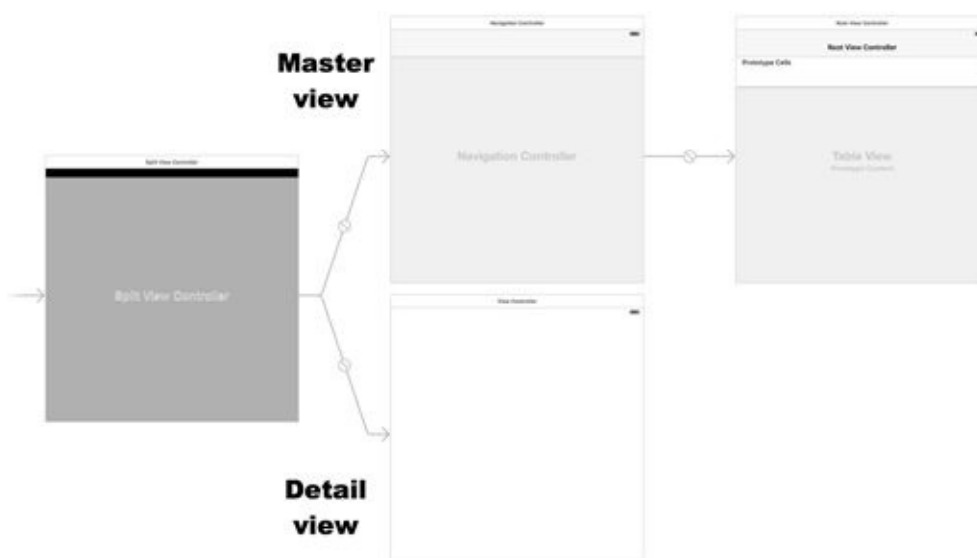
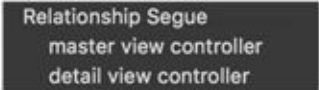


Figure 16-4: *Split View Controller in the Storyboard*

The Split View Controller is the view on the left, and the Master and Detail views are on the right. In this standard configuration, the Master view comes embedded in a Navigation Controller to provide tools for navigation. The Master and Detail views are connected to the Split View Controller with special Segues called *Master View Controller* and *Detail View Controller*.



Relationship Segue
master view controller
detail view controller

Figure 16-5: *Segues for Split View Controllers*

The Split View Controller offered by the Object Library already comes connected with a Master view and a Detail view, but we can delete these views and incorporate our own. After adding the views, all we have to do is to control-drag a line between the Split View Controller and the views and select the type of Segue according to their roll in the interface (Figure 16-5).

UISplitViewController

Split View Controllers automatically expand and collapse to adapt to the space available, but we can perform some adjustments and improve the interface from the properties and methods included in the `UISplitViewController` class. The following are the most frequently used.

viewControllers—This property sets or returns an array with references to the view controllers currently shown on the screen. In an expanded interface, the array contains two view controllers, the Master at index 0 and the Detail at index 1 (also called *primary* and *secondary*), but in a collapsed interface, the array contains only one view controller (usually the Detail).

collapsed—This property returns a Boolean value that determines the state of the interface. The value `true` means the interface is collapsed (the Split View Controller is presenting only one view and it can't be expanded). The value `false` means the interface is expanded (the Split View Controller is presenting or can present two views).

presentsWithGesture—This property sets or returns a Boolean value that determines whether or not a hidden view can be shown with a swipe gesture. Some configurations, like the iPad in portrait mode, present only one view and allow the user to make the other visible with a swiping gesture. This property activates or deactivates the feature (`true` by default).

preferredPrimaryColumnWidthFraction—This property sets or returns a `CGFloat` value that determines the width of the primary view in relation to the secondary view. The value must be specified between 0.0 and 1.0. For example, a value of 0.5 gives the primary view a size of 50% the width of the space available.

primaryColumnWidth—This property returns a `CGFloat` value with the width of the primary view expressed in points.

minimumPrimaryColumnWidth—This property sets or returns a `CGFloat` value that determines the minimum width of the primary view in points (usually the Master view).

maximumPrimaryColumnWidth—This property sets or returns a `CGFloat` value that determines the maximum width of the primary view in points (usually the Master view).

preferredDisplayMode—This property sets or returns the preferred display mode. It is an enumeration of type `UISplitViewControllerDisplayMode` with the values `Automatic`, `PrimaryHidden`, `AllVisible`, and `PrimaryOverlay`.

displayMode—This property returns the current display mode. It is an enumeration of type `UISplitViewControllerDisplayMode` with the values `Automatic`, `PrimaryHidden`, `AllVisible`, and `PrimaryOverlay`.

displayModeButtonItem()—This method returns a `UIBarButtonItem` object designed to create a button that lets the user change the Split View Controller's display mode.

UISplitViewControllerDelegate

Split View Controllers can also designate a delegate to report changes in the views. The delegate must adopt the `UISplitViewControllerDelegate` protocol. The following are some of the methods defined in this protocol.

splitViewController(svc, willChangeToDisplayMode: displayMode)—This method is called by the Split View Controller to tell the delegate that the display mode is about to change. The **displayMode** attribute is an enumeration of type `UISplitViewControllerDisplayMode` with the values `Automatic`, `PrimaryHidden`, `AllVisible`, and `PrimaryOverlay`.

splitViewController(splitViewController, collapseSecondaryViewController: secondaryViewController, ontoPrimaryViewController: primaryViewController)—This method is called by the Split View Controller on its delegate to know whether or not to incorporate the secondary view into the interface. The method must return `false` if we want the Split View Controller to try to incorporate the secondary view or `true` otherwise.

targetDisplayModeForActionInSplitViewController(svc)—This method is called by the Split View Controller on its delegate to know the display mode to apply after the user presses the bar button created by the `displayModeButtonItem()` method or

swipe his finger over the secondary view, for example. The value returned is an enumeration of type `UISplitViewControllerDisplayMode` with the values `Automatic`, `PrimaryHidden`, `AllVisible`, and `PrimaryOverlay`.

Implementing Split View Controllers

An application based on a Split View Controller works like any other we have built so far. The elements for the interface are added to the views and view controllers are assigned to the views to manage their content. For example, based on the views provided by a standard Split View Controller, we can create an application to manage a list of books. When a book is selected in the Master view, the Detail view shows its cover and title. Figure 16-6 illustrates the modifications we have to make on the Master and Detail views to get a simple application like this.

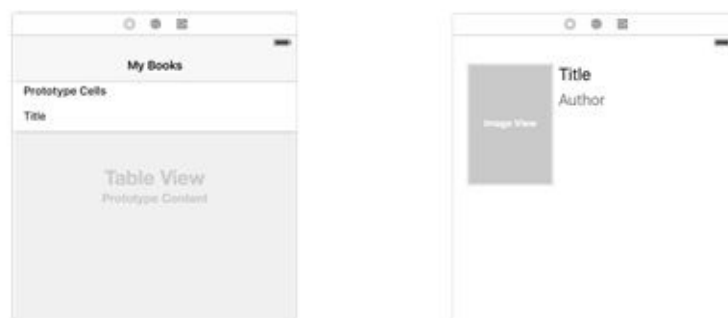


Figure 16-6: Master and Detail views

Notice that only the Master view and the Detail view have been modified, the rest of the views in the Storyboard are containers and therefore don't have content of their own. In the Master view we changed the title of the Navigation Bar and turned the custom cell provided by default to a Basic cell. The Detail view, on the other hand, was improved with new elements, including an Image View to present the cover of the selected book, and two labels for the title and the author.

Do It Yourself: Create a new Single View Application project. Erase the main view and add a Split View Controller from the Object Library (Figure 16-3). Select the Split View Controller and activate the option *Is Initial View Controller* from the Attributes Inspector panel. Select the cell in the Master view, change its style to Basic, and give it the identifier "myCell" (Figure 14-7). Double click on the Navigation Bar to change its title. Add an Image View and two labels to the Detail view, as shown in Figure 16-6. Assign Space constraints to the elements to pin them to the sides of the view and with each other, and give the Image View a fixed size with Width and Height constraints (we used a size of 120 x 170).

The Master and Detail views work like any view studied before. The only difference is that they have a specific area assigned on the screen. If we connect another view to the Master view with a Show Segue, for example, the second view will replace the current view and become the Master view. If we use the normal type of Segues studied in Chapter 13, the views replace one another in the same area, but Split View Controllers include a special type of Segue to connect the Master view with the Detail view called *Show Detail*. Every time a Segue of this type is triggered from the Master view, the view connected to this Segue is opened in the area of the Detail view and therefore it becomes the new Detail view. Figure 16-7 illustrates how we implement a Show Detail Segue in our example.

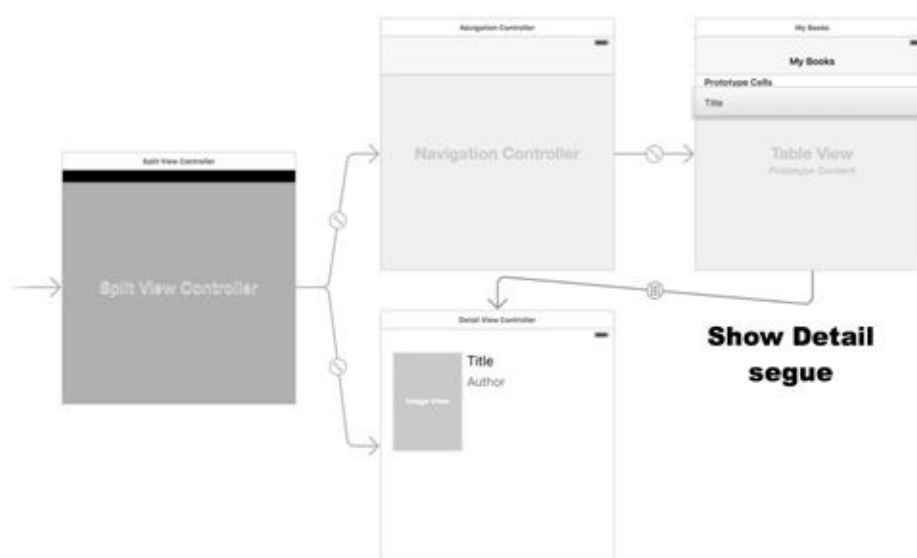


Figure 16-7: Show Detail Segue

The Show Detail Segue was connected from the cell in the Master view to the Detail view. Now every time the user selects a cell in the Master view, the Segue is triggered and a new Detail view is opened. Despite this new type of connection, the codes for the view controllers don't change too much. The following is the code for the Master view of our example.

Listing 16-1: Controlling the Master view

```
import UIKit

class MasterViewController: UITableViewController {
    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return AppData.items.count
    }

    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
    UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)
```

```

let file = AppData.items[indexPath.row]
if let data = AppData.itemsData[file] {
    cell.textLabel?.text = data[0]
}
return cell
}

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showDetail" {
        let controller = segue.destinationViewController as! DetailViewController
        let path = tableView.indexPathForSelectedRow
        controller.selected = path?.row
    }
}
}

```

The view created by Xcode for the Master view is a Table View Controller and therefore our view controller has to be a subclass of the `UITableViewController` class. In Listing 16-1 we called this subclass `MasterViewController` and implement inside the delegate methods `tableView(tableView, numberOfRowsInSection: section)` and `tableView(tableView, cellForRowAtIndexPath: indexPath)` to provide the data to the table, and the `prepareForSegue()` method to send the information of the selected cell to the Detail view when the Show Detail Segue is triggered. Notice that for this example we identified the Segue with the string “showDetail” and used the same model and images implemented in the examples of Chapter 15 (see Listing 15-2).

The view controller for the Detail view, on the other hand, is a subclass of the `UIViewController` class and it just has to provide the Outlets to access the elements in the main view and modify their values according to the information received through the `selected` property.

Listing 16-2: Controlling the Detail view

```

import UIKit

class DetailViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    var selected: Int!

    override func viewDidLoad() {
        super.viewDidLoad()

        if selected != nil {
            let file = AppData.items[selected]

```

```

bookCover.image = UIImage(named: file)
if let data = AppData.itemsData[file] {
    bookTitle.text = data[0]
    bookAuthor.text = data[1]
}
}
}
}
}

```

The codes for this application don't introduce anything new, but because the views are part of a Split View Controller and connected through a Show Detail Segue they now can be presented on the screen at the same time and work together to show the information requested by the user. The Split View Controller decides whether or not there is enough space on the screen to show both views or if it has to show one and include navigation tools to let the user access the other.

Do It Yourself: Control-drag a line from the cell in the Master view to the Detail view and select the option Show Detail to create the Segue. Select the Segue and insert the identifier “showDetail” from the Attributes Inspector panel. Create a file with a subclass of `UITableViewController` called `MasterViewController` and assign it to the Master view. Create a file with a subclass of `UIViewController` called `DetailViewController` and assign it to the Detail view. Connect the elements in the Detail view with the Outlets in the `DetailViewController` class. Create a Swift file with the model of Listing 15-2 (you will also need to add to the Assets Catalog the images with the books' covers used for the examples of Chapter 15). Run the application in the simulators for iPads and iPhones. Rotate the simulator to see how the Split View Controller responds in every device.

In other iPhones than the 6 Plus, the views are presented one at a time, similar to when the iPhone 6 Plus is in portrait mode, but iPads exhibit a slightly different behaviour. They also present both views in landscape mode and one view in portrait mode, but instead of offering a Back button, they let the user swipe the finger to open the Master view as an overlay. The view shows up from the left of the screen and partially covers the Detail view, as shown in Figure 16-8.

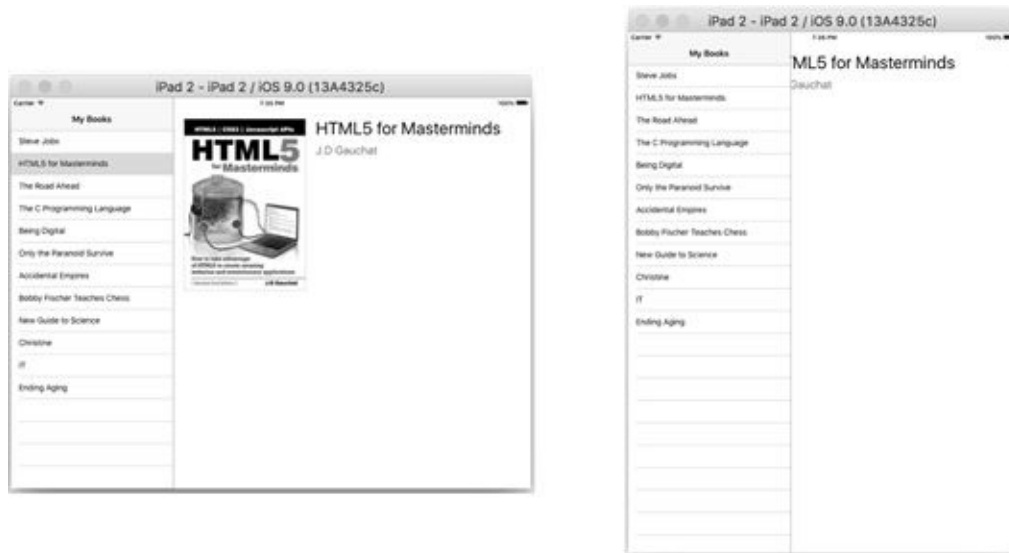


Figure 16-8: PrimaryOverlay mode in iPad

The way views are presented is determined by modes defined by the Split View Controller. The mode used when the iPad is in portrait mode is called **PrimaryOverlay** (Figure 16-8, right). There are a total of four modes: **Automatic** (the Split View Controller decides the appropriate mode), **PrimaryHidden** (the primary view is hidden), **AllVisible** (both views are visible), and **PrimaryOverlay** (the primary view is displayed over the secondary view). If we don't want to let the Split View Controller decide how the views should be presented, we can modify the value of the **preferredDisplayMode** property. The Split View Controller considers this value and changes the mode when possible. For example, if instead of an overlay we want to show both views at the same time when the iPad is in portrait mode, we can assign the value **AllVisible** to this property.

Listing 16-3: Changing the Split View Controller mode

```
override func viewDidLoad() {
    super.viewDidLoad()
    splitViewController?.preferredDisplayMode = .AllVisible
}
```

Modifications to the configuration of the Split View Controller are usually performed in the Master view's view controller. View controllers offer the **splitViewController** property to access the Split View Controller they belong to. In the **viewDidLoad()** method of Listing 16-3 we use this property to get a reference to the Split View Controller and assign the value **AllVisible** to its **preferredDisplayMode** property. Now, instead of the overlay used to access the Master view in an iPad in portrait mode, we will see the Master view along with the Detail view in both orientations.



Figure 16-9: AllVisible mode in iPad

Do It Yourself: Add the `viewDidLoad()` method of Listing 16-3 to the `MasterViewController` class. Run the application in the iPad simulator and rotate the screen. You should see something similar to Figure 16-9.

If we want to change the mode only when the current mode is of a certain type, we can check the value of the `displayMode` property first. For example, in an iPad in portrait mode, the Master view is presented when the user swipes a finger over the Detail view. To hide the Master view, the user has to tap outside the view. If we want to close the Master view when a cell is selected, we have to implement the `tableView(tableView, didSelectRowAtIndexPath: indexPath)` method to change the mode to `PrimaryHidden`. This mode hides the Master view, but the problem is that we only want to do it when the Master view was presented as an overlay. The solution is to check the current mode and only change it if it is of type `PrimaryOverlay`.

Listing 16-4: Changing the mode from `PrimaryOverlay` to `PrimaryHidden`

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let mode = splitViewController?.displayMode
    if mode == .PrimaryOverlay {
        splitViewController?.preferredDisplayMode = .PrimaryHidden
    }
}
```

With the implementation of the method of Listing 16-4 in our `MasterViewController` class we make the code hide the Master view when a cell is selected, but only when the view was presented as an overlay. The Master view is not affected in other devices and configurations.

Do It Yourself: Remove the `viewDidLoad()` method added in the previous example. Add the method of Listing 16-4 to the `MasterViewController` class. Run the application in an iPad simulator. In portrait mode, swipe your finger to show the Master view (Figure 16-8, right), and tap on a cell to select it. The Detail view should show the selected book and the Master view should be hidden.

IMPORTANT: The mode set by the `preferredDisplayMode` property is only considered by the Split View Controller when the interface is expanded (both views are shown or have the possibility to be shown simultaneously on the screen). The value assigned to this property is a suggestion to the Split View Controller rather than an order. The Split View Controller takes the suggestion and accommodates the interface when possible.

16.2 Improving the Interface

Split View Controllers manage the views for us, but there are still a lot of things we have to take care of ourselves. An important aspect of working with Split View Controllers is that in most cases the Detail view is shown on the screen before the user requests any information. This means that we have to prepare the Detail view to show something when no content is available. In previous examples, we just displayed the labels with the text defined in the Storyboard and an empty Image View, but this is not enough for a professional application. Fortunately, there are several alternatives we can adopt to improve our app's design.

Item by Default

The simplest option to provide relevant information for the user in the Detail view presented by default is to search for any item in the model and show it as a placeholder. For instance, the `viewDidLoad()` method of the `DetailViewController` class of our example could get and show the information of the first book in the `items` array if no book was selected yet.

Listing 16-5: Selecting a book by default

```
import UIKit

class DetailViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    var selected: Int!

    override func viewDidLoad() {
        super.viewDidLoad()

        if selected == nil && !AppData.items.isEmpty {
            selected = 0
        }

        if selected != nil {
            let file = AppData.items[selected]
            bookCover.image = UIImage(named: file)

            if let data = AppData.itemsData[file] {
                bookTitle.text = data[0]
                bookAuthor.text = data[1]
            }
        }
    }
}
```

```
}  
}  
}
```

The code of Listing 16-5 compares the `selected` property to the value `nil` to know if a book was already selected and, if the `items` array is not empty, assigns the index 0 to the property. Now the first book on the list will be shown in the Detail view until the user selects a different one.

Do It Yourself: Replace the code in your `DetailViewController` class by the code of Listing 16-5 and run the application. The first screen you should see is the Detail view with the data of the book at index 0.

Detail View by Default

The previous solution doesn't solve all of our problems. If the `items` array is empty, we still have an empty Detail view. A better solution is to assign a view by default with a message that explains to the user what is going on, and replace that view with another one when there is real content to show. The easiest way to do it is to designate the view containing the message as the Detail view of our Split View Controller. This way, the view with the message will always be shown unless the user performs an action that replaces it with the Detail view we really want to show.

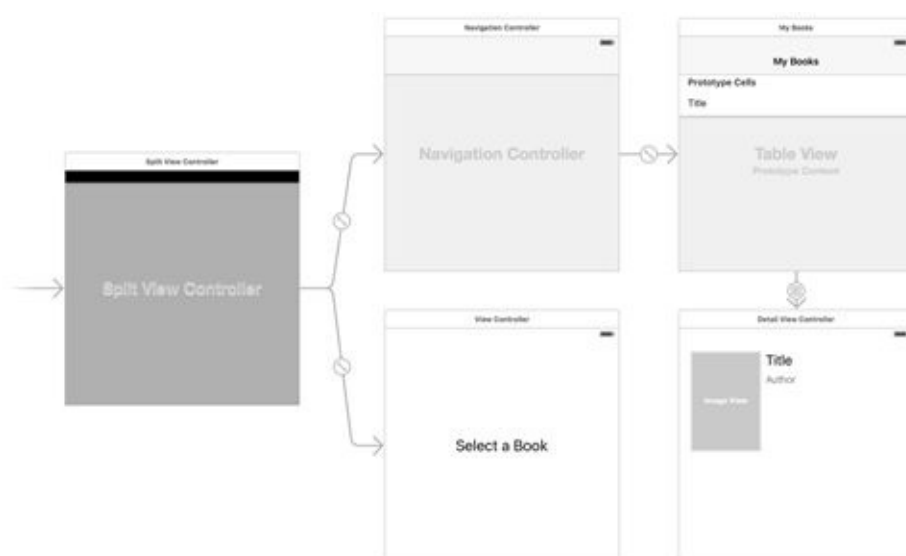


Figure 16-10: Split View Controller with a Detail view by default

The interface of Figure 16-10 designates a new Detail view for our Split View

Controller, but the second Detail view is still connected to the Master view with a Show Detail Segue and it will be shown in the Detail view's area every time the user selects a cell. As a result, the view with the message "Select a Book" is shown as soon as the app is launched, but it is replaced when a book is selected.



Figure 16-11: *Default Detail view*

Do It Yourself: Delete the Detail View Controller Segue. Move the Detail view to the right and add in its place a new Scene from the Object Library (View Controller option). Add a label in the middle of the view with the message "Select a Book". Create a new Detail View Controller Segue from the Split View Controller to this view. Run the application. The initial screen should look like Figure 16-11.

Display Mode Button

At this point, the interface provides everything the user needs to navigate and access the information, but there are still issues we have to solve. First, when only the Detail view is visible, there are not enough clues for the user to know that there is another view available. And second, when the interface is expanded, there is no way for the user to show the Master view or remove it from the screen. The `UISplitViewController` class offers the `displayModeButtonItem()` method to solve both problems. The method returns a bar button item called *Display Mode Button* that gives a hint to the users that there is more to see, and lets them show or remove the views at will. The button returned by this method is a `UIBarButtonItem` object, so we have to embed the Detail view in a Navigation Controller to be able to incorporate it to the interface. Because we have two Detail views, one to present the initial message and another to present the book's data, we have to repeat the process twice, embedding both views in their respective Navigation Controllers.

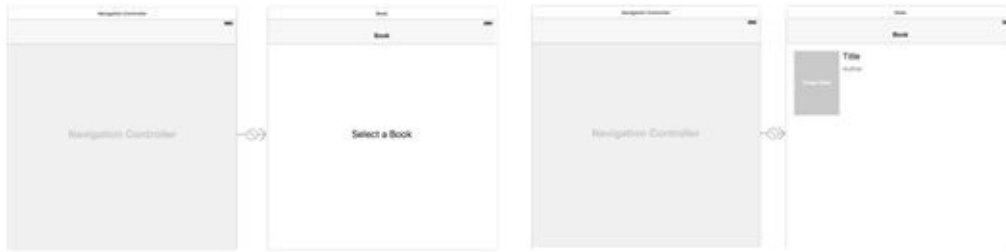


Figure 16-12: *Detail views embedded in Navigation Controllers*

Now that every Detail view has its Navigation Bar, we can add the bar button item to each one of them. The following example expands the `DetailViewController` class to add a Display Mode Button to the second Detail view and configure the bar.

Listing 16-6: *Adding a Display Mode Button to the Navigation Bar of the second Detail view*

```
import UIKit

class DetailViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    var selected: Int!
    override func viewDidLoad() {
        super.viewDidLoad()

        let button = splitViewController?.displayModeButtonItem()
        navigationItem.leftBarButtonItem = button
        navigationItem.leftItemsSupplementBackButton = true
        if selected != nil {
            let file = AppData.items[selected]
            bookCover.image = UIImage(named: file)
            if let data = AppData.itemsData[file] {
                bookTitle.text = data[0]
                bookAuthor.text = data[1]
            }
        }
    }
}
```

The code of Listing 16-6 stores the button returned by the `displayModeButtonItem()` method in a constant and then assigns that constant to the `leftBarButtonItem` property. This adds the button at the left side of the bar, which is a convenient place for a button that removes or

shows the Master view, but because Navigation Bars hide the Back button when other buttons are present on the same side, we also have to modify the `leftItemsSupplementBackButton` property to tell the Navigation Bar to present both buttons at the same time (see Navigation Controllers in Chapter 13).

The previous code adds the button to the second Detail view, but we still have to add it to the initial Detail view (the one that is displayed when the app is launched). We hadn't created a view controller for this view before, so we have to do it now. For this example we called it `EmptyViewController`. The code to add the button is exactly the same implemented in the `DetailViewController` class.

Listing 16-7: Adding a Display Mode Button to the Navigation Bar of the initial Detail view

```
import UIKit

class EmptyViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let button = splitViewController?.displayModeButtonItem()
        navigationItem.leftBarButtonItem = button
        navigationItem.leftItemsSupplementBackButton = true
    }
}
```

There is one more thing we have to do for our example to work properly. Because the second Detail view is now embedded in a Navigation Controller, it is no longer the Segue's destination view controller, so we have to modify the `prepareForSegue()` method in the `MasterViewController` class to incorporate the new path. First, we have to get a reference to the Navigation Controller and then access the `DetailViewController` through its `topViewController` property (the `DetailViewController` object is now the top view controller of the Navigation Controller). If we don't perform this change, the `prepareForSegue()` method will try to modify the value of a property that doesn't exist in the Navigation Controller and the app will crash.

Listing 16-8: Referencing the Detail view through the Navigation Controller

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?){
    if segue.identifier == "showDetail" {
        let navigation = segue.destinationViewController as! UINavigationController
        let controller = navigation.topViewController as! DetailViewController
        let path = tableView.indexPathForSelectedRow
        controller.selected = path?.row
    }
}
```

```
}  
}
```

If both views are present on the screen, the button shows a double arrow (Figure 16-13, left). When the user taps on this arrow, the Master view is hidden and the button turns into a single arrow pointing to the left (Figure 16-13, right).



Figure 16-13: Display Mode Button in action

Do It Yourself: Embed the initial Detail view (the one with the message “Select a Book”) in a Navigation Controller (Editor/Embed In/Navigation Controller). Do the same for the second Detail view (the view that presents the book’s data). Copy the code of Listing 16-6 in the `DetailViewController` class. Create a new file with a subclass of the `UIViewController` class called `EmptyViewController` and assign it to the initial Detail view. Complete the subclass with the code of Listing 16-7. Replace the `prepareForSegue()` method of the `MasterViewController` class by the method of Listing 16-8. Run the application in the iPhone 6 Plus simulator. Turn the simulator to landscape mode. Press the Display Mode Button. You should see something similar to Figure 16-13.

Split View Controller Delegate

When the app is first launched, the Split View Controller decides how to display the views. If there is only space available for one view, by default it will show the Detail view. This means that, for example, when we launch the app for the first time in an iPhone in portrait mode, we will always see the Detail view until we press the Back button. The `UISplitViewControllerDelegate` protocol defines the `splitViewController(splitViewController, collapseSecondaryViewController:secondaryViewController, ontoPrimaryViewController:primaryViewController)` method that lets us change this predefined behaviour. The method is called by the Split View Controller to know what we want to do with the secondary view (the Detail view) when there is a possibility to include it in the interface. Returning the value `false` from this method tells the Split View Controller to try to incorporate the secondary view, but if the value returned is `true`, the Split View Controller does nothing

and the view is not shown on the screen.

Usually the view controller for the Master view is designated as the delegate of the Split View Controller. In the following example we improve the **MasterViewController** class to include the protocol method and force the Split View Controller to show the Master view when the app is launched.

Listing 16-9: Showing the Master view instead of the Detail view when the app is launched

```
import UIKit
```

```
class MasterViewController: UITableViewController, UISplitViewControllerDelegate {
    override func viewDidLoad() {
        super.viewDidLoad()
        splitViewController?.delegate = self
    }
    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return AppData.items.count
    }
    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
    UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)
        let file = AppData.items[indexPath.row]
        if let data = AppData.itemsData[file] {
            cell.textLabel?.text = data[0]
        }
        return cell
    }
    override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        if segue.identifier == "showDetail" {
            let navigation = segue.destinationViewController as! UINavigationController
            let controller = navigation.topViewController as! DetailViewController
            let path = tableView.indexPathForSelectedRow
            controller.selected = path?.row
        }
    }
    func splitViewController(splitViewController: UISplitViewController, collapseSecondaryViewController
    secondaryViewController: UIViewController, ontoPrimaryViewController primaryViewController:
    UIViewController) -> Bool {
        return true
    }
}
```


Do It Yourself: Replace the code in the `MasterViewController` class by the code of Listing 16-9. Run the application in an iPhone in portrait mode. You should see the list of books instead of the “Select a Book” message you have seen so far.

16.3 Expanding the Interface

As we already mentioned, the interface can be expanded to include all the views we need. When the Master and Detail views are embedded in Navigation Controllers, the rest of the views can be connected with Show Segues, as we did before for other interfaces that include Navigation Controllers. For example, we can add a button below the book's cover in the detail view of our interface and connect it to a separate view to enlarge the cover.

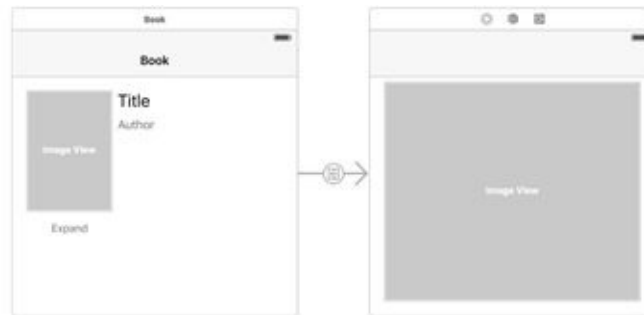


Figure 16-14: Expanding the interface

In Figure 16-14 we have incorporated the button below the book's cover with the title “Expand” and connected it with a Show Segue to a view that includes an Image View inside. The Segue was identified with the string “showPicture”. When this Segue is triggered, we have to pass the value of the `selected` property in the Detail view to the new view. For this purpose, the `DetailViewController` class has to implement the `prepareForSegue()` method.

Listing 16-10: Sending the index of the selected book to `PictureViewController`

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?){  
    if segue.identifier == “showPicture” {  
        let controller = segue.destinationViewController as! PictureViewController  
        controller.selected = selected  
    }  
}
```

The view controller for the new view has to include a property called `selected` to receive the index of the selected book sent by the Detail view and then update the Image View using this value. We called this view controller `PictureViewController`.

Listing 16-11: Showing a maximized version of the book's cover

```
import UIKit
```

```

class PictureViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    var selected: Int!
    override func viewDidLoad() {
        super.viewDidLoad()
        if selected != nil {
            let file = AppData.items[selected]
            bookCover.image = UIImage(named: file)
        }
    }
}

```

The code of Listing 16-11 declares the `selected` property to receive the index of the book and an Outlet to access the Image View. When the view is loaded, the `viewDidLoad()` method gets the information from the model, creates the image, and shows it on the screen. Figure 16-15 shows the application running in an iPhone 6 Plus. When the Expand button is pressed, the Detail view with the information of the book is replaced by another view that shows a bigger version of the cover.

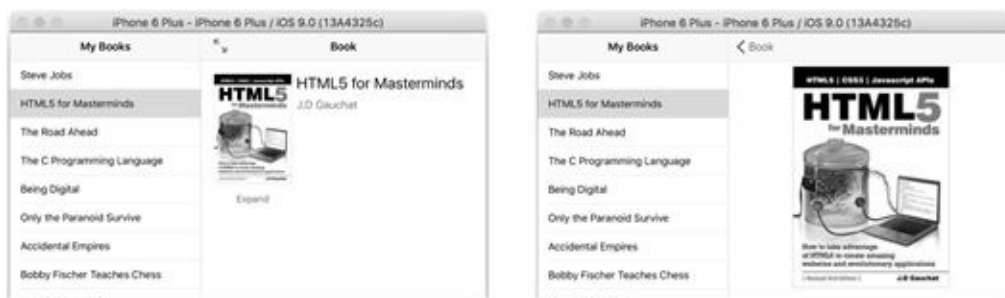


Figure 16-15: Navigation in the Detail view's area

Do It Yourself: Add a button with the title “Expand” below the Image View in the Detail view. Add a Scene to the Storyboard. Connect the button with the new view using a Show Segue. Assign the identifier “showPicture” to the Segue. Add an Image View to the new view, as shown in Figure 16-14. Create a subclass of the `UIViewController` class called `PictureViewController` and assign it to the new view. Complete the class with the code of Listing 16-11 and connect the Image View to the `bookCover` Outlet. Add the method of Listing 16-10 to the `DetailViewController` class of Listing 16-6. Run the application, select a book, and press the Expand button. You should see something similar to Figure 16-15.

Modal Views

Besides adding views to the stack of a Navigation Controller, as we did in the previous example, we can expand the interface with modal views and popovers. We have already studied modal views in Chapter 13. They are normal views but connected with a Segue called *Present Modally*. In iPhones and iPods, these views are always opened full screen and with the transition set by the Segue, but in iPads and iPhones 6 Plus in landscape mode they can be presented with designs that take advantage of larger screens. These designs are called *Presentation Styles*, and there are several available.

- **Full Screen** presents the view over the current interface and with the size of the screen.
- **Current Context** presents the view in the place and with the size of the designated area.
- **Page Sheet** presents the view as a sheet that emerges from the bottom of the screen.
- **Form Sheet** presents the view as a sheet over the current interface and in the center of the screen.
- **Over Full Screen** presents the view over the interface and with the size of the screen (but without hiding the current views).
- **Over Current Context** presents the view in the place and with the size of the designated area (but without hiding the current view).

The presentation style is established by the Segue. The options are available from the Attributes Inspector panel when the Segue is selected. The panel includes the option to select the Presentation Style and also change the type of Segue. Figure 16-16 illustrates the different options available in this panel for Show Segues and Present Modally Segues.

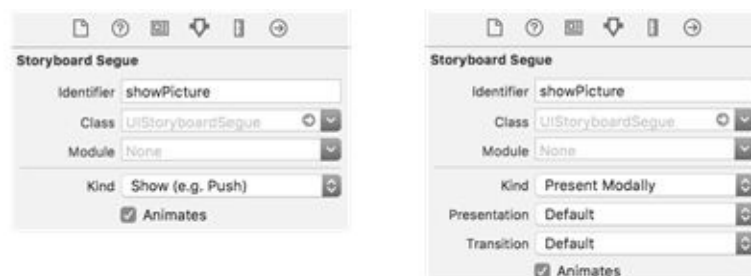


Figure 16-16: Segue configuration

If we turn the Show Segue introduced to our interface in Figure 16-14 into a Present Modally Segue, we will see something similar to Figure 16-17 below. The Segue looks the

same and it is still connecting the Expand button to the view, but now the view is not part of the Navigation Controller stack anymore, and therefore it doesn't include a Navigation Bar. Modal Segues open independent views that present information not associated to the main interface, and therefore they have to provide their own navigation tools. For this new interface, we added a Close button at the top to dismiss the view.

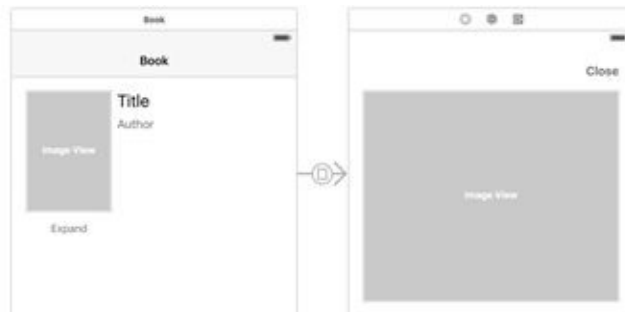


Figure 16-17: Interface with a Modal Segue

Depending on the presentation style selected and the device running the application, the view in Figure 16-17 will adopt different designs. For example, on an iPhone 6, the view is always shown full screen with a transition determined by the Transition option, but on an iPad the view will be shown with the selected presentation style (in iPhones 6 Plus in landscape mode, a presentation other than Full Screen always defaults to Page Sheet). Figure 16-18, next, illustrates what we see when we present the view with the Page Sheet mode in an iPad.

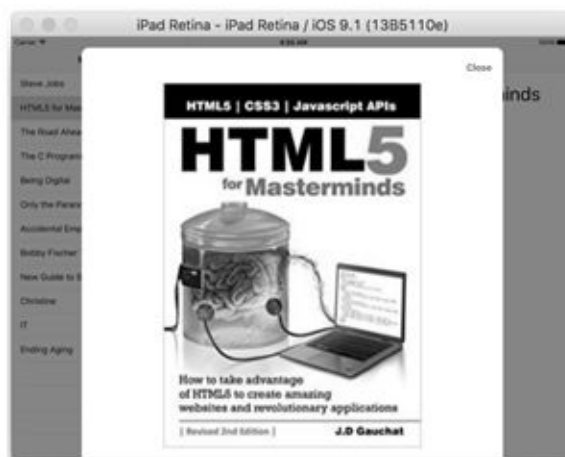


Figure 16-18: Page Sheet presentation style

Do It Yourself: Select the Show Segue created for our interface in Figure 16-14. Change its type to Present Modally from the Attributes Inspector panel (Figure 16-16). Click on the Presentation option and select the presentation style Page Sheet. Add a button with the title Close to the view managed by the `PictureViewController` class. Set the constraints between the button and the Image View. If you pinned the

elements to the margin of the main view, remember to set the vertical priority of the button higher than the Image View or otherwise the button will stretch vertically to occupy the space available. Run the application on the iPad simulator, select a book, and press the Expand button. You will see something similar to Figure 16-18.

When the view is presented as a Form Sheet or a popover (as we will see later), we can specify its size from the Attributes Inspector panel. The option is called Use Preferred Explicit Size, and is located at the bottom of the panel (Figure 16-19, number 1).



Figure 16-19: Use Preferred Explicit Size option

This is not the same option as the Simulated Size offered by the Size Inspector panel (Figure 11-13). The Simulated Size option determines the size of the view in the Storyboard, while the Use Preferred Explicit Size option sets the size the view is going to have when the application is running. We can set both options to the same values if we want to work on a view that matches what the user is going to see on the screen. Figure 16-20 shows the modal view from the previous example presented on an iPad as a Form Sheet and with an explicit size of 300 x 400.



Figure 16-20: Form Sheet presentation with an explicit size

At this moment the Close button does nothing, but its purpose is to provide the functionality to close the view. In Chapter 13 we studied how to do this with an Unwind Segue, but the `UIViewController` class also offers some properties and methods to present and

dismiss views programmatically that simplify our work when we don't need to pass data from one view controller to another or when we need to create a view dynamically.

modalPresentationStyle—This property sets or returns the presentation style. When we present a view programmatically, we first have to set the presentation style with this property. The property is an enumeration of type `UIModalPresentationStyle` with the values `FullScreen`, `PageSheet`, `FormSheet`, `CurrentContext`, `OverFullScreen`, `OverCurrentContext`, `Popover`, `None`, and `Custom`.

modalTransitionStyle—This property sets or returns the transition style. When we present a view programmatically, we first have to set the transition style with this property. The property is an enumeration of type `UIModalTransitionStyle` with the values `CoverVertical`, `FlipHorizontal`, `CrossDissolve`, and `PartialCurl`. Notice that some of these transitions are only available for the Full Screen presentation style.

presentViewController(controller, animated: animated, completion: block)—This method presents a modal view. The `controller` attribute is a reference to the view controller of the view we want to present, the `animated` attribute is a Boolean property that determines whether or not the process will be animated, and the `block` property is a conditional closure to be executed after the view is presented on the screen.

dismissViewControllerAnimated(animated, completion: block)—This method dismisses the view. The `animated` attribute is a Boolean value that determines whether or not the process will be animated, and the `block` attribute is a conditional closure to be executed after the view is closed.

To dismiss the view opened by the Modal Segue of Figure 16-17 we have to create an Action for the Close button that calls the `dismissViewControllerAnimated()` method. The following example implements the Action in the `PictureViewController` class.

Listing 16-12: Dismissing a modal view programmatically

```
import UIKit
```

```
class PictureViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    var selected: Int!
    override func viewDidLoad() {
        super.viewDidLoad()
        if selected != nil {
            let file = AppData.items[selected]
```

```

    bookCover.image = UIImage(named: file)
}
}
@IBAction func closePicture(sender: UIButton) {
    dismissViewControllerAnimated(true, completion: nil)
}
}

```

The view controller that presents the view is responsible for dismissing it, but we can call the `dismissViewControllerAnimated()` method from the presented view and the system will take care of asking the presenter to dismiss it. Listing 16-12 includes the `closePicture()` Action for the Close button. When the user taps on the button, the `dismissViewControllerAnimated()` method is executed and the view is closed.

Do It Yourself: Update the `PictureViewController` class with the code of Listing 16-12. Connect the Close button added to the interface in Figure 16-17 to the `closePicture()` Action. Run the application, select a book, press the Expand button, and then press the Close button to close the modal view.

Creating modal views in the Storyboard is easy, but there are times in which our application requires the creation of the views dynamically from code. There are at least three ways to do this. We can do it entirely from code, create separate files for the view and its view controller, or create an isolated view in the Storyboard and load it from the Storyboard's object. To demonstrate each alternative, we are going to open a view from a bar button in the Detail view called Help.

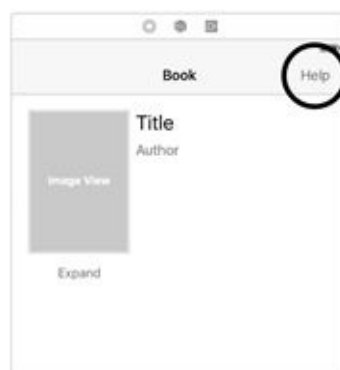


Figure 16-21: *Help button*

The first alternative we are going to explore is creating the view and its view controller from code. This demands a lot of writing but in some circumstances it may be the only way to achieve our goal. For this purpose we have to connect the Help button to an Action

in the view controller that is going to present the view (**DetailViewController** in our example). Inside the Action we have to create everything, from the view and the view controller object to the elements of the interface and their constraints. Our example creates a view with the message “Press Expand to maximize the cover” and a Close button to dismiss it.

Listing 16-13: Creating a modal view from code

```
import UIKit

class DetailViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    var selected: Int!
    var controllerHelp: UIViewController!

    override func viewDidLoad() {
        super.viewDidLoad()
        let button = splitViewController?.displayModeButtonItem()
        navigationItem.leftBarButtonItem = button
        navigationItem.leftItemsSupplementBackButton = true
        if selected != nil {
            let file = AppData.items[selected]
            bookCover.image = UIImage(named: file)
            if let data = AppData.itemsData[file] {
                bookTitle.text = data[0]
                bookAuthor.text = data[1]
            }
        }
    }

    override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        if segue.identifier == “showPicture” {
            let controller = segue.destinationViewController as! PictureViewController
            controller.selected = selected
        }
    }

    @IBAction func showHelp(sender: UIBarButtonItem) {
        let label = UILabel(frame: CGRectZero)
        label.translatesAutoresizingMaskIntoConstraints = false
        label.text = “Press Expand to maximize the cover”
        label.textAlignment = .Center
        label.numberOfLines = 2
        label.font = UIFont.systemFontOfSize(24)
```

```

let closeButton = UIButton(type: .System)
closeButton.translatesAutoreresizingMaskIntoConstraints = false
closeButton.setTitle("Close", forState: .Normal)
closeButton.addTarget(self, action: "closeHelp", forControlEvents: .TouchUpInside)
controllerHelp = UIViewController()
controllerHelp.modalPresentationStyle = .PageSheet
controllerHelp.view.backgroundColor = UIColor.whiteColor()
controllerHelp.view.addSubview(label)
controllerHelp.view.addSubview(closeButton)
var constraints = NSLayoutConstraint.constraintsWithVisualFormat("H:|-[mylabel]-|", options: [], metrics:
nil, views: ["mylabel": label])
controllerHelp.view.addConstraints(constraints)
constraints = NSLayoutConstraint.constraintsWithVisualFormat("H:|-[mybutton]-|", options: [], metrics: nil,
views: ["mybutton": closeButton])
controllerHelp.view.addConstraints(constraints)
constraints = NSLayoutConstraint.constraintsWithVisualFormat("V:|-20-[mylabel]-20-[mybutton]", options:
[], metrics: nil, views: ["mylabel": label, "mybutton": closeButton])
controllerHelp.view.addConstraints(constraints)
presentViewController(controllerHelp, animated: true, completion: nil)
}
func closeHelp() {
    controllerHelp.dismissViewControllerAnimated(true, completion: nil)
}
}

```

The code of Listing 16-13 expands the view controller created for the Detail view in previous examples. We have added a property and two methods. The `controllerHelp` property stores a reference to the view controller we are going to create for the modal view, and the `showHelp()` and `closeHelp()` methods are the Actions for the Help and Close buttons, respectively. When the Help button in the Detail view is pressed, we perform a series of tasks to get our modal view ready. First we create the `UILabel` and `UIButton` objects for the label and the button. The objects are instantiated and then configured with the right values. We also assign the target and action for the button with the `addTarget()` method to execute the `closeHelp()` method when the button is pressed (see `UIControl` in Chapter 4). Next, we create the `UIViewController` object to control the view. After the object is initialized, we set its presentation style as `PageSheet`, configure its view with a white background color, and add the label and the button as subviews. The last step to get the view ready is to set the constraints. We generate horizontal and vertical Space constraints for both elements, the label and the button, and finally present the view with the `presentViewController()` method. The result is illustrated in Figure 16-22.



Figure 16-22: Modal view created from code

Do It Yourself: Add a bar button item to the Navigation Bar of the Detail view called Help. Update the `DetailViewController` class with the code of Listing 16-13. Run the application in the iPad simulator, select a book, and press the Help button. You should see the modal view emerge from the bottom.

The next alternative is to create a view controller with its own view and then load it from code. This greatly simplifies our work since we have a view in Interface Builder that we can edit the same way as the views in the Storyboard. The file for the view controller is created as always; we go to the File menu at the top of the screen, select New/File... and then click on Cocoa Touch Class, but in the window where we declare the name and the class for the view controller we have to activate the option *Also create XIB file*. This option generates, along with the view controller's file, a file containing a single main view.

For our example, we called the class `HelpViewController`, therefore our project now has two files: `HelpViewController.swift` and `HelpViewController.xib`. The `HelpViewController.xib` file works exactly like the Storyboard, but it is capable of managing only a single view. If we click on this file, the view is displayed in the Editor Area and we can begin adding elements to it. Figure 16-23 shows the view created for this example. We have added the label, the button, and the necessary constraints.



Figure 16-23: View created with a XIB file

The view is already associated with its view controller, so we can connect the elements with Outlets and Actions right away. The following example adds an Action to the **HelpViewController** class to dismiss the view when the Close button is pressed.

Listing 16-14: Dismissing the view

```
import UIKit

class HelpViewController: UIViewController {
    @IBAction func closeHelp(sender: UIButton) {
        dismissViewControllerAnimated(true, completion: nil)
    }
}
```

Now that we have created the view and prepare its view controller, we have to load it and present it from the Detail view. Because everything was done on Interface Builder, all the **DetailViewController** has to do is to create an instance of the view controller and present its view on the screen with the **presentViewController()** method.

Listing 16-15: Instantiating the view controller and presenting its view on the screen

```
import UIKit

class DetailViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    var selected: Int!
    override func viewDidLoad() {
        super.viewDidLoad()
        let button = splitViewController?.displayModeButtonItem()
        navigationItem.leftBarButtonItem = button
        navigationItem.leftItemsSupplementBackButton = true
        if selected != nil {
            let file = AppData.items[selected]
            bookCover.image = UIImage(named: file)
            if let data = AppData.itemsData[file] {
                bookTitle.text = data[0]
                bookAuthor.text = data[1]
            }
        }
    }
}
```

```

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showPicture" {
        let controller = segue.destinationViewController as! PictureViewController
        controller.selected = selected
    }
}

@IBAction func showHelp(sender: UIBarButtonItem) {
    let controllerHelp = HelpViewController()
    controllerHelp.modalPresentationStyle = .PageSheet
    presentViewController(controllerHelp, animated: true, completion: nil)
}
}

```

Do It Yourself: Go to the File menu at the top of the screen. Select the options New/File... and click on Cocoa Touch Class. In the next window insert the name **HelpViewController** for the class, select **UIViewController** for the superclass and activate the option Also create XIB file. Your project now has two more files: **HelpViewController.swift** and **HelpViewController.xib**. Click on the **HelpViewController.xib** file. You should see an empty view, like the one in Figure 16-23. Add a label, a button and the necessary constraints. Connect the button to the **HelpViewController** class with an Action called **closeHelp()**. Complete the Action with the code of Listing 16-14. Replace the code in your **DetailViewController** class by the code of Listing 16-15. Test the application in the iPad simulator. It should create a modal view similar to Figure 16-22.

The last and final alternative we are going to explore involves the creation of the views in the Storyboard. This is not the same we have done so far, because these views are not going to be connected to any other view and therefore they are not part of the path set by the Storyboard.

As we explained in previous chapters, when the app is executed the Storyboard is loaded and processed. The objects representing the views and their content are created and connected with each other, but the system also creates an object that represents the Storyboard itself. This object is created from the **UIStoryboard** class and offers the following methods to access its content.

instantiateInitialViewController()—This method instantiates and returns the view controller of the initial view in the Storyboard.

instantiateViewControllerWithIdentifier(identifier)—This method instantiates and returns the view controller of the view in the Storyboard identified with the string specified by the **identifier** attribute.

Single views are added to the Storyboard the same way we do with other views, but they are not connected to the Storyboard itself. Once we added the view, the process to generate its content and view controller are the same. For our example we have added the view of Figure 16-24 to our interface with a view controller called `SingleViewController`.



Figure 16-24: Single view in the Storyboard

The view controller for this view only needs an Action for the button, as shown next.

Listing 16-16: Dismissing the single view

```
import UIKit

class SingleViewController: UIViewController {
    @IBAction func closeHelp(sender: UIButton) {
        dismissViewControllerAnimated(true, completion: nil)
    }
}
```

To be able to present the view from the Detail view we have to assign an identifier for the view from the Identity Inspector panel. The option is called *Storyboard ID*. For our example, we called the view “helpView”, as shown in Figure 16-25.

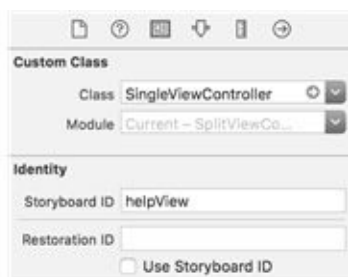


Figure 16-25: Option to assign the identifier to views

The `DetailViewController` class has to instantiate the view controller for this view with the `instantiateViewControllerWithIdentifier(identifier)` method and present the view on the screen with the `presentViewController()` method.

Listing 16-17: Instantiating the view from the Storyboard

import UIKit

```
class DetailViewController: UIViewController {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    var selected: Int!
    override func viewDidLoad() {
        super.viewDidLoad()
        let button = splitViewController?.displayModeButtonItem()
        navigationItem.leftBarButtonItem = button
        navigationItem.leftItemsSupplementBackButton = true
        if selected != nil {
            let file = AppData.items[selected]
            bookCover.image = UIImage(named: file)
            if let data = AppData.itemsData[file] {
                bookTitle.text = data[0]
                bookAuthor.text = data[1]
            }
        }
    }
    override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        if segue.identifier == "showPicture" {
            let controller = segue.destinationViewController as! PictureViewController
            controller.selected = selected
        }
    }
    @IBAction func showHelp(sender: UIBarButtonItem) {
        if let story = storyboard {
            let controllerHelp = story.instantiateViewControllerWithIdentifier("helpView") as! SingleViewController
            controllerHelp.modalPresentationStyle = .PageSheet
            presentViewController(controllerHelp, animated: true, completion: nil)
        }
    }
}
```

To provide access to the `UINavigationController` object, the `UIViewController` class offers the `storyboard` property. The property returns a reference to the `UINavigationController` object the view controller was created from. In Listing 16-17 we read this property and execute the `instantiateViewControllerWithIdentifier(identifier)` method to create an instance of the `SingleViewController` class. The instance is stored in the `controllerHelp` constant and its view is presented in the screen as we did before.

Do It Yourself: Add a new view to the Storyboard. Add a label and a button with their respective constraints to the view (Figure 16-24). Create a new `UIViewController` subclass called `SingleViewController`. Select the view, open the Identity Inspector panel, and assign the subclass to the view and the string “helpView” to the Storyboard ID option (Figure 16-25). Connect the Close button to the `SingleViewController` class with an Action called `closeHelp()` and complete the class with the code of Listing 16-16. Replace the code in your `DetailViewController` class by the code of Listing 16-17. Test the application in the iPad simulator. It should create a modal view similar to Figure 16-22.

UIPresentationController

UIKit defines a class called `UIPresentationController` to manage the presentations of modal views, including popovers and the Alert Views we will study in the next chapter. When a modal view is created and presented by any of the techniques we just introduced, an object of the `UIPresentationController` class is automatically created to manage the presentation. The object is assigned to a property of the view controller that is being presented called `presentationController`. From this property we can access the object and read its current configuration. The following are some of its properties and methods.

presentingViewController—This property returns a reference to the view controller that initiated the presentation.

presentedViewController—This property returns a reference to the view controller that is being presented.

containerView—This property returns a reference to the view in which the presentation occurs.

presentationStyle—This property returns the presentation style used to present the view. It is an enumeration of type `UIModalPresentationStyle` with the values `FullScreen`, `PageSheet`, `FormSheet`, `CurrentContext`, `OverFullScreen`, `OverCurrentContext`, `Popover`, `None`, and `Custom`.

adaptivePresentationStyle()—This method returns the presentation style used when the horizontal Size Class becomes compact.

We can also designate a delegate for the **UIPresentationController** object to change its configuration according to the current state of the interface. The delegate must adopt the **UIAdaptivePresentationControllerDelegate** protocol and implement any of the following methods.

adaptivePresentationStyleForPresentationController(controller)—This method is called on the delegate to know the presentation style to use when the horizontal Size Class becomes compact.

presentationController(controller, viewControllerForAdaptivePresentationStyle: style)—This method is called on the delegate to get the view controller to present for the style determined by the **style** attribute. The method is used to assign a different view to each presentation style.

The **UIPresentationController** object has two presentation styles, one for the normal state and one for the adaptive state. The adaptive state is currently considered to be the state in which the horizontal Size Class is compact. This is the state we will find in most iPhones and the iPod Touch. When a modal view is presented on these devices, the presentation style is automatically changed to **FullScreen**, but we can specify the style we want implementing the protocol methods. The advantage of these methods is that they are called not only to know the presentation style to use in the adaptive state but also to get the view controller we want to present, and therefore we can designate a completely different view controller for each state. The following example adds another view to our interface to present as the modal view when we are in an adaptive state.

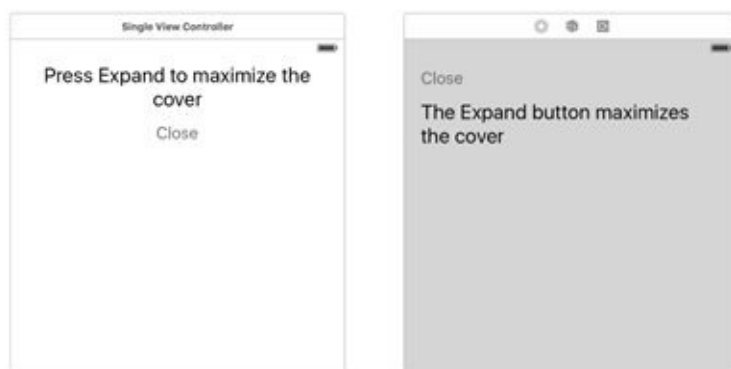


Figure 16-26: Two views for the same modal view

The view on the left is the same view we introduced in Figure 16-24, and the view on

the right is the new view we will open in the adaptive state. What we did for this view is add a Visual Effect View in the background so on iPads the modal view will have a solid white background but on iPhones it will be translucent and blurry. To control this new view we have created a class called **iPhoneViewController**, the only purpose of which is to close the view when the Close button is pressed.

Listing 16-18: Dismissing the adaptive view

```
import UIKit

class iPhoneViewController: UIViewController {
    @IBAction func closeHelp(sender: UIButton) {
        dismissViewControllerAnimated(true, completion: nil)
    }
}
```

We now have the two views we are going to present when the Help button is pressed on the Detail view, so all that is left to do is to implement the protocol methods in the **DetailViewController** class to load the second view every time the horizontal class goes compact.

Listing 16-19: Showing an alternative modal view

```
import UIKit

class DetailViewController: UIViewController, UIAdaptivePresentationControllerDelegate {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    var selected: Int!
    override func viewDidLoad() {
        super.viewDidLoad()
        let button = splitViewController?.displayModeButtonItem()
        navigationItem.leftBarButtonItem = button
        navigationItem.leftItemsSupplementBackButton = true
        if selected != nil {
            let file = AppData.items[selected]
            bookCover.image = UIImage(named: file)
            if let data = AppData.itemsData[file] {
                bookTitle.text = data[0]
                bookAuthor.text = data[1]
            }
        }
    }
}
```

```

}
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showPicture" {
        let controller = segue.destinationViewController as! PictureViewController
        controller.selected = selected
    }
}

@IBAction func showHelp(sender: UIBarButtonItem) {
    if let story = storyboard {
        let controllerHelp = story.instantiateViewControllerWithIdentifier("helpView") as! SingleViewController
        controllerHelp.modalPresentationStyle = .FormSheet
        let presentation = controllerHelp.presentationController
        presentation?.delegate = self
        presentViewController(controllerHelp, animated: true, completion: nil)
    }
}

func adaptivePresentationStyleForPresentationController(controller: UIPresentationController) ->
UIModalPresentationStyle {
    return .OverFullScreen
}

func presentationController(controller: UIPresentationController, viewControllerForAdaptivePresentationStyle
style: UIModalPresentationStyle) -> UIViewController? {
    var controller: iPhoneViewController!
    if style == .OverFullScreen {
        if let story = storyboard {
            controller = story.instantiateViewControllerWithIdentifier("iPhoneView") as! iPhoneViewController
        }
    }
    return controller
}
}

```

As always, the first thing we have to do is to adopt the protocol and declare the object as the delegate. Although we have done this dozens of times before, in this opportunity the object we want our class to be delegate of is the **UIPresentationController** object of the view controller that is being presented, so we have to get a reference to this object from the **presentationController** property first and then modify its **delegate** property. In our example, we have to access this property from the instance created by the **showHelp()** method. We get the reference first, store it in the **presentation** constant, and then assign **self** to its **delegate** property. With the delegate set, we can now implement its methods.

By default the presentation style set for the adaptive state is **FullScreen**. This is why in previous examples, the modal views were opened full screen in iPhones and as **PageSheet** in

iPads. Usually this behaviour is the right one for our app, but this time we have created an alternative view with a transparent background to let the user see a blurry image of the Detail view in the back, so we need to change the presentation style for the adaptive state to **OverFullScreen**. When the view is in this state, the **UIPresentationController** object calls the **adaptivePresentationStyleForPresentationController(controller)** method to know what presentation style to use, so we have to return the **OverFullScreen** value from this method and the presentation is set to the style we want.

At this point the presentation style is different for an iPad than an iPhone, but the view shown on the screen is always the same. To specify a different view for the adaptive state we have to implement the **presentationController(controller, viewControllerForAdaptivePresentationStyle: style)** method. This method is called by the **UIPresentationController** object to know which view it has to present in every style, so it is our chance to instantiate the view controller of the second view we just added in Figure 16-26 and return it when the style is **OverFullScreen** (we have identified this view in the Storyboard with the string “iPhoneView”). The result is shown in Figure 16-27. If we open the view on an iPad or an iPhone 6 Plus in landscape mode, the original view is presented on the screen, but if we do the same in an iPhone or an iPod, the new view with its translucent background is presented instead.

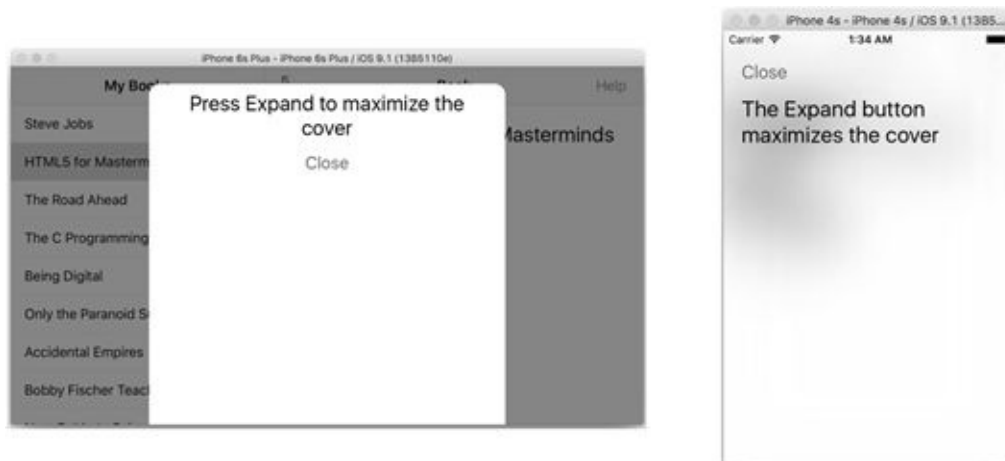


Figure 16-27: Alternative views for different devices

Do It Yourself: Add a new view to the Storyboard. Select its main view, go to the Attributes Inspector panel and assign the Clear color as its background color. Add a Visual Effect View and pin it to the edges. Also add a button and a label to reproduce the interface of Figure 16-26. Create a new **UIViewController** subclass called **iPhoneViewController**. Select the Scene, open the Identity Inspector panel, assign the subclass to the view, and give the view the identifier “iPhoneView”. Connect the Close button to an Action called **closeHelp()** and complete the class with the code of Listing 16-18. Update the **DetailViewController** class with the code of Listing 16-19. Run the application in the iPad and iPhone simulator to see how the modal views are presented.

IMPORTANT: The delegate methods are only called when the original presentation style is `FormSheet`, `Popover`, or `Custom`. This is the reason why we assigned the presentation style `FormSheet` to the original view instead of the `PageSheet` used in previous examples.

UIPopoverPresentationController

The `UIPresentationController` objects have a presentation style to present the modal views as popovers, but when we select this style, the presentation object is not created from the `UIPresentationController` class but from one of its subclasses called `UIPopoverPresentationController`. This is because the system needs more information to present a popover, such as its margins or the anchor that determines its position. The following are some of the properties added by the subclass to configure the popover.

popoverLayoutMargins—This property sets or returns the margins that define the maximum portion of the screen designated for the popover. It is a value of type `UIEdgeInsets`.

permittedArrowDirections—This property sets or returns a value that determines the position of the popover’s arrow. Every popover contains a little arrow that points to the element that triggered the presentation. This property is used to set the position of that arrow before the popover is presented. It is a structure of type `UIPopoverArrowDirection` with the properties `Up`, `Down`, `Left`, `Right`, `Any`, and `Unknown`.

arrowDirection—This property returns the direction of the arrow. It is a structure of type `UIPopoverArrowDirection` with the properties `Up`, `Down`, `Left`, `Right`, `Any`, and `Unknown`.

sourceView—This property sets or returns the view to which the popover is anchored. It is an optional value of type `UIView`.

sourceRect—This property sets or returns the rectangle of the view to which the popover is anchored. The rectangle is considered to be inside the view assigned to the `sourceView` property. The property is of type `CGRect`.

barButtonItem—This property sets or returns the bar button item to which the popover is anchored. It is an optional value of type `UIBarButtonItem`.

The `UIPopoverPresentationController` object can also work with a delegate to report the state of the popover. The delegate must adopt the `UIPopoverPresentationControllerDelegate` protocol, which defines the following methods.

prepareForPopoverPresentation(popoverPresentationController)—This method is called to notify the delegate that the popover is about to be presented on the screen.

popoverPresentationControllerShouldDismissPopover(popoverPresentationController)—This method is called to know if the popover should be dismissed or not.

popoverPresentationControllerDidDismissPopover(popoverPresentationController)—This method is called to notify the delegate that the popover was dismissed.

popoverPresentationController(popoverPresentationController, willRepositionPopoverToRect: rect, inView: view)—This method is called to notify the delegate that the popover is going to be repositioned in the rectangle determined by the **rect** attribute. The attribute is of type **CGRect**.

The steps to add a popover to the Storyboard are the same as for modal views, with the exception that we have to use a specific Segue called *Present As Popover*. Figure 16-28 shows a small view connected to a button in our Detail view with this type of Segue. For this example we have created a button called Total Books below the author's label and a view with an explicit size of 300 x 100.

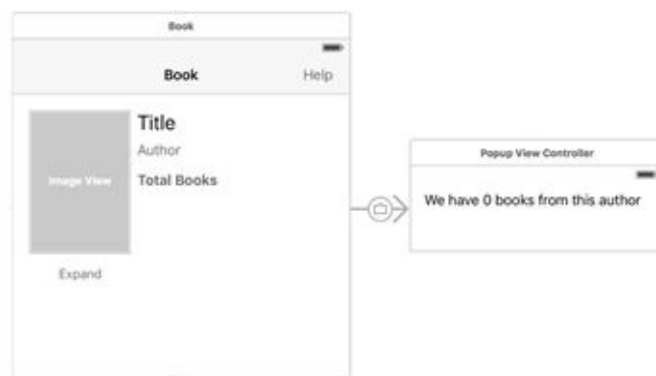


Figure 16-28: *Popover on the Storyboard*

The Segue assigns values by default to the **UIPopoverPresentationController** object, but we can modify some of them from the Attributes Inspector panel.

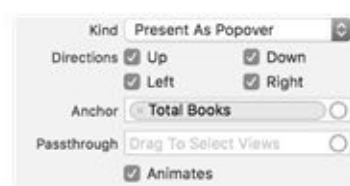


Figure 16-29: *Popover configuration*

The most important option is the `Directions`. This indicates where the arrow that points to the element that triggered the presentation is going to be located. For example, if we position the arrow at the top (Up), the view is going to be positioned below the element. By default, the `Segue` activates all the values to let the system decide the most appropriate, but we can suggest a specific position by activating only the ones we want.

The popover of our example counts the number of books that belong to the selected author and shows a message with the value on the screen, so we have to pass the index of the selected book from the Detail view to the popover and then perform a search in the model from the popover's view controller to get the total of books available. The first step is to modify the `prepareForSegue()` method of the `DetailViewController` class to send the index of the book when the popover's `Segue` is triggered. The `Segue` was identified with the "showPopover" string (notice that we are still working over the same example introduced in this chapter, so the code for the "showPicture" `Segue` is still available).

Listing 16-20: Preparing the popover

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?){
    if segue.identifier == "showPicture" {
        let controller = segue.destinationViewController as! PictureViewController
        controller.selected = selected
    } else if segue.identifier == "showPopover" {
        let controller = segue.destinationViewController as! PopoverViewController
        controller.selected = selected
    }
}
```

Next, we have to create a new view controller for the popover and assign it to the view. The view controller has to perform the search to count the books available from the selected author and show it on the screen. We called it `PopoverViewController`.

Listing 16-21: Counting books

```
import UIKit

class PopoverViewController: UIViewController {
    @IBOutlet weak var messageLabel: UILabel!
    var selected: Int!
    override func viewDidLoad() {
        super.viewDidLoad()
        var author = ""
        if selected != nil {
```

```

let file = AppData.items[selected]
if let data = AppData.itemsData[file] {
    author = data[1]
}
let search = AppData.itemsData.filter({ (item) in
    let data = item.1
    if data[1] == author {
        return true
    }
    return false
})
let total = search.count
messageLabel.text = "We have \ \(total) books from this author"
}
}
}

```

The code of Listing 16-21 gets the name of the selected author first and then calls the `filter()` method to search for the name inside the `itemsData` dictionary of our model (see Listing 15-2). The `filter()` method returns an array with the items that match the name of the author, so we count the elements in the array and assign a string containing this value to the `messageLabel` label. In an iPad, we will see something similar to Figure 16-30.

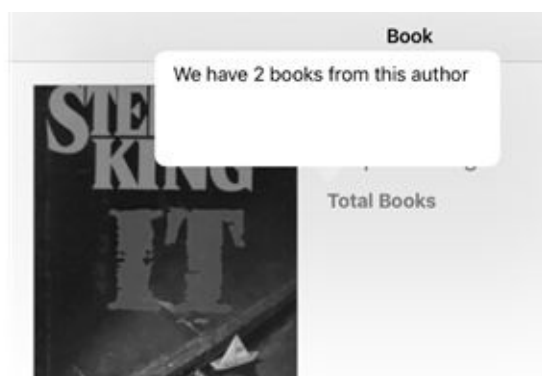


Figure 16-30: Popover with Down arrow

Do It Yourself: Add a button with the title Total Books below the author’s label in the Detail view. Add a new Scene to the interface. Add a label to the view, as shown in Figure 16-28. Connect the button to the view with a Present As Popover Segue. Give the Segue the identifier “showPopover”. Select the new view and modify its explicit size from the Attributes Inspector panel (Figure 16-21). You can also modify the size in the Storyboard from the Size Inspector panel (Figure 11-13). Create a subclass of `UIViewController` called `PopoverViewController` and assign it to the new view.

Complete the subclass with the code of Listing 16-21. Connect the label to the `messageLabel` Outlet. Replace the `prepareForSegue()` method in the `DetailViewController` class of Listing 16-19 by the method of Listing 16-20. Run the application, select a book, and press the Total Books button. You should see something similar to Figure 16-30.

If we let the Segue with the configuration by default, the system decides where to position the arrow and the view for us. In this case, it decided that it was better to present the view over the Total Books button. But in this place the view covers the title and the author. To move the view below the button and keep the labels visible, we have to deactivate the directions Left, Right, and Down from the Attributes Inspector panel and let only the Up option activated. This forces the system to show the view where we want, but there is still a problem. The view now covers the button.

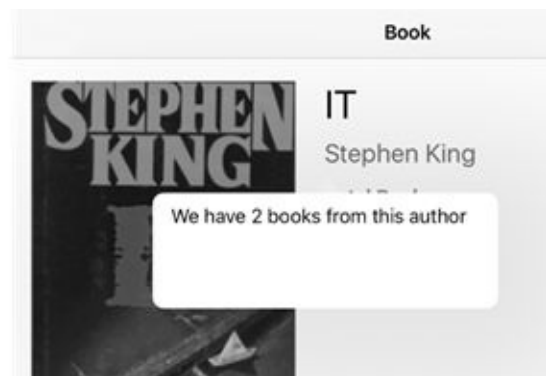


Figure 16-31: Popover with Up arrow

This occurs because the anchor point is always at the coordinates 0, 0 of the button's frame, independently of the direction of the arrow. To modify the anchor point, we have to assign a new `CGRect` value to the `sourceRect` property of the `UIPopoverPresentationController` object. The following are the modifications we have to introduce to the `prepareForSegue()` method of the `DetailViewController` to set the anchor at the bottom of the Total Books button.

Listing 16-22: Moving the anchor point

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?){
    if segue.identifier == "showPicture" {
        let controller = segue.destinationViewController as! PictureViewController
        controller.selected = selected
    } else if segue.identifier == "showPopover" {
        let controller = segue.destinationViewController as! PopoverViewController
        controller.selected = selected
        let presentation = controller.presentationController as! UIPopoverPresentationController
        let frame = presentation.sourceView?.bounds
        presentation.sourceRect = frame!
    }
}
```

```
}  
}
```

The `presentationController` property returns a `UIPresentationController` object that we have to cast as `UIPopoverPresentationController` to have access to the properties of this class. The properties we need are `sourceView` and `sourceRect`. The `sourceView` property gives us access to the `UIView` object to which the popover is anchored, and the `sourceRect` property is the rectangle inside that view to which the popover's arrow points to. Therefore, to displace the arrow and the view we have to assign a new `CGRect` value to the `sourceRect` property. In the `prepareForSegue()` method of Listing 16-22 we get the internal frame of the view from its `bounds` property and assign it to the `sourceRect` property. This sets the rectangle as the size of the view and therefore the arrow points to the bottom and the center of the button, positioning the view below. The result is shown in Figure 16-32.

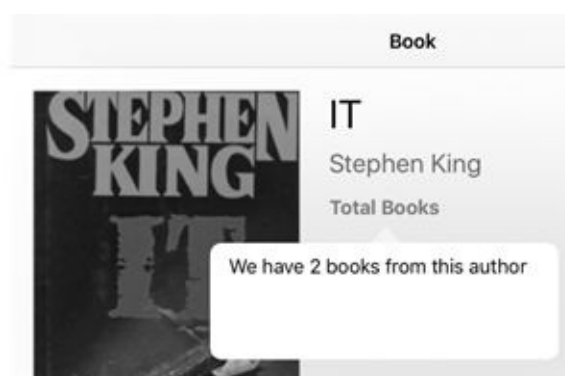


Figure 16-32: Position of popover set from code

Do It Yourself: Select the Segue that connects the Total Books button with the popover view, go to the Attributes Inspector panel and deactivate the directions Left, Right, and Down. Replace the `prepareForSegue()` method in the `DetailViewController` class by the method of Listing 16-22. Run the application in the iPad simulator, select a book, and then press the Total Books button. You should see something similar to Figure 16-32.

This works fine on iPads. The view is presented as a popover and it can be dismissed tapping anywhere else on the screen. But in a compact horizontal Size Class, the popover is shown full screen and there is no way to dismiss it. To solve this problem, we can implement the protocol methods we used before for modal views. One alternative is to return the `None` value from the `adaptivePresentationStyleForPresentationController(controller)` method. This tells the `UIPresentationController` object that we want the view to be presented with its original presentation style, so the view always looks like a popover, no matter the Size Class.

Because we have already declared the `DetailViewController` as the delegate of the view

opened by the Help button (see Figure 16-26 and Listing 16-19), we are going to designate the **PopoverViewController** as the delegate this time. But there is a catch. The **UIPopoverPresentationController** includes its own **delegate** property but for objects that adopt the **UIPopoverPresentationControllerDelegate** protocol and the protocol we want to adopt is **UIAdaptivePresentationControllerDelegate**, so we have to assign the object to the **delegate** property of the **UIPresentationController** object (the object without casting). The following is an update of the **prepareForSegue()** method that designates this delegate.

Listing 16-23: Assigning the delegate

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?){
    if segue.identifier == "showPicture" {
        let controller = segue.destinationViewController as! PictureViewController
        controller.selected = selected
    } else if segue.identifier == "showPopover" {
        let controller = segue.destinationViewController as! PopoverViewController
        controller.selected = selected
        let presentation = controller.presentationController as! UIPopoverPresentationController
        let frame = presentation.sourceView?.bounds
        presentation.sourceRect = frame!
        controller.presentationController?.delegate = controller
    }
}
```

At the end of the method of Listing 16-23 we get a reference to the **UIPresentationController** object assigned to the **UIPopoverViewController** object and declare its delegate to be the same **UIPopoverViewController** object. Now we have to adopt the **UIAdaptivePresentationControllerDelegate** protocol from the class and implement its method.

Listing 16-24: Implementing the protocol's methods

```
import UIKit

class PopoverViewController: UIViewController, UIAdaptivePresentationControllerDelegate {
    @IBOutlet weak var messageLabel: UILabel!
    var selected: Int!

    override func viewDidLoad() {
        super.viewDidLoad()
        var author = ""
        if selected != nil {
            let file = AppData.items[selected]
            if let data = AppData.itemsData[file] {
```

```

author = data[1]
}
let search = AppData.itemsData.filter({ (item) in
let data = item.1
if data[1] == author {
return true
}
return false
})
let total = search.count
messageLabel.text = "We have \ \(total) books from this author"
}
}

func adaptivePresentationStyleForPresentationController(controller: UIPresentationController) ->
UIModalPresentationStyle {
    return .None
}
}

```

The new **PopoverViewController** is practically the same as before, we just adopted the protocol and return the value **None** from the **adaptivePresentationStyleForPresentationController(controller)** method. From now on, every time the popover is about to be presented in a compact horizontal Size Class, the **UIPopoverPresentationController** object receives the **None** value and therefore does not adapt the presentation style. Figure 16-33 shows what the popover looks like on an iPhone in portrait.



Figure 16-33: Popover on an iPhone in portrait mode

Do It Yourself: Replace the **prepareForSegue()** method in the **DetailViewController** class by the method of Listing 16-23. Update the **PopoverViewController** class with the code of Listing 16-24. Run the application in the iPhone simulator, select a book, and press

the Total Books button. You should see something similar to Figure 16-33.

That was a simple solution, but popovers don't look good in devices with small screens. The best alternative is to present a different view for every style, as we did before for modal views. For example, we can add a new single view to the Storyboard with a Close button and the same message as the popover and then open this view when the presentation style is **FullScreen**.



Figure 16-34: Alternative view for the popover

The view controller for this view shares the same code with the **PopoverViewController** class, but it also includes an Action for the Close button to dismiss the view. We called it **SinglePopoverViewController**.

Listing 16-25: Controlling the alternative view

```
import UIKit

class SinglePopoverViewController: UIViewController {
    @IBOutlet weak var messageLabel: UILabel!
    var selected: Int!
    override func viewDidLoad() {
        super.viewDidLoad()
        var author = ""
        if selected != nil {
            let file = AppData.items[selected]
            if let data = AppData.itemsData[file] {
                author = data[1]
            }
            let search = AppData.itemsData.filter({ (item) in
                let data = item.1
                if data[1] == author {
```

```

        return true
    }
    return false
})
let total = search.count
messageLabel.text = "We have \ \(total) books from this author"
}
}
@IBAction func closePopover(sender: UIButton) {
    dismissViewControllerAnimated(true, completion: nil)
}
}

```

To present this view when the popover is shown full screen, we have to implement the `presentationController(controller: UIPresentationController, viewControllerForAdaptivePresentationStyle: style) method` in the delegate (the `PopoverViewController` class).

Listing 16-26: Presenting the alternative view

```

func presentationController(controller: UIPresentationController, viewControllerForAdaptivePresentationStyle style:
UIModalPresentationStyle) -> UIViewController? {
    var controller: SinglePopoverViewController!
    if style == .FullScreen || style == .FormSheet {
        if let story = storyboard {
            controller = story.instantiateViewControllerWithIdentifier("singlePopover") as! SinglePopoverViewController
            controller.selected = selected
        }
    }
    return controller
}

```

Notice that we can't implement this method along with the method of Listing 16-24. When the `UIPresentationController` object receives the value `None` from the previous method, it considers that as an order not to adapt the presentation style and therefore never calls the method of Listing 16-26. Also, besides the `FullScreen` style we have to check for the `FormSheet` style. This is because iPhones 6 Plus in landscape mode adapts the presentation style to `FormSheet` instead of `FullScreen`.

Do It Yourself: Add a new Scene to the Storyboard. Include a button and a label, as shown in Figure 16-34. Create a subclass of `UIViewController` with the name `SinglePopoverViewController` and assign it to the view. Give to the view the identifier

“singlePopover” (Figure 16-25). Complete the subclass with the code of Listing 16-25 and connect the Close button to the Action. Erase the `adaptivePresentationStyleForPresentationController(controller)` method from the `PopoverViewController` class and add the method of Listing 16-26. Run the application in the iPhone simulator. Open the popover in portrait mode. You should see the view of Figure 16-34 full screen instead of a popover.

Chapter 17

Alert Views

17.1 Alert Views

Alert views are predefined modal views that can display messages and receive input from the user. Their purpose is to convey important information that requires immediate attention. For example, an Alert View may be used to ask confirmation from the user before deleting a file or data from the model. They may include a title, message, Text Fields, and buttons, and be presented full screen or as popovers in some devices. UIKit includes a subclass of the `UIViewController` class called `UIAlertController` to create these views, but there is no option to add them to the Storyboard, they have to be introduced from code using the following initializer.

`UIAlertController(title: title, message: message, preferredStyle: style)`—

This initializer creates a `UIAlertController` object configured with the values assigned to the attributes. The **`title`** attribute is a string shown at the top of the view, the **`message`** attribute is a string for a message shown below the title, and the **`style`** attribute is an enumeration of type `UIAlertControllerStyle` with the values `Alert` and `ActionSheet`. This last attribute indicates the type for the view we want to create (Alert or Action Sheet).

The class also includes properties and methods to add and configure the view's elements.

`title`—This property sets or returns the title.

`message`—This property sets or returns the message

`actions`—This property returns an array with references to the actions (buttons) in the view.

`message`—This property returns an array with references to the Text Fields in the view.

`addAction(action)`—This method adds a new action to the view. Actions are objects of the `UIAlertAction` class that create buttons for the view with specific configurations and purposes.

`addTextFieldWithConfigurationHandler(handler)`—This method adds a Text Field to the view. The **`handler`** attribute is a closure that provides the configuration for the Text Field.

Alert Views present buttons along with the text to let the user respond. The buttons are

created from the `UIAlertAction` class. The class includes the following initializer and properties.

UIAlertAction(title: `title`, style: `style`, handler: `handler`)—This initializer creates a `UIAlertAction` object configured with the values assigned to the attributes. The **title** attribute is the title for the button. The **style** attribute is an enumeration of type `UIAlertActionStyle` that defines the purpose of the button. The possible values are **Default** (for a generic type of button), **Cancel** (for a button that cancels the request), and **Destructive** (for a button that performs a destructive task, such as deleting a file). The **handler** attribute is a closure that is executed when the button is pressed.

title—This property returns a string with the title of the button.

style—This property returns the type of the button. It is an enumeration of type `UIAlertActionStyle` with the values **Default**, **Cancel**, and **Destructive**.

enabled—This property sets or returns a Boolean value that determines whether the button is enabled or not.

There are two types of Alert Views available: Alert and Action Sheet. Alert Views of type Alert are presented in the center of the screen, like the Form Sheet modal views studied in Chapter 16, while those of type Action Sheet are anchored to the bottom of the screen in small devices like the iPhones or presented as popovers in devices with bigger screens like the iPads. They may both include messages and buttons, but only Alert types are capable of including Text Fields.

Alerts

Alert Views of type Alert are usually presented to report to the user that a specific task has been completed or something wrong happened. For example, we may show an Alert View of type Alert to report an error when the user tries to save the value of an empty Text Field. The following interface includes a Text Field and a button to let the user insert a name. If the user presses the button before inserting any text, the code shows an Alert View reminding the user that a name must be inserted.



Figure 17-1: Interface to test Alert Views

The view controller for this view has to check whether the Text Field contains text or not and show the alert if it is empty.

Listing 17-1: Presenting an Alert View

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var nameText: UITextField!
    @IBAction func saveName(sender: UIButton) {
        var text = nameText.text!
        text = text.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
        if text == "" {
            showAlert()
        } else {
            print("Value stored: \(text)")
            nameText.text = ""
        }
    }
}

func showAlert() {
    let alert = UIAlertController(title: "Error", message: "Insert the name in the field", preferredStyle: .Alert)
    let action = UIAlertAction(title: "OK", style: .Default, handler: nil)
    alert.addAction(action)
    presentViewController(alert, animated: true, completion: nil)
}
```

The view controller of Listing 17-1 includes an Outlet for the Text Field called **nameText** and an Action for the button called **saveName**. When the user presses the button, the Action checks the value of the field's **text** property. If the value is empty, it calls the **showAlert()** method to present an Alert View, otherwise it prints a message on the console and cleans the field.

Our **showAlert()** method is responsible for showing the Alert View. The first step is to

create the view controller for the Alert View with the `UIAlertController()` initializer. The initializer lets us define the title and the message for the view, and also its type (Alert in this case). Next, we have to create and add to the view a `UIAlertAction` object for every button we want to include. The initializer lets us define the title for the button, its purpose, and a closure that is going to be executed when the button is pressed. For this example we need only one button of type Default. We are not going to perform any task when the button is pressed so we declared the handler as `nil`. The action is added to the Alert View with the `addAction()` method and then the view is presented on the screen with the `presentViewController()` method, as we did for other modal views in Chapter 16. If we run the application and press the Save button before inserting any text in the field, an Alert View is opened to remind us that we have to insert the name, as shown in Figure 17-2.



Figure 17-2: Alert View of type Alert with a single button

Do It Yourself: Create a new Single View Application project. Add a label, a Text Field and a button to the main view. Connect the Text Field to the view controller with an Outlet called `nameText` and the button to an Action called `saveName()`. Complete the `ViewController` class with the code of Listing 17-1. Run the application and press the button. You should see something similar to Figure 17-2.

The Alert View can only be dismissed pressing one of its buttons. If we want to perform another task other than dismissing the view when the button is pressed, we have to provide a closure for the **handler** attribute of the `UIAlertAction` initializer. For example, the following code changes the background color of the Text Field after the OK button is pressed.

Listing 17-2: Changing the background color of the Text Field when the button is pressed

```
func showAlert() {  
    let alert = UIAlertController(title: "Error", message: "Insert the name in the field", preferredStyle: .Alert)  
    let action = UIAlertAction(title: "OK", style: .Default, handler: { (action) in
```

```

    self.nameText.backgroundColor = UIColor(red: 255.0/255.0, green: 230.0/255.0, blue: 230.0/255.0, alpha: 1.0)
  })
  alert.addAction(action)
  presentViewController(alert, animated: true, completion: nil)
}

```

The closure receives a reference to the `UIAlertAction` object, so we have to include an attribute for the closure to capture this reference and then declare the statements after the `in` instruction. In this example, we change the background color of the field to red.

IMPORTANT: Properties inside a closure must always be referenced from `self`. This is a requirement of Swift to remind us that referencing a property of the class from inside the closure could create a strong reference cycle (see Chapter 3). For more information on how to avoid strong reference cycles from a closure, visit our website and follow the links for this chapter.

We can add more buttons to the Alert View. The following example asks the user if the application should store the empty fields anyways and offers a Cancel button to cancel the operation.

Listing 17-3: Adding a Cancel button

```

func showAlert() {
    let alert = UIAlertController(title: "The field is empty", message: "Are you sure do you want to store an empty string?",
    preferredStyle: .Alert)

    let action = UIAlertAction(title: "Yes", style: .Default, handler: { (action) in
        print("Value stored")
    })
    alert.addAction(action)

    let cancel = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
    alert.addAction(cancel)

    presentViewController(alert, animated: true, completion: nil)
}

```

The `UIAlertController` object places the buttons inside the view in the order they were created in the code. We may add all the buttons we need, except for buttons of type `Cancel`, of which only one is permitted per view.

Because we didn't include a closure for the handler attribute of the Cancel button, the function of the button is just to dismiss the view, but all buttons can perform additional tasks, including this one.

Do It Yourself: Replace the `showAlert()` method of Listing 17-1 by the methods of Listings 17-2 and 17-3 to try each example.

Alert Views of type Alert may also include Text Fields. Text Fields are usually added to an Alert View to create sign-up or log-in forms. For instance, we can create an interface with an initial view that let the user log in, and a second view that is opened if the values inserted by the user coincide with the data we have in our model. The interface of Figure 17-3 includes a welcoming window with a button at the top called My Account to open an Alert View to insert an email and a password.



Figure 17-3: Interface to log in

Text Fields are added directly to the Alert View with the `addTextFieldWithConfigurationHandler()` method. In our example we need two, one for the email and another for the password. The view controller of the initial view has to include an Action for the My Account button to create the Alert View with the Text Fields.

Listing 17-4: Creating and processing a login form

```
import UIKit
```

```
class ViewController: UIViewController {  
    @IBAction func loginUser(sender: UIBarButtonItem) {  
        let alert = UIAlertController(title: "Insert Email and Password", message: nil, preferredStyle: .Alert)  
        let cancel = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)  
        alert.addAction(cancel)  
        let action = UIAlertAction(title: "Login", style: .Default, handler: { (action) in  
            if let fields = alert.textFields {  
                let email = fields[0].text  
                let password = fields[1].text  
                if email == "test@yahoo.com" && password == "12345" {  
                    self.performSegueWithIdentifier("showMainScreen", sender: self)  
                }  
            }  
        })  
        alert.addAction(action)  
        present(alert, animated: true, completion: nil)  
    }  
}
```

```

    }
    }
    })
    alert.addAction(action)
    alert.addTextFieldWithConfigurationHandler({ (textField) in
        textField.placeholder = "Email"
    })
    alert.addTextFieldWithConfigurationHandler({ (textField) in
        textField.placeholder = "Password"
        textField.secureTextEntry = true
    })
    presentViewController(alert, animated: true, completion: nil)
}
}

```

The `addTextFieldWithConfigurationHandler()` method includes an attribute to provide a closure where we can configure the Text Field. For the email we only need to set a placeholder, but the Text Field for the password has to be declared as secure with the `secureTextEntry` property (see Chapter 7). The Alert View includes two buttons: Cancel and Login. When the user presses the Login button we have to compare the values inserted in the fields with the data we have in our model (usually a database in the device or in an external server). To access the Text Fields from the closure in the action for the Login button we read the `textFields` property of the Alert View. This property contains an array with references to the Text Fields in the view. The indexes of the array follow the order of the Text Fields in the code (the first one is at index 0, the second at index 1, etc.). For this example we keep it simple and compare the values of each Text Field with hard-coded strings. If the values coincide, we trigger the Segue to open the second view where the user can begin working with our app (the Segue is a Show Segue identified with the string “showMainScreen”).



Figure 17-4: Logging in

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add a bar button with the title My Account to the initial view. Add a second view to the Storyboard. Add a label with the text “You are logged in” to the second view. Connect the initial view to the second view with a Show Segue. Connect the button in the initial view with an Action called `loginUser()`. Complete the Action with the code of Listing 17-4. Run the application and press the My Account button. Insert the email “test@yahoo.com” and the password “12345”. Press the Login button in the Alert View. You should see screens similar to Figure 17-4.

Action Sheets

Alert Views of type Action Sheet are usually presented when our application needs the user to make a decision and there is more than one option available. As shown in the following example, the creation of an Action Sheet is similar to Alert Views, but we have to use the `ActionSheet` style for the initializer. The example assumes that the interface contains a button connected to the `loginUser()` Action in the `ViewController` class.

Listing 17-5: Creating an Action Sheet

```
import UIKit

class ViewController: UIViewController {
    @IBAction func openSheet(sender: UIButton) {
        let alert = UIAlertController(title: “Emails”, message: “What do you want to do with the message?”,
        preferredStyle: .ActionSheet)

        let action1 = UIAlertAction(title: “Move to Inbox”, style: .Default, handler: nil)
        alert.addAction(action1)

        let action2 = UIAlertAction(title: “Delete”, style: .Destructive, handler: nil)
        alert.addAction(action2)

        let cancel = UIAlertAction(title: “Cancel”, style: .Cancel, handler: nil)
        alert.addAction(cancel)

        presentViewController(alert, animated: true, completion: nil)
    }
}
```

The code of Listing 17-5 creates an Alert View of type Action Sheet when the user presses the button in the interface. This Action Sheet contains a total of three buttons: a **Default** button called Move to Inbox, a **Destructive** button called Delete, and a **Cancel** button called Cancel. We didn’t include any handler for the buttons, so their only purpose is to dismiss the view, but the example illustrates how an Action Sheet is created and how the

buttons are organized inside the view. No matter the position of the buttons in the code, the **Cancel** button is always displayed at the end, in a separate view, and the **Destructive** button is highlighted in color red.

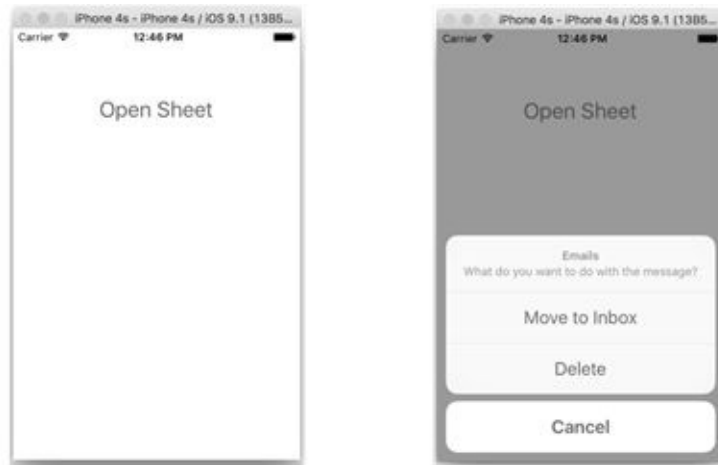


Figure 17-5: Action Sheet in action

Do It Yourself: Create a new Single View Application project. Add a button called Open Sheet to the initial view. Connect the button to the **ViewController** with an Action called `openSheet()`. Complete the Action with the code of Listing 17-5. Run the application. You should see something similar to Figure 17-5.

As we already mentioned, Action Sheets are presented as modal views at the bottom of the screen in small devices, but as popovers in iPads. The problem is that the system can't display a popover without the necessary information. If we execute the previous application on an iPad we get an error and the application crashes. To make sure that the code works in every device we have to change the configuration of the presentation. For this purpose, the **UIAlertController** object includes the **popoverPresentationController** property with a reference to the **UIPopoverPresentationController** object that is in charge of the presentation when the presentation style is **Popover**. The values we need to provide to this object are the view to which the Action Sheet is anchored, the rectangle inside the view the popover's arrow points to, and the directions the arrow can take.

Listing 17-6: *Presenting the Action Sheet as a popover in iPads*

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var sheetButton: UIButton!
    @IBAction func openSheet(sender: UIButton) {
        let alert = UIAlertController(title: "Emails", message: "What do you want to do with the message?", preferredStyle:
        .ActionSheet)
```

```

if let popover = alert.popoverPresentationController {
    popover.sourceView = sheetButton
    popover.sourceRect = sheetButton.bounds
    popover.permittedArrowDirections = [.Up]
}
let action1 = UIAlertAction(title: "Move to Inbox", style: .Default, handler: nil)
alert.addAction(action1)
let cancel = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
alert.addAction(cancel)
let action2 = UIAlertAction(title: "Delete", style: .Destructive, handler: nil)
alert.addAction(action2)
presentViewController(alert, animated: true, completion: nil)
}
}

```

The view controller of Listing 17-6 adds the configuration for the popover to the previous example. We decided to anchor the popover to the button that opens the view, so we created an Outlet to reference the button called `sheetButton` and then assigned that reference to the `sourceView` property of the `UIPopoverPresentationController` object. This anchors the popover to the button, but we still have to determine to which part of the button's view the arrow is going to point to. Assigning the internal frame of the button's view to the `sourceRect` property we make sure that the arrow always points to one of the sides of the button and the button is not covered by the view (see Listing 16-22). Finally, we set the `permittedArrowDirections` property with only the value `Up`, so the view is positioned below the button, as shown in Figure 17-6.



Figure 17-6: Action Sheet on an iPad

The Action Sheet not only adapts its presentation to the space available but also its design. If we run the example of Listing 17-6 in an iPad, the Cancel button is automatically removed because the user can already dismiss the popover by tapping anywhere else on the screen.

Chapter 18

Notifications

18.1 Notification Center

Besides the techniques we have seen so far to transfer data between different parts of an application, such as sending values from one view controller to another or providing a common model from which every view controller can get the information it needs, we can also send notifications to report changes across the application. Foundation includes the `NSNotificationCenter` class from which the system creates an object that serves as a Notification Center for the whole application. We can send notifications to this object and then listen to those notifications from anywhere in the code. The class includes the following type method to get a reference to this object.

defaultCenter()—This type method returns the `NSNotificationCenter` object assigned to the application.

The Notification Center is like a bulletin board; we can post a notification and then read it from anywhere in the code. The readers are called *observers*. These are just objects we add to the Notification Center to tell it who wants to know when a particular notification arrives. For example, our model may post a notification every time some of its values change. We can set a view controller to perform a task every time the notification arrives. When the Notification Center receives the notification it checks what objects are interested in it and finds our view controller. The view controller is informed of the notification and calls a method to perform the task. The `NSNotificationCenter` class includes the following methods to post notifications and also add and remove observers.

postNotificationName(name, object: sender, userInfo: info)—This method posts a notification to the Notification Center. The **name** attribute is a string with the name of the notification, the **sender** attribute is a reference to the object that sent the notification, and the **info** attribute is a dictionary with the information we want to pass to the observer.

addObserver(observer, selector: action, name: name, object: sender)—This method adds an observer for a specific notification to the Notification Center. The **observer** attribute is a reference to the object that is going to listen to the notification, the **action** attribute is a string with the name of the method that is going to be executed when the notification is received, the **name** attribute is a string with the name of the notification, and the **sender** attribute is a reference to the object that is sending the notification.

removeObserver(observer, name: name, object: sender)—This method removes an observer for a specific notification from the Notification Center. The attributes are the same provided by the `addObserver()` method when the observer was

created.

Notifications are used for multiple purposes. We can post a notification after a long process is over to tell a view controller to update its interface, we can communicate distant view controllers in a large interface, or we can keep a view controller up to date posting notifications from the model. The following example includes a model that posts a notification every time a new value is inserted. The interface presented in Figure 18-1 includes two views to show the amount of values available and insert new ones.

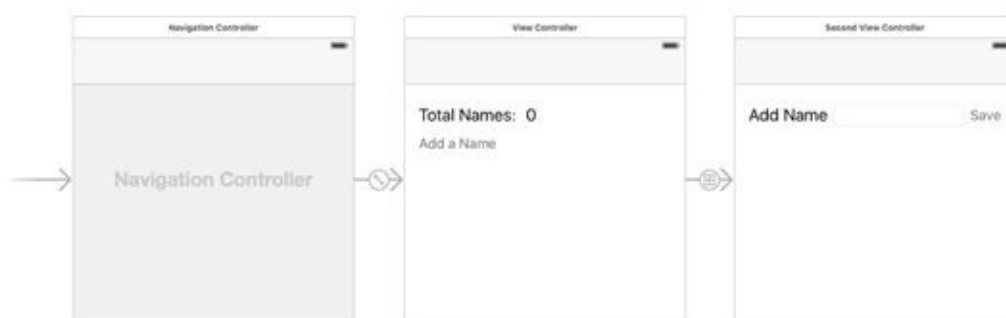


Figure 18-1: Interface to test notifications

The model has to store the value inserted by the user in the Text Field and then post a notification so the initial view can update its interface.

Listing 18-1: Sending notifications from the model

```
import Foundation

struct ApplicationData {
    var names: [String]

    mutating func addNewName(newName: String) {
        names.append(newName)
        let center = NSNotificationCenter.defaultCenter()
        center.postNotificationName("Update Data", object: nil, userInfo: nil)
    }
}

var AppData = ApplicationData(names: [])
```

The **ApplicationData** structure created for this example contains only one property called **names**, but we also included a method called **addNewName()** to assign new values to this property. After adding the new value to the **names** array, the method gets a reference to the **NSNotificationCenter** object assigned to our app and posts a notification called “Update Data”

to report the action. The view controller for the initial view has to add itself as an observer of this notification and update its interface every time the notification is received.

Listing 18-2: Adding an observer

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var counter: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        let center = NSNotificationCenter.defaultCenter()
        center.addObserver(self, selector: "updateCounter:", name: "Update Data", object: nil)
    }

    func updateCounter(notification: NSNotification) {
        let current = AppData.names
        counter.text = String(current.count)
    }
}
```

Observers are usually added as soon as the object is loaded and after the initialization is over. This is to avoid conflicts between the values managed by the object and those produced when the notification is received. Again, we have to get a reference to the **NSNotificationCenter** object assigned to our app first and then add the view controller as the observer of the “Update Data” notification posted by the model. The first attribute for the **addObserver()** method is the reference to the observer (in this case the **ViewController** object), and the second attribute is the method we are going to execute when the notification is received. The method, which we called **updateCounter()**, gets the amount of elements in the **names** property of the model and updates the label with this value.

The values in the model are added from the second view. The view controller for this view has to take the value inserted in the Text Field and call the **addNewName()** method to add it to the model.

Listing 18-3: Adding new values to the model

```
import UIKit

class SecondViewController: UIViewController {
    @IBOutlet weak var name: UITextField!

    @IBAction func saveName(sender: UIButton) {
        if let value = name.text {
            AppData.addNewName(value)
        }
    }
}
```

```

    }
    name.text = ""
}
}

```

When we insert a new name into the Text Field and press the Save button, the `saveName()` method adds the value to the model with the `addNewName()` method. After this, the model receives the value, stores it in the `names` array, and posts the “Update Data” notification. The Notification Center informs the `ViewController` object that an “Update Data” notification has arrived and the object executes the `updateCounter()` method. This method counts the values in the `names` array and shows the result on the screen.

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add a second view. Add two labels and a button to the main view, and a label, a Text Field, and a button to the second view, as shown in Figure 18-1. Connect the button in the initial view to the second view with a Show Segue. Create a subclass of `UIViewController` called `SecondViewController` and assign it to the second view. Complete the `ViewController` class with the code of Listing 18-2, and the `SecondViewController` class with the code of Listing 18-3. Connect the elements to their respective Outlets and Actions. Run the application, press the button to go to the second view, and insert a few names. Move back to the initial view. The view should show the number of names inserted.

The `postNotificationName()` method creates an object of the `NSNotification` class to hold the information for the notification. The class includes the following properties to store the values.

name—This property returns the name of the notification.

object—This property returns a reference to the object that posted the notification.

userInfo—This property returns the dictionary assigned to the notification.

On the other side of the communication, the `addObserver()` method sends to the action a reference of the `NSNotification` object that represents the notification. From this reference we can read the notification’s properties. The most useful is `userInfo`, because it allows us to send information from one object to another. The values allowed for this dictionary are called Property List values (values allowed in a plist file). This includes the classes `NSNumber`, `NSString`, `NSDate`, `NSArray`, `NSDictionary`, `NSData`, and also their equivalents in Swift. Swift values of type `Int`, `Float`, `Double`, or `Bool` are converted into `NSNumber` objects, `String` values are converted into `NSString` objects, and Swift arrays and dictionaries are converted

into `NSArray` and `NSDictionary` objects, respectively. There are multiple applications for this property. Considering our previous example, we may include a string to describe the value that was updated in the model.

Listing 18-4: Sending information along with the notification

```
import Foundation

struct ApplicationData {
    var names: [String]

    mutating func addNewName(newName: String) {
        names.append(newName)
        let info = ["type": "New Name"]
        let center = NotificationCenter.defaultCenter()
        center.postNotificationName("Update Data", object: nil, userInfo: info)
    }
}

var AppData = ApplicationData(names: [])
```

The code of Listing 18-4 declares a dictionary with the key “type” and the value “New Name” and assigns it to the `userInfo` attribute of the `postNotificationName()` method. In the `ViewController` class we can check this value and decide whether or not to update the label.

Listing 18-5: Reading the value received by the observer

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var counter: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        let center = NotificationCenter.defaultCenter()
        center.addObserver(self, selector: "updateCounter:", name: "Update Data", object: nil)
    }

    func updateCounter(notification: NSNotification) {
        if let info = notification.userInfo {
        let type = info["type"] as? String
        if type == "New Name" {
            let current = AppData.names
            counter.text = String(current.count)
        }
    }
```

```
}  
}
```

The value of the `userInfo` property is an optional, so we have to unwrap it first and then read its content. The dictionary we get from this property is of type `[NSObject : AnyObject]`. The `NSObject` type is the basic type for all the objects created in Objective-C, which means we can assign any of the Foundation classes as the key, such as `NSString` or `NSNumber` (or their equivalents in Swift), and the `AnyObject` type represents any kind of object, including custom ones. Because the value of the dictionary is retrieved as an `AnyObject` type, we have to cast it to a `String` type before comparing it to the string “New Name”. If the values match, we modify the label.

Do It Yourself: Update the `ApplicationData` structure with the code of Listing 18-4 and the `ViewController` class with the code of Listing 18-5. Run the application. It should work like before. Change the string “New Name” in Listing 18-5 by a different string. Now the label is not updated anymore.

If we don’t want the object to respond to new notifications or the application doesn’t need it anymore, we have to remove the observer from the Notification Center. The `NSNotificationCenter` class provides the `removeObserver()` method for this purpose. For example, we can remove the observer in the `ViewController` class of our example so the label is only updated after the first notification arrives.

Listing 18-6: Removing the observer

```
import UIKit  
  
class ViewController: UIViewController {  
    @IBOutlet weak var counter: UILabel!  
    var center: NSNotificationCenter!  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        center = NSNotificationCenter.defaultCenter()  
        center.addObserver(self, selector: "updateCounter:", name: "Update Data", object: nil)  
    }  
    func updateCounter(notification: NSNotification) {  
        if let info = notification.userInfo {  
            let type = info["type"] as? String  
            if type == "New Name" {  
                let current = AppData.names  
                counter.text = String(current.count)  
            }  
        }  
    }  
}
```

```

    }
    }
    center.removeObserver(self, name: "Update Data", object: nil)
}
}

```

System Notifications

Besides the notifications our app posts, the system also posts notifications to the Notification Center all the time to report changes in the interface or on other objects running the application. At this time there are almost 200 notifications for the iOS system, including those defined in some of the classes we already studied. They work exactly like the custom notifications we studied before but are preprogramed in the classes, so the only thing we have to do is to respond to them adding observers. We will probably never use most of the notifications, but some are very useful. For example, the `UIWindow` class defines a few notifications to manage the keyboard.

UIKeyboardWillShowNotification—This notification is posted before the keyboard is shown on the screen.

UIKeyboardDidShowNotification—This notification is posted after the keyboard was shown on the screen.

UIKeyboardWillHideNotification—This notification is posted before the keyboard is hidden.

UIKeyboardDidHideNotification—This notification is posted after the keyboard was hidden.

UIKeyboardWillChangeFrameNotification—This notification is posted before the keyboard's frame changes (due to a device rotation).

UIKeyboardDidChangeFrameNotification—This notification is posted after the keyboard's frame changes (due to a device rotation).

These notifications not only report the state of the keyboard but they also provide the values of the keyboard's frame through the `userInfo` property, so we can calculate the space available to reposition the elements in the interface. The values in the `userInfo` dictionary are identified by constants. The most frequently used constants are `UIKeyboardFrameBeginUserInfoKey` (returns a value with the frame occupied by the keyboard when closed) and `UIKeyboardFrameEndUserInfoKey` (returns a value with the frame occupied by the keyboard when opened).

The following example introduces the interface we are going to use to learn how to work with system notifications. The main view contains a button and a Text View. When the user tries to edit the text, the keyboard appears and the Text View adapts its size to the remaining space, but when the button is pressed the keyboard is closed and the Text View goes back to its original size.

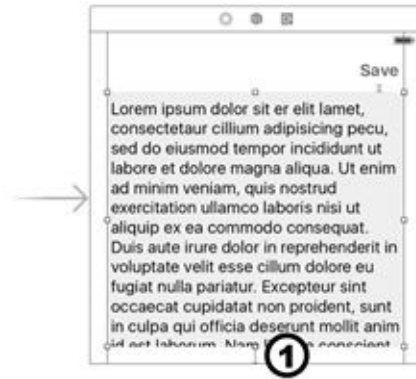


Figure 18-2: Interface to test keyboard notifications

There are different ways to change the size of an element and rearrange the interface. In this example we are going to change the size of the Text View by modifying the Space constraint that pins the bottom edge of the Text View to the bottom of the main view (Figure 18-2, number 1). Extending this constraint, we can change the size of the Text View to make it fit in the space left by the keyboard.

Listing 18-7: Observing keyboard’s notifications

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var heightBottom: NSLayoutConstraint!
    @IBOutlet weak var mainText: UITextView!
    var constraintHeight: CGFloat!

    override func viewDidLoad() {
        super.viewDidLoad()
        constraintHeight = heightBottom.constant
        let center = NotificationCenter.defaultCenter()
        center.addObserver(self, selector: "openKeyboard:", name: UIKeyboardWillShowNotification, object: nil)
        center.addObserver(self, selector: "closeKeyboard:", name: UIKeyboardWillHideNotification, object: nil)
    }

    func openKeyboard(notification: NSNotification) {
        if let info = notification.userInfo {
            let value = info[UIKeyboardFrameEndUserInfoKey] as! NSValue
            let frame = value.CGRectValue()

```

```

    let newHeight = frame.size.height + constraintHeight
    heightBottom.constant = newHeight
}
}
func closeKeyboard(notification: NSNotification) {
    heightBottom.constant = constraintHeight
}
@IBAction func saveText(sender: UIButton) {
    mainText.resignFirstResponder()
}
}

```

In the **ViewController** class of Listing 18-7 we have connected the constraint at the bottom of the Text View to an Outlet called **heightBottom** and defined a property called **constraintHeight** to store the height of the constraint when the keyboard is closed. The **viewDidLoad()** method reads the current height of the constraint, assigns it to the **constraintHeight** property and then adds two observers to the Notification Center for the **UIKeyboardWillShowNotification** and **UIKeyboardWillHideNotification** notifications. When the keyboard is opened or closed, the **UIWindow** object posts these notifications and the observers execute the **openKeyboard()** and **closeKeyboard()** methods, respectively. The **openKeyboard()** method reads the **userInfo** property, gets the frame of the keyboard with the **UIKeyboardFrameEndUserInfoKey** key, and calculates the new size of the constraint by adding the height of the keyboard to the normal height of the constraint (the one defined in the Storyboard).

Notice that the value returned by the **UIKeyboardFrameEndUserInfoKey** key is of type **NSValue**. This is a basic class in Foundation that can store values of different type, including all the structures from the Core Graphics framework. The only problem is that the class was designed to store objects, not structures, so it provides some methods to convert the value into the structure we want. The following are the most frequently used.

CGRectValue()—This method creates and returns a **CGRect** value.

CGSizeValue()—This method creates and returns a **CGSize** value.

CGPointValue()—This method creates and returns a **CGPoint** value.

CGVectorValue()—This method creates and returns a **CGVector** value.

Do It Yourself: Create a new Single View Application project. Add a Save button and a Text View to the initial view. Pin both elements with Space constraints to the sides of the main view and between each other. Connect the Text View to the **ViewController** class with an Outlet called **mainText** and the Space constraint at the

bottom of the Text View with an Outlet called `heightBottom` (Figure 18-2, number 1). Connect the Save button to an Action called `saveText` and complete the `ViewController` class with the code of Listing 18-7. Run the application and tap on the Text View. The keyboard should emerge and the Text View should adapt to the remaining space.

IMPORTANT: There are also useful notifications available for some of the classes we have already studied and others we will learn about in further chapters. For example, the `UITextView` class includes notifications like `UITextViewTextDidChangeNotification` to let the observer know that the content of the Text View changed. For more information, visit our website and follow the links for this chapter.

18.2 Local Notifications

A different type of notification, that has nothing to do with those posted to the Notification Center, are Local Notifications. These are notifications shown to the users by the system when the app has an event to report, such as the completion of a task or a real-life event that users want the app to remind them of.

There are three different types of notifications: alert, icon badge, and sound. They can be scheduled all at once or in separate notifications. For instance, we can schedule a notification that displays an alert and plays a sound, another that displays a badge on the app's icon and plays a sound, or another one that just plays a sound to warn the user that something happened (e.g., an alarm clock). The alerts may be displayed as a banner, an Alert View, or a message on the lock screen, depending on the current state of the device and the configuration set by the user.

UIKit includes the `UILocalNotification` class to create a notification. The following are some of the properties included by the class to configure the notification.

alertBody—This property sets or returns the text of the notification.

alertAction—This property sets or returns the title for the right button when the notification is presented as an Alert View.

applicationIconBadgeNumber—This property sets or returns the number to show in the icon's badge. The value by default is 0, which means no badge is shown on the icon.

soundName—This property sets or returns the name of the file that contains the sound to play with the notification. The system provides a constant called `UILocalNotificationDefaultSoundName` to assign the sound by default to the notification. The value by default is `nil`, which means no sound will be played.

fireDate—This property sets or returns the date and time when the notification is going to be delivered by the system. It is a value of type `NSDate`.

timeZone—This property sets or returns the time zone used to schedule the notification. If we don't specify a value, the system uses the GMT zone, which is usually not where the user is located. It is a value of type `NSTimeZone`.

repeatInterval—This property sets or returns a value that determines the interval to reschedule the notification. It is a structure of type `NSCalendarUnit`, but only the following properties are considered: **Year**, **Month**, **Day**, **Hour**, and **Minute**. The value by default is 0, which means the notification is only delivered once.

userInfo—This property sets or returns a dictionary with Property List values to pass custom data to the application.

Before starting the scheduling of notifications we have to inform to the system the type of notifications our app wants to deliver. UIKit offers the `UIUserNotificationSettings` class for this purpose. The class includes the following initializer.

`UIUserNotificationSettings(forTypes: types, categories: categories)`—This initializer creates a `UIUserNotificationSettings` object that establishes the types of notifications our app wants to deliver. The **`types`** attribute is a set of properties defined in the `UIUserNotificationType` structure. The properties available are `None`, `Badge`, `Sound`, and `Alert`. The **`categories`** attribute is an optional set of `UIUserNotificationCategory` objects that define custom actions for the buttons of a notification presented as an Alert View.

The `UIApplication` class includes the following properties and methods to get a list of the notifications already scheduled, schedule new ones, cancel, and register the settings on the system.

`scheduledLocalNotifications`—This property contains an array with all the local notifications already scheduled in the system.

`registerUserNotificationSettings(settings)`—This method registers the settings for our app. The **`settings`** attribute is an object of the `UIUserNotificationSettings` class.

`scheduleLocalNotification(notification)`—This method schedule a local notification. The **`notification`** attribute is an object of the `UILocalNotification` class that represents the notification.

`cancelLocalNotification(notification)`—This method cancels the notification indicated by the **`notification`** attribute. The attribute is a reference to the `UILocalNotification` object that represents the notification.

`cancelAllLocalNotifications()`—This method cancels all the local notifications already scheduled.

We can set the type of notifications we want to use from anywhere in our code, but we have to consider that the system asks the user for permission to show the notifications the first time the app is executed, so it is better to do it as soon as possible. A common place is the app's delegate, as shown in the following example.

Listing 18-8: Asking permission to the user

```
import UIKit
```

```

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let app = UIApplication.sharedApplication()
        let setNotifications = UIUserNotificationSettings(forTypes: [.Alert, .Sound, .Badge], categories: nil)
        app.registerUserNotificationSettings(setNotifications)
        return true
    }
}

```

The `registerUserNotificationSettings()` method is provided by the `UIApplication` class, so we have to get a reference to the `UIApplication` object created for our app first and then call the method to register the settings we want. For this example we decided to deliver notifications with alerts, sounds, and badges (`[.Alert, .Sound, .Badge]`). When this code is executed for the first time, the system shows an Alert View asking the user for permission. If permission is granted, we can start posting notifications. The interface of Figure 18-3 was designed for this purpose. We included a Text Field to introduce the message that the notification is going to show and a button to schedule the notification.



Figure 18-3: Interface to schedule notifications

When the button is pressed, we have to get the text inserted in the Text Field and use it to configure the notification. We also have to set the time the notification is going to be shown and the time zone. The following example shows a notification 30 seconds after it was created.

Listing 18-9: Scheduling a notification

```

import UIKit

class ViewController: UIViewController {

```

```
@IBOutlet weak var messageField: UITextField!
```

```
@IBAction func saveNotification(sender: UIButton) {  
    let message = messageField.text  
    let date = NSDate(timeIntervalSinceNow: 30)  
  
    let notification = UILocalNotification()  
    notification.fireDate = date  
    notification.timeZone = NSTimeZone.defaultTimeZone()  
    notification.alertBody = message  
  
    let app = UIApplication.sharedApplication()  
    app.scheduleLocalNotification(notification)  
    messageField.text = ""  
}  
}
```

The process to schedule a notification is simple. We have to create an instance of the `UILocalNotification` class, configure its properties and then call the `scheduleLocalNotification()` method of the `UIApplication` object to add the notification to the list. We need to define at least three properties for the notification to be delivered: the `fireDate` property to set the date when the notification is going to be shown, the `timeZone` property to set the time zone that has to be considered to calculate the delivery date, and any of the properties required for the type of notification we want to show. In this example we decided to show an alert and therefore we had to set the `alertBody` property with the message we want the user to see, but we could have also declared the `soundName` property to play a sound (e.g., `notification.soundName = UILocalNotificationDefaultSoundName`) or the `applicationIconBadgeNumber` to assign a badge to the app's icon (e.g., `notification.applicationIconBadgeNumber = 1`).

Alert notifications adopt different configurations depending on the state of the device. If the screen is locked, for example, the notification is shown as a message at the center of the screen, but when the device is displaying the home screen, the notification is shown as a banner at the top. If the user sets the app to show an Alert View instead of a message, the system displays the Alert View with buttons to close the app or open the application.

Do It Yourself: Create a new Single View Application project. Add a label, a Text Field and a button to the initial view, as in the interface of Figure 18-3. Connect the Text Field to the `ViewController` class with an Outlet called `messageField` and the button with an Action called `saveNotification()`. Complete the `ViewController` class with the code of Listing 18-9. Update the `AppDelegate` class with the code of Listing 18-8. The first time you run the application, the system will ask you to allow the app to deliver notifications. Press the OK button. Insert a text in the field and press the Save Notification button. Open the Hardware menu at the top of the screen and select the

Home or the Lock options. After a few seconds you should see the notification popping up on the screen, as shown in Figure 18-4.

If the user is working with the app when a notification is delivered, the notification is not displayed on the screen but the system calls the `application(application, didReceiveLocalNotification: notification)` method of the `UIApplicationDelegate` protocol on the app's delegate to let the app know that the notification was effectively delivered. We can implement this method in the `AppDelegate` class to perform any task we want. In the following example, we just print a message on the console to let us know that the method was executed, but we could have done something more meaningful, like updating the model or the interface.

Listing 18-10: Responding to notifications when the app is in the foreground

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let app = UIApplication.sharedApplication()
        let setNotifications = UIUserNotificationSettings(forTypes: [.Alert, .Sound, .Badge], categories: nil)
        app.registerUserNotificationSettings(setNotifications)
        return true
    }

    func application(application: UIApplication, didReceiveLocalNotification notification: UILocalNotification) {
        print("An event has occurred")
    }
}
```

Do It Yourself: Update the code in the `AppDelegate` class with the code of Listing 18-10. Run the application, insert a message and press the Save Notification button. Wait for 30 seconds. You should see the message “An event has occurred” printed on the Xcode’s console.

If the device is locked, the user can slide the notification to open the app. The same happens if the user taps on the banner in the Home screen or presses the Open button in the Alert View (Figure 18-4). When the system opens the application from a notification, it calls the `application(application, didFinishLaunchingWithOptions: options)` method of the `UIApplicationDelegate` protocol on the app's delegate and assigns a dictionary to the `options`

attribute with an item that contains the constant `UIApplicationLaunchOptionsLocalNotificationKey` as key and the `UILocalNotification` object that represents the notification as value. Reading this value we are able to know when the app is opened from a notification and which one it was. For example, we can assign some values to the `userInfo` property when the notification is scheduled and then read these values from the `AppDelegate` object to know which notification opened the app. The following example assumes that we have assigned the dictionary `[“type”: “Event”]` to the `userInfo` property of the `UILocalNotification` object in the `ViewController` class.

Listing 18-11: Reading values from the notification’s `userInfo` property

```
import UIKit
```

```
@UIApplicationMain
```

```
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
    var window: UIWindow?
```

```
    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
```

```
        let app = UIApplication.sharedApplication()
```

```
        let setNotifications = UIUserNotificationSettings(forTypes: [.Alert, .Sound, .Badge], categories: nil)
```

```
        app.registerUserNotificationSettings(setNotifications)
```

```
        if let options = launchOptions {
```

```
            if let notification = options[UIApplicationLaunchOptionsLocalNotificationKey] as? UILocalNotification {
```

```
                if let info = notification.userInfo {
```

```
                    let type = info[“type”] as! String
```

```
                    if type == “Event” {
```

```
                        print(“An event has occurred”)
                    }
                }
            }
        }
```

```
    }
```

```
    }
```

```
    }
```

```
    }
```

```
    return true
```

```
    func application(application: UIApplication, didReceiveLocalNotification notification: UILocalNotification) {
        print(“An event has occurred”)
    }
}
```

There are some values we have to unwrap before we can read the `userInfo` property. The `options` attribute is an optional, so this is the first one. If we get a value, then we have to get the item with the `UIApplicationLaunchOptionsLocalNotificationKey` key. If this returns a value, we finally unwrap the `userInfo` property and compare the value of its “type” key against the

string “Event”. If they match, that means that the application was opened from one of the notifications scheduled from our code.

Do It Yourself: Assign the dictionary [“type”: “Event”] to the `userInfo` property of the `UILocalNotification` object created in the `ViewController` class (`notification.userInfo = [“type”: “Event”]`). Update the `AppDelegate` class with the code of Listing 18-11. Run the application, schedule a new notification, and wait on the Home screen. When the notification shows up, tap on it to open the app. You should see the message “An event has occurred” on the Xcode’s console.

There is a limit of 64 local notifications per app, but we can create notifications that are automatically rescheduled and they count as one. These types of notifications are defined by the `repeatInterval` property. The following example modifies the notification schedule by our app to reschedule every minute. The value of `fireDate` now becomes the starting point of the cycle.

Listing 18-12: Rescheduling a notification

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var messageField: UITextField!
    @IBAction func saveNotification(sender: UIButton) {
        let message = messageField.text
        let date = NSDate(timeIntervalSinceNow: 30)

        let notification = UILocalNotification()
        notification.fireDate = date
        notification.timeZone = NSTimeZone.defaultTimeZone()
        notification.alertBody = message
        notification.repeatInterval = .Minute
        let app = UIApplication.sharedApplication()
        app.scheduleLocalNotification(notification)
        messageField.text = ""
    }
}
```

If we need to cancel a notification before it is delivered or stop a rescheduling cycle, we can call any of the `UIApplication` methods for this purpose. For instance, we can read the values of the `scheduledLocalNotifications` property with a `for in` loop and cancel only a specific notification with the `cancelLocalNotification()` method, or we can call the

`cancelAllLocalNotifications()` method and cancel all the notifications at once, as in the following example.

Listing 18-13: Cancelling notifications

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var messageField: UITextField!

    @IBAction func saveNotification(sender: UIButton) {
        let app = UIApplication.sharedApplication()
        app.cancelAllLocalNotifications()
        let message = messageField.text
        let date = NSDate(timeIntervalSinceNow: 30)
        let notification = UILocalNotification()
        notification.fireDate = date
        notification.timeZone = NSTimeZone.defaultTimeZone()
        notification.alertBody = message
        notification.repeatInterval = .Minute
        app.scheduleLocalNotification(notification)
        messageField.text = ""
    }
}
```

In Listing 18-13 we cancel all the notifications every time a new one is scheduled. As a result, only the last notification inserted remains.

IMPORTANT: Besides the local notifications scheduled by our app, we can schedule remote notifications (also called *Push Notifications*). These types of notifications are delivered from our server to the Apple Push Notification Service and from there to the user's device. An example is the Gmail app. Every time an email arrives we get a notification on our device. The topic is beyond the scope of this book. For more information, visit our website and follow the links for this chapter.

18.3 Key/Value Observing

KVO (Key/Value Observing) is a system we can use to perform a task when the value of a property is modified. This is similar to what the `didSet()` method does for the properties of our own structures and classes (see Property Observers in Chapter 3), with the difference that some predefined classes provided by Apple come ready to work with this system and therefore we can monitor the values of their properties as we do with our own.

There are several situations in which KVO can be useful. For example, we may have a property that reflects the status of an operation and is required by other parts of the application to know when the process is over. The problem is that we don't know when that is going to happen and which are going to be the objects that have to be notified when a change occurs. Constantly checking the property from every object to see if its value has changed is not practical. KVO allows us to add observers in every object that will keep an eye on the property. Every time the property is modified, the system notifies these observers so the objects can perform the task they need, such as updating the interface or initiating another operation.

KVC

For the KVO system to be able to monitor the values of a property, the property's object must comply with the `NSKeyValueCoding` protocol. This protocol defines another system called KVC (Key/Value Coding) that allows us to access properties using their names as keys. Instead of using dot notation to access the values of a property we call methods defined in this protocol with strings that identify the property we want to modify. The following are the most frequently used.

setValue(value, forKey: key)—This method assigns a value to the property specified by the **key** attribute. The **value** attribute is the value we want to assign to the property and the **key** attribute is a string with the name of the property.

valueForKey(key)—This method retrieves the value of the property identified by the **key** attribute. The **key** attribute is a string with the name of the property.

Some of the frameworks provided by Apple define classes that are ready to work with KVC, but we can define our own. An easy way to make our custom classes KVC compliant is to declare them as subclasses of the `NSObject` class. This is the class from which the rest of the Objective-C classes are created and it already includes an implementation of the protocol. For example, the following code defines a subclass of

`NSObject` called `MyControl`.

Listing 18-14: Defining a custom class to work with KVC

```
import Foundation

class MyControl: NSObject {
    var round = false
}
```

Because the `MyControl` class is a subclass of `NSObject` we can call the `NSKeyValueCoding` protocol methods on instances of this class to set or retrieve the value of the `round` property. In the following example, we present a `ViewController` class that creates an instance of `MyControl` and modifies the value of the `round` property every time the user modifies the value of a Stepper. The example assumes that the initial view contains an interface with a Stepper and a label, as the one introduced in Figure 5-28.

Listing 18-15: Modifying property values using KVC

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var counterLabel: UILabel!
    var control: MyControl!

    override func viewDidLoad() {
        super.viewDidLoad()
        control = MyControl()
    }

    @IBAction func updateValue(sender: UIStepper) {
        let current = sender.value
        if current % 10 == 0 {
            control.setValue(true, forKey: "round")
        } else {
            control.setValue(false, forKey: "round")
        }
        counterLabel.text = String(current)
    }
}
```

The `ViewController` class of Listing 18-15 creates an instance of the `MyControl` class and then implements an Action for the Stepper to change the value of its `round` property every

time the value of the `Stepper` is modified. The Action gets the `Stepper`'s current value and then sets the value of the `round` property depending on whether the current value is a multiple of 10 or not (`true` or `false`).

Do It Yourself: Create a new Single View Application project. Add a `Stepper` and a label to the main view. Connect the `Stepper` with an Action called `updateValue()` and the label with an Outlet called `counterLabel`. Create a new Swift file called `MyControl` and copy the code of Listing 18-14 inside. Update the `ViewController` class with the code of Listing 18-15.

The Basics: KVC presents the alternative of referencing a property with a string instead of the traditional = sign or dot notation (e.g., `control.round = true`). There are circumstances in which this can be useful, like when we have to store the values of a series of properties in a dictionary. We can set the key of every item as the name of the property and then use the KVC methods to set or retrieve the value, for example. As we will see in further chapters, other frameworks like Core Data use this system extensively.

KVO

When properties are set using the `setValue()` method instead of the = sign, they automatically become KVO compliant and therefore notifications are posted to report when their values change. As with other notifications, these notifications can be read from any object that has registered an observer for them. Foundation includes the `NSKeyValueObserving` protocol, which is adopted by view controller classes, to define methods to add and remove observers. The following are the most frequently used.

addObserver(observer, forKeyPath: path, options: options, context: data)—This method adds an observer to the object. The **observer** attribute is the object that responds to the notification, the **path** attribute is a string with the name or the path to the property, the **options** attribute is a set with enumeration values of type `NSKeyValueObservingOptions` to set the values that are going to be sent to the method that responds to the notification (possible values are `New`, `Old`, `Initial`, and `Prior`), and the **data** attribute is a generic value that identifies the observer (used when a class and its subclasses observe the same property).

removeObserver(observer, forKeyPath: path, context: data)—This method removes an observer from the object. The attributes are the same values specified in the `addObserver()` method when the observer was added.

With Key/Value Observing, an object can observe the property of another object. When the value of the property changes, the method that responds to the notification is executed. This is not a custom method but a predefined method called `observeValueForKeyPath()`. Every observer added to an object calls this same method, as shown in the following example.

Listing 18-16: Observing the value of a property

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var counterLabel: UILabel!
    var control: MyControl!

    override func viewDidLoad() {
        super.viewDidLoad()
        control = MyControl()
        control.addObserver(self, forKeyPath: "round", options: [], context: nil)
    }

    @IBAction func updateValue(sender: UIStepper) {
        let current = sender.value
        if current % 10 == 0 {
            control.setValue(true, forKey: "round")
        } else {
            control.setValue(false, forKey: "round")
        }
        counterLabel.text = String(current)
    }

    override func observeValueForKeyPath(keyPath: String?, ofObject object: AnyObject?, change: [String : AnyObject]?, context: UnsafeMutablePointer<Void>) {
        if keyPath == "round" {
            if control.round {
                counterLabel.textColor = UIColor.redColor()
            } else {
                counterLabel.textColor = UIColor.blackColor()
            }
        }
    }
}
```

In Listing 18-16 we update the `ViewController` class of the previous example to add an observer to monitor the value of the `round` property. When the user modifies the value of

the `Stepper`, the `updateValue()` method is called and the value of the `round` property is modified with the `setValue()` method. This makes the property post a notification to report the change. The observer added to the `ViewController` class receives this notification and executes the `observeValueForKeyPath()` method. In this method we check if the name or path of the property matches the name or path of the property we want to read and then perform the task according to its value. In this example, if the value of `round` is `true` we change the color of the label to red, and if it is `false` we change it back to black.

Do It Yourself: Update the `ViewController` class with the code of Listing 18-16 and run the application. Press the `Stepper` to change the value. The `counterLabel` label should turn red every time the value is a multiple of 10.

Setting the value of the property with the `setValue()` method makes the property post a notification, but this means that every time we want the changes in a property to be noticed by the observers we have to perform those changes calling the `setValue()` method. If we want every change to be reported, including those performed with traditional methods like the `=` sign, we have to make the property compliant with KVO by implementing some of the methods provided by the `NSKeyValueObserving` protocol. The simplest way to do this is by calling the following protocol methods from the property's `didSet()` and `willSet()` methods.

`willChangeValueForKey(key)`—This method informs the class that the value of the property is about to change. The `key` attribute is a string with the property's name or path.

`didChangeValueForKey(key)`—This method informs the class that the value of the property was changed. The `key` attribute is a string with the property's name or path.

The following example implements these two methods to make the `round` property report the changes before and after they happen.

Listing 18-17: Reporting changes to the class

```
import Foundation

class MyControl: NSObject {
    var round = false {
        didSet {
            self.didChangeValueForKey("round")
        }
        willSet {
            self.willChangeValueForKey("round")
        }
    }
}
```

```
}  
}  
}
```

Now we can assign new values to the `round` property as we always do and the property will post a notification every time. The following example shows the new action for the Stepper.

Listing 18-18: Working with a KVO compliant property

```
@IBAction func updateValue(sender: UIStepper) {  
    let current = sender.value  
    if current % 10 == 0 {  
        control.round = true  
    } else {  
        control.round = false  
    }  
    counterLabel.text = String(current)  
}
```

Do It Yourself: Update the `MyControl` class with the code of Listing 18-17 and the `updateValue()` method of the `ViewController` class with the code of Listing 18-18. Run the application. Again, you should see the label changing colors according to the Stepper’s current value.

IMPORTANT: With Key/Value Observing you can only observe the properties of one object from another. To observe properties of the same object you can use Property Observers (see Chapter 3). Usually you don’t need this technique to observe values in your own classes, but some frameworks include classes that require the use of Key/Value Observing, as we will see in Chapter 18.

Chapter 19

Error Handling

19.1 Errors

Simple frameworks and classes usually work as expected. If we call a method from those classes it completes its task without failure. If the method does fail, it returns a neutral value like 0 or `nil` to indicate that there was a problem. But this is not always enough. More complex frameworks can't guarantee success and may encounter multiple problems while trying to fulfill their purpose. For this reason, Swift introduces a systematic process to report and handle errors called *Error Handling*.

Throwing Errors

When a method reports an error, it is said that it *throws* the error. Several frameworks provided by Apple were already programmed to throw errors, as we will see in further chapters, but we can also do it from our own structures and classes. To throw an error we have to use the `throw` and `throws` keywords. The `throw` keyword is used to throw the error and the `throws` keyword is specified in the method's declaration to indicate to the rest of the code that the method can throw errors (methods that throw errors are called *throwing methods*).

Because a method can throw multiple errors, we also have to indicate the type of error found with values of an enumeration type. This is a custom enumeration that adopts the `ErrorType` protocol. For instance, let's consider the following example.

Listing 19-1: Getting an error inside a method

```
struct Stock {  
    var totalLamps = 5  
    mutating func sold(amount: Int) {  
        totalLamps = totalLamps - amount  
    }  
}  
  
var mystock = Stock()  
  
mystock.sold(8)  
print("Lamps in stock: \(mystock.totalLamps)") // "Lamps in stock: -3"
```

The code of Listing 19-1 defines a structure called `Stock` that manages the stock of lamps available at the store. The class includes the `totalLamps` property to store the number of lamps we still have available and the `sold()` method to process the lamps sold. The method updates the stock by subtracting the amount of lamps we have sold from the value of the

`totalLamps` property. If the amount of lamps sold is less than the amount of lamps in stock, everything is fine, but when we sell more lamps than we have, as in this example, there is clearly a problem we have to report. To throw an error, we have to declare the types of errors available for the method, declare the method as a throwing method adding the **throws** keyword in the declaration (between the attributes and the returning data types), detect the error, and throw it with the **throw** keyword.

Listing 19-2: Throwing errors

```
enum Errors: ErrorType {
    case OutOfStock
}

struct Stock {
    var totalLamps = 5
    mutating func sold(amount: Int) throws {
        if amount > totalLamps {
            throw Errors.OutOfStock
        } else {
            totalLamps = totalLamps - amount
        }
    }
}

var mystock = Stock()
```

In Listing 19-2 we declare an enumeration called **Error** that adopts the **ErrorType** protocol and includes a case called **OutOfStock**. Declaring `sold()` as a throwing method with the **throws** keyword, we can now throw the **OutOfStock** error every time we try to sell more lamps than we have. If the lamps sold are more than the number of lamps in stock, the method throws the error, otherwise the stock is updated.

Handling Errors

Now that we have a method that can throw errors, we have to handle the errors when the method is executed. Swift includes the **try** keyword and the **do catch** statements for this purpose. The **do catch** statements create two blocks of code. If the statements inside the **do** block return an error, the statements in the **catch** block are executed. To execute a method that throws errors we have to declare these statements and call the method inside the **do** statement with the **try** keyword in front of it, as shown in the following example.

Listing 19-3: Handling errors

```
enum Errors: ErrorType {
    case OutOfStock
}

struct Stock {
    var totalLamps = 5
    mutating func sold(amount: Int) throws {
        if amount > totalLamps {
            throw Errors.OutOfStock
        } else {
            totalLamps = totalLamps - amount
        }
    }
}

var mystock = Stock()

do {
    try mystock.sold(8)
} catch Errors.OutOfStock {
    print("We don't have enough lamps")
}
```

The code of Listing 19-3 expands the previous example to handle the error thrown by the `sold()` method. Because of the addition of the `try` keyword, the system tries to execute the `sold()` method in the `mystock` structure and check for errors. If the method returns the `OutOfStock` error, the statements inside the `catch` block are executed. This pattern allows us to respond every time there is an error and report it to the user or correct the situation without having to crash the app or produce unexpected results.

Do It Yourself: Create a new Playground file. Copy the code of Listing 19-3 inside the file. You should see the message “We don’t have enough lamps” printed in the Results Side Bar. Replace the number 8 with the number 3. Now the message should disappear because there are enough lamps in stock.

The Basics: You can add as many errors as you need to the `Errors` enumeration. The errors can be checked later with multiple `catch` statements in sequence. Also, you may add all the statements you need to the `do` block. The statements before `try` are always executed, while the statements after `try` are only executed if no error is found.

If we don’t care about the error returned, we can force the `try` keyword to return an

optional with the syntax `try?`. If the method throws an error, the instruction returns `nil`, and therefore we can avoid the use of the `do catch` statements.

Listing 19-4: Catching errors with `try?`

```
enum Errors: ErrorType {
    case OutOfStock
}

struct Stock {
    var totalLamps = 5
    mutating func sold(amount: Int) throws {
        if amount > totalLamps {
            throw Errors.OutOfStock
        } else {
            totalLamps = totalLamps - amount
        }
    }
}

var mystock = Stock()
try? mystock.sold(8)
```

The instruction of Listing 19-4 returns the value `nil` if the method throws an error, or an optional with the value returned by the method if everything goes right. Because in this example we are not processing the value returned, Xcode shows a warning (“Result of `try?` is unused”). To avoid this warning we can assign the value returned to an underscore, as in `_ = try? mystock.sold(8)`. Another alternative is to call the method in an `if` statement using optional binding. Again, if we don’t need the value returned we can assign the instruction to an underscore.

Listing 19-5: Checking errors with optional binding

```
if let _ = try? mystock.sold(8) {
    print(“We have enough stock”)
} else {
    print(“We need more lamps”)
}
```

Do It Yourself: Replace the code in your Playground file with the code of Listing 19-4. You should see the value `nil` in the Results Side Bar. Replace the last instruction of Listing 19-4 by the code of Listing 19-5. Now you should see the message “We need more lamps” instead.

Sometimes we know beforehand that a throwing method is not going to throw an error and therefore we want to avoid writing unnecessary code. In cases like this, we can use the syntax `try!`. For instance, the following code checks if there are enough lamps before calling the `sold()` method, so we know that the instruction will never throw the `OutOfStock` error.

Listing 19-6: Ignoring the errors

```
enum Errors: ErrorType {
    case OutOfStock
}

struct Stock {
    var totalLamps = 5
    mutating func sold(amount: Int) throws {
        if amount > totalLamps {
            throw Errors.OutOfStock
        } else {
            totalLamps = totalLamps - amount
        }
    }
}

var mystock = Stock()

if mystock.totalLamps > 3 {
    try! mystock.sold(3)
}

print("Lamps in stock: \(mystock.totalLamps)")
```

IMPORTANT: As well as with optionals, if the method throws an error and we declare the `try` keyword with an exclamation mark, the app crashes, so we have to be sure that the method is not going to throw an error before creating instructions with `try!`.

Chapter 20

Operation Queues

20.1 Queues

iOS can separate the code in operations. Operations are objects used to wrap a set of statements that perform a common task. These operations can then be put in a queue and executed in order, one after another, according to a level of priority assigned by the system or explicitly by the application. The advantage of this process is that the operations can be performed concurrently, which means they can take advantage of the hardware capacity to execute multiple instructions at the same time. For example, we may have a code that downloads a big file from the Internet and another code that shows the progress on the screen. In cases like this, we can't wait for one code to be over to execute the other; we need the two pieces of code to work at the same time. Using operations we can wrap the code to download the file in one operation and the code to show a progress bar in another, and then make them work simultaneously.

Operations

Foundation includes the `NSOperation` class to create operation objects. The class provides all the necessary functionality to configure an operation, but it must be subclassed to include the statements the operation is required to perform. To simplify the process, the framework defines a standard subclass of `NSOperation` called `NSBlockOperation`. This subclass can create an operation from the statements provided by a closure. The following is the class initializer.

`NSBlockOperation(block: closure)`—This initializer creates an operation with the statements specified in the closure.

Because the `NSBlockOperation` class is a subclass of `NSOperation`, it has access to its properties and methods. The following are the most useful.

`completionBlock`—This property sets the closure with the statements we want to execute after the operation is completed.

`queuePriority`—This property sets the operation's priority. It is an enumeration of type `NSOperationQueuePriority` with the values `VeryLow`, `Low`, `Normal`, `High`, and `VeryHigh`.

`cancel()`—This method asks the system to cancel the operation.

After the operations are created, we have to add them to a queue. For this purpose, Foundation includes the `NSOperationQueue` class. The class offers the following methods to create a queue and add operations to it.

addOperation(operation)—This method adds the operation specified by the operation attribute to the queue. The attribute is an object created from an `NSOperation` subclass.

addOperationWithBlock(closure)—This method creates an `NSOperation` object from the statements provided by the **closure** attribute and adds it to the queue. The attribute is a closure with the statements we want to include in the operation.

cancelAllOperations()—This cancels all the operations in the queue.

By default, iOS places the application’s code in a single queue called *Main Queue*. If we want some code to be executed in a different queue, we have to create an operation and add it to a new queue.

Listing 20-1: Adding operations to a queue

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let operation = NSBlockOperation(block: {
            var total: Double = 1
            for var f = 1; f < 100; f++ {
                total = log(total + Double(f))
            }
            print("Total: \(total)")
        })
        let queue = NSOperationQueue()
        queue.addOperation(operation)
        print("Printed in the Main Queue")
    }
}
```

The operation is created by the `NSBlockOperation()` initializer. This initializer takes a closure with the statements we want to execute (in this case, a loop to calculate the

logarithm of a sequence of values) and creates the operation. With the operation ready, we initialize a new `NSOperationQueue` object to represent the queue and add the operation to it.

When this code is executed, the system creates the operation, adds it to the new queue and runs it as soon as it can. At this point, the system is running two codes at the same time, the code in the main queue and the code in the new queue we just created. As a result, we will see the message “Printed in the Main Queue” in the console first, and then the message produced by the operation.

This code guarantees that the code in the main queue is executed, no matter how much time the statements in the operation take to finish the process. If we wouldn’t have created the operation and added it to a new queue, the system would have had to wait until the execution of the loop was over to execute the last statement and print its message in the console.

Do It Yourself: Create a new Single View Application project. Update the `ViewController` class of the initial view with the code of Listing 20-1 and run the application. The Xcode’s console should print the message “Printed in the Main Queue” first and then the operation’s message after the loop is over.

Main Queue

Some frameworks, like UIKit, can only work in the main queue. If we try to access the elements of the interface from a different queue, for example, the application will produce unexpected results. The `NSOperationQueue` class offers the following type method to get a reference to the main queue when we need to add operations to it.

mainQueue()—This type method returns a reference to the main queue.

Operations are not the only units of code that can be executed outside the main queue; also blocks of code (closures) may sometimes be executed in a different queue. Every time we have to execute code inside an operation or one of these blocks, we have to consider whether that code can be executed in any queue or it only works properly in the main queue. If the code requires it to be executed in the main queue, we have to get a reference to this queue calling the `mainQueue()` method and add an operation to it.

Listing 20-2: Adding operations to the main queue

```
import UIKit
```

```

class ViewController: UIViewController {
    @IBOutlet weak var totalLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        let operation = NSBlockOperation(block: {
            var total: Double = 1
            for var f = 1; f < 100; f++ {
                total = log(total + Double(f))
            }
        })

        let main = NSOperationQueue.mainQueue()
        main.addOperationWithBlock({
            self.totalLabel.text = "Total: \(total)"
        })

        let queue = NSOperationQueue()
        queue.addOperation(operation)
        print("Printed in the Main Queue")
    }
}

```

In Listing 20-2, instead of printing a message to the console we update a label with the result of the loop. The elements of the interface can only be modified from the main queue, so we get a reference to this queue and then modify the label from inside an operation added with the `addOperationWithBlock()` method. This method creates a new operation with the code inside the block and adds it to the queue, simplifying the process. At the end, we get a similar result to the previous application, the loop is executed along with the rest of the code, but when it is over, the operation that modifies the label is executed, showing the result on the screen.

Do It Yourself: Add a label to the initial view and connect it to the view controller with an Outlet called `totalLabel`. Update the `ViewController` class of the previous example with the code of Listing 20-2. Run the application. You should see the result produced by the loop on the screen.

The Basics: Computer processors are designed to execute multiple instructions at the same time. Operative systems take advantage of this capacity and split the code in different threads of execution. This allows the system to process different pieces of code simultaneously, improving efficiency. When we create an operation and assign it to a queue, the system designates a thread for that operation and runs it along with the rest of the code. We will see more detailed examples in further chapters.

Chapter 21

Storage

21.1 User Preferences

Up to this point, all the data for the app was stored in arrays and dictionaries created in the model. The values were hard-coded, which means they are always the same every time the app is launched. If we add a new value to the model, it is only preserved for the time the app is running, but as soon as the app is closed, we lose that value and everything is back to the initial state. The data stored this way is called *temporary data* and it is useful when our app requires values that never change or when a long process generates partial results that we have to access later to complete the task. Temporary data is always necessary for our apps, but users require persistent sources of information they can modify at will and preserve for as long as they want. To fulfill these demands, Apple includes several systems to store persistent data. They are all based on files, but can take several forms, including organized collections of data called Databases. The simplest system available in iOS is called *Users Defaults*. This system was designed to store user preferences, which may include values set by the user to determine how our app should work and also values set by the app itself to restore a previous state. These values are stored in a database managed by the system and therefore they continue to exist after the app is closed and as long as the user or the app decides to keep them.

NSUserDefaults

The Foundation framework defines a class called `NSUserDefaults` to manage the preferences of our application. Only one `NSUserDefaults` object is assigned per application, so we can reference this object anywhere in the code and always process the same values. The `NSUserDefaults` class offers the following type method to get a reference to the object created for the app.

`standardUserDefaults()`—This type method returns a reference to the app's `NSUserDefaults` object.

The values are assigned to the `NSUserDefaults` object with an associated key that we can use to retrieve them later, similar to how dictionaries work. In theory we can work with almost any type of value, including custom objects, but in practice they are restricted to the Property List values introduced in Chapter 18 (`NSNumber`, `NSString`, `NSDate`, `NSArray`, `NSDictionary`, `NSData`, and their equivalents in Swift). The `NSUserDefaults` class includes the following methods to store, retrieve, and delete values of any of these types.

setObject(value, forKey: key)—This method stores a new value with the key specified by the **key** attribute. If the value already exists, it is updated. The **value** attribute is a value of any of the types admitted by the `NSUserDefaults` class and the **key** attribute must be declared as a string.

objectForKey(key)—This method retrieves the value associated with the key specified by the **key** attribute. If the value doesn't exist, it returns `nil`. The attribute is the same string used to store the value.

removeObjectForKey(key)—This method removes the value associated with the key specified by the **key** attribute. The attribute is the same string used to store the value.

Because the methods above process values of any type, they return them as objects of type `AnyObject`. This means that we have to cast the values to the right type before using them. To simplify our work, the class includes methods to set and retrieve the values for every type available.

setBool(value, forKey: key)—This method stores a value of type `Bool`.

setFloat(value, forKey: key)—This method stores a value of type `Float`.

setInteger(value, forKey: key)—This method stores a value of type `Int`.

setDouble(value, forKey: key)—This method stores a value of type `Double`.

setURL(value, forKey: key)—This method stores a value of type `NSURL`. The value is converted to an `NSData` value before storing (`NSURL` and `NSData` are studied next).

boolForKey(key)—This method retrieves a value of type `Bool`. If there is no value set for the key, the method returns the value `false`.

floatForKey(key)—This method retrieves a value of type `Float`. If there is no value set for the key, the method returns the value `0.0`.

integerForKey(key)—This method retrieves a value of type `Int`. If there is no value set for the key, the method returns the value `0`.

doubleForKey(key)—This method retrieves a value of type `Double`. If there is no value set for the key, the method returns the value `0.0`.

stringForKey(key)—This method retrieves a value of type `String`.

arrayForKey(key)—This method retrieves an array.

dictionaryForKey(key)—This method retrieves a dictionary.

dataForKey(key)—This method retrieves a value of type `NSData`.

URLForKey(key)—This method retrieves a value of type `NSURL`.

The user defaults system can store any amount of data we want, but it is recommendable to use it to store simple strings and values. Its main purpose is to serve as a storage system for the app's settings. For example, we may have an app that plays videos and we want to let the user set the volume by default. Using the user defaults system we can store this value and set the volume back to its last state every time the app is executed. The interface of Figure 21-1 includes a Stepper and a label to illustrate how the process works.



Figure 21-1: Interface to store settings

The view controller for this view has to update the value of the label when the view is shown on the screen and store it in the user defaults system every time the user sets a new one.

Listing 21-1: Storing values in the user defaults system

```
import UIKit
```

```
class ViewController: UIViewController {
    @IBOutlet weak var counter: UIStepper!
    @IBOutlet weak var counterLabel: UILabel!
    var defaultValues: UserDefaults!

    override func viewDidLoad() {
        super.viewDidLoad()

        defaultValues = UserDefaults.standardUserDefaults()
        if let number = defaultValues.objectForKey("counter") as? Double {
            counter.value = number
            counterLabel.text = String(number)
        }
    }

    @IBAction func incrementValue(sender: UIStepper) {
```

```

let current = counter.value
defaultValues.setObject(current, forKey: "counter")
counterLabel.text = String(current)
}
}

```

The way we process the value is simple. We have to get a reference to the `NSUserDefaults` object of our application and then call any of the methods provided by the object to read and store new values. In Listing 21-1 we declared a property called `defaultValues` to store the reference to the object and the `objectForKey()` and `setObject()` methods to read and store a value with the key “counter”. The `objectForKey()` method returns an optional of type `AnyObject`, so we have to unwrap the value using an if statement and cast it to the real type (in this case a `Double`). If there is a value stored with the key “counter”, we assign it to the `value` property of the `Stepper` to start counting from that number and also to the label to show it on the screen.

The `Stepper` is connected to an Action called `incrementValue()`. When the user presses the buttons of the `Stepper`, this method is executed. Here we get the current value, store it in the user defaults system with the key “counter” and update the label. In consequence, the label always reflects the current value, even when the app is closed.

Do It Yourself: Create a new Single View Application project. Add a `Stepper` and a label to the main view. Assign the value “0.0” to the label. Connect the `Stepper` to the view controller with an Outlet called `counter` and the label with an Outlet called `counterLabel`. Connect the `Stepper` with an Action called `incrementValue()`. Complete the `ViewController` class with the code of Listing 21-1 and run the application. Press the buttons of the `Stepper` to change the value on the label. Stop the execution of the app from the Stop button in Xcode. Run the application again. The value of the label should be the last one selected.

As we mentioned, we can also store values generated by the app. For example, we may add a value to user settings to keep track of the time the user has spent without using the app. The following code improves the last view controller to store this value. The code reads the value when the main view is loaded, compares it with the current date, and prints the difference in the console. At the end, we update the value with the current date for the next cycle.

Listing 21-2: Storing app values

```

import UIKit

class ViewController: UIViewController {

```

```

@IBOutlet weak var counter: UIStepper!
@IBOutlet weak var counterLabel: UILabel!
var defaultValues: UserDefaults!
override func viewDidLoad() {
    super.viewDidLoad()
    defaultValues = UserDefaults.standardUserDefaults()
    if let number = defaultValues.objectForKey("counter") as? Double {
        counter.value = number
        counterLabel.text = String(number)
    }
    if let lastDate = defaultValues.objectForKey("lastDate") as? NSDate {
        let calendar = NSCalendar.currentCalendar()
        let components = calendar.components([.Year, .Month, .Day, .Hour, .Minute, .Second], fromDate: lastDate,
toDate: NSDate(), options: [])
        print("You haven't use this app in \(components.year) years, \(components.month) months, \(components.day)
days, \(components.hour) hours, \(components.minute) minutes, \(components.second) seconds")
    }
    defaultValues.setObject(NSDate(), forKey: "lastDate")
}
@IBAction func incrementValue(sender: UIStepper) {
    let current = counter.value
    defaultValues.setObject(current, forKey: "counter")
    counterLabel.text = String(current)
}
}

```

The interface of Figure 21-1 was designed for didactic purposes, but settings views are usually set apart from the main interface. Depending on the design of our app, they may be presented modally to denote its dissociation with the main interface or as part of the navigation stack of a Navigation Controller. The following example implements this last strategy to illustrate how the settings view is usually implemented in professional applications. Figure 21-2 introduces the new interface. We have embedded the initial view in a Navigation Controller, added a Text View and a bar button called Settings to the main view, and a Table View Controller with a static table containing two sections and three cells to present the controls users can manipulate to set their preferences. The cell in the first section contains a Segmented Control with three buttons to set the color of the text: Dark, Medium, and Light, and the cells in the second section contain Switches to set whether the Text View is going to be editable and have auto correction activated or not.

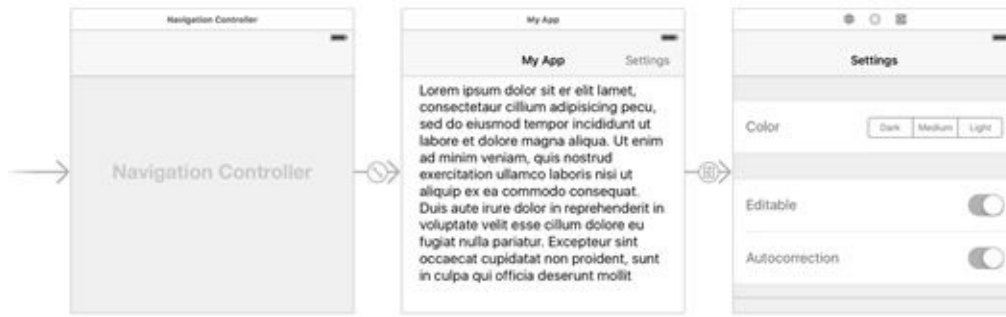


Figure 21-2: Interface with a settings view

IMPORTANT: When an element that inherits from the `UIScrollView` class, including Text Views, is the top element of the main view when embedded in a Navigation Controller, its insets are automatically adjusted to regenerate a margin between the top of the view and its content. Apple wants developers to incorporate transparency in their apps, and therefore Xcode configures the Text View to be positioned behind the Navigation Bar and let the user see the text through the bar when it is scrolled down. To take advantage of this feature you have to pin the view to the top of the main view and not to the Navigation Bar. The `UIViewController` class provides a property called `automaticallyAdjustsScrollViewInsets` that we can set to `false` in any view controller to avoid this behaviour.

The view controller for the initial view has to read the values and update the Text View.

Listing 21-3: Configuring the Text View according to the values set by the user

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var textEditor: UITextView!
    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)

        let defaultValues = UserDefaults.standardUserDefaults()
        if let color = defaultValues.objectForKey("color") as? Int {
            let colorList = [UIColor.blackColor(), UIColor.grayColor(), UIColor.lightGrayColor()]
            textEditor.textColor = colorList[color]
        }
        if let editable = defaultValues.objectForKey("editable") as? Bool {
            textEditor.editable = editable
        }
        if let correction = defaultValues.objectForKey("correction") as? Bool {
            if correction {
                textEditor.spellCheckingType = .Yes
            }
        }
    }
}
```

```

    } else {
        textEditor.spellCheckingType = .No
    }
}
}
}
}
}

```

For this example we use three values with the keys “color”, “editable”, and “correction” to store the current settings. If the values were already set, which means that the user changed them at some point in the past, we configure the Text View with those values, otherwise the application uses the values assigned in the Storyboard. For example, the color for the text in the Text View is set as black by default in the Storyboard, but if there is a value stored with the key “color” we use it to get the right color from an array (black, gray, or light gray).

In the view controller for the settings view we have to read the values to update the elements as well, but we also have to get the user’s input and store the new values for future reference.

Listing 21-4: Storing settings

```

import UIKit

class SettingsViewController: UITableViewController {
    @IBOutlet weak var controlColors: UISegmentedControl!
    @IBOutlet weak var controlEditable: UISwitch!
    @IBOutlet weak var controlCorrection: UISwitch!
    var defaultValues: UserDefaults!

    override func viewDidLoad() {
        super.viewDidLoad()

        defaultValues = UserDefaults.standardUserDefaults()
        if let color = defaultValues.objectForKey(“color”) as? Int {
            controlColors.selectedSegmentIndex = color
        }
        if let editable = defaultValues.objectForKey(“editable”) as? Bool {
            controlEditable.on = editable
        }
        if let correction = defaultValues.objectForKey(“correction”) as? Bool {
            controlCorrection.on = correction
        }
    }

    @IBAction func saveColor(sender: UISegmentedControl) {
        let current = controlColors.selectedSegmentIndex
    }
}

```

```

    defaultValues.setObject(current, forKey: "color")
}
@IBAction func saveEditable(sender: UISwitch) {
    let current = controlEditable.on
    defaultValues.setObject(current, forKey: "editable")
}
@IBAction func saveCorrection(sender: UISwitch) {
    let current = controlCorrection.on
    defaultValues.setObject(current, forKey: "correction")
}
override func tableView(tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
    let titles = ["Design", "Features"]
    return titles[section]
}
}

```

The settings view contains a Segmented Control with the buttons Dark, Medium, and Light, and two Switches to set the edition and auto correction features of the Text View. These elements were connected to Outlets that we use in the `viewDidLoad()` method to update their states according to the values already selected, and also to Actions that update the settings every time the values of the elements change (the Value Changed event is fired). In consequence, every time the user opens the settings view, the state of the elements is changed according to the values previously stored, and as soon as the user modifies the position of a Switch or taps another button on the Segmented Control, the new value is stored.

Notice also that this view controller is a subclass of the `UITableViewController` class and therefore we were able to implement the `tableView(tableView, titleForHeaderInSection: section)` method to provide a title for each section (see Listing 14-4).

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add a bar button called Settings and a Text View to the main view. Add a Table View Controller to the Storyboard. Connect the Settings button to the table with a Show Segue. Select the table, go to the Attributes Inspector panel, and change the Content option to Static Cells, the value of Sections to 2, and the Style to Grouped. Leave only one cell for the first section and two for the second section, and change their heights to 75 (Figure 14-10). Add labels, a Segmented Control with the buttons Dark, Medium, and Light, and also switches to the interface as shown in Figure 21-2. Create a subclass of the `UITableViewController` class called `SettingsViewController` and assign it to the settings view. Complete the `ViewController` class and the `SettingsViewController` class with the codes of Listings 21-3 and 21-4. Connect the elements to their respective Outlets and Actions. Run the application and set new

values. Stop the application and run it again. The Text View should be always configured according to the values you set before.

So far we have used generic methods like `setObject()` and `objectForKey()` to set and retrieve the values. These methods work with any of the types available in the user defaults system, but the methods for specific types provided by the `NSUserDefaults` class may simplify our code in some circumstances. For example, we can modify the `ViewController` class of Listing 21-3 to read the values with the methods according to their type.

Listing 21-5: Reading values with specific methods

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var textEditor: UITextView!
    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        let defaultValues = NSUserDefaults.standardUserDefaults()
        let color = defaultValues.integerForKey("color")
        let colorList = [UIColor.blackColor(), UIColor.grayColor(), UIColor.lightGrayColor()]
        textEditor.textColor = colorList[color]
        textEditor.editable = defaultValues.boolForKey("editable")

        let correction = defaultValues.boolForKey("correction")
        if correction {
            textEditor.spellCheckingType = .Yes
        } else {
            textEditor.spellCheckingType = .No
        }
    }
}
```

The code is simplified because we no longer have to unwrap the values, but there is a problem. Some of these methods return values by default. For example, the `integerForKey()` method in Listing 21-5 will return the value 0 if no value was stored before for the “color” key. In the case of the color, we actually want the value by default to be 0, so the text is displayed in color black the first time the user runs the application, but the value by default for Booleans is `false`, which means that the features we set for the Text View will be deactivated by default. The solution is to provide our own default values. The `NSUserDefaults` class includes the `registerDefaults()` method to set these values. The method receives a dictionary that contains the keys and values we want to set by default, and stores them in the user defaults system. Because the values may be modified from

anywhere in our app, it is good practice to set the values by default in the app's delegate object.

Listing 21-6: Setting values by default

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let list = ["color": 0, "editable": true, "correction": false]
        let defaultValues = UserDefaults.standardUserDefaults()
        defaultValues.registerDefaults(list)
        return true
    }
}
```

Do It Yourself: Replace the code in your **ViewController** class by the code of Listing 21-5. Update the **AppDelegate** class with the code of Listing 21-6. Open the simulator, select the option Home in the Hardware menu at the top of the screen, and delete the application to clean the user default system (the simulator works exactly like a real device. You have to keep pressing the mouse button on the icon of the app you want to delete and click on the X button to remove it). Run the application. The settings should be set according to the values you specified in the **list** dictionary.

IMPORTANT: iOS allows you to include the settings for your app in the Settings application on your device. To add your app to the list of applications in Settings you have to create a Settings Bundle. The option is available in the Resource window when you create a new file. The topic is beyond the scope of this book. For more information visit our website and follow the links for this chapter.

21.2 Files

iOS, as any other operative system, allows us to create and access files, but the iOS file system is like no other. The system restricts applications to their own space and a handful of directories to ensure that they don't interfere with each other. This means that we can only access the files and directories that belong to our app.

Several frameworks and classes use files to store data in a variety of formats. For this reason, Foundation defines a class called `NSFileManager` that has the sole purpose of managing files and directories. One object of this class is assigned to the app and from it we can create, delete, copy, and move files and directories in the storage space designated to our app. The class offers the following type method to get a reference to the object.

defaultManager()—This type method returns a reference to the `NSFileManager` object assigned to the app.

NSFileManager

The `NSFileManager` class offers multiple properties and methods to manage files and directories. The following are the most frequently used.

URLsForDirectory(directory, inDomains: domain)—This method returns an array with the location of the directory specified by the **directory** attribute. The **directory** attribute is an enumeration of type `NSSearchPathDirectory` with values that represent common system directories, such as the Documents directory. The **domain** attribute is an enumeration of type `NSSearchPathDomainMask` with values that determine the domain in which the files are located.

createDirectoryAtPath(path, withIntermediateDirectories: value, attributes: attributes)—This method creates a new directory. The **path** attribute specifies the location of the directory, including its name. The **value** attribute is a Boolean value to indicate whether or not intermediate directories will also be created. The **attributes** attribute is a dictionary with values that determine the directory's attributes (e.g., ownership). The value `nil` sets the attributes by default, which are usually enough for iOS applications.

createFileAtPath(path, contents: data, attributes: attributes)—This method creates a file. The **path** attribute specifies the location of the file, including its name and extension, the **data** attribute is an optional of type `NSData` with the content of the file, and the **attributes** attribute is a dictionary with values that determine the file's attributes.

(e.g., ownership). The value `nil` sets the attributes by default, which are usually enough for iOS applications.

contentsAtPath(path)—This method returns the content of the file at the path specified by the **path** attribute. The value returned is an optional of type `NSData`.

contentsOfDirectoryAtPath(path)—This method returns an array with the paths of the files and directories inside the directory indicated by the **path** attribute.

copyItemAtPath(origin, toPath: destination)—This method copies the file or directory at the path specified by the **origin** attribute to the path specified by the **destination** attribute.

moveItemAtPath(origin, toPath: destination)—This method moves the file or directory at the path specified by the **origin** attribute to the path specified by the **destination** attribute.

removeItemAtPath(path)—This method removes the file or directory at the path indicated by the **path** attribute.

fileExistsAtPath(path)—This method returns a Boolean value that determines whether the file or the directory at the path specified by the **path** attribute exists or not.

attributesOfItemAtPath(path)—This method returns a dictionary with the attributes of the file or dictionary at the location indicated by the **path** attribute. The `NSFileManager` class includes constants to define the attributes, including `NSFileCreationDate` (creation date), `NSFileModificationDate` (last time it was modified), `NSFileSize` (size), and `NSFileType` (type), among others.

URLs and Paths

As in any other operative system, files in iOS are organized in directories. There is a basic directory called *root* that can contain files and also other directories. These directories in turn can contain other files and directories, creating a tree-like structure. For example, we may have a directory inside the root directory called Pictures and another directory inside Pictures called Travels and inside Travels a file called hawaii.png with an image of Hawaii. To access the picture, we have to move inside the root directory, then inside the Pictures directory, and then inside the Travels directory. To reproduce this route in code, iOS adopts a conventional syntax that separates each part of the route with a forward slash (/), starting with a single slash to indicate the root directory. Using this syntax, the previous route would look like this `/Pictures/Travels/Hawaii.png`. This is called *path*, and it is used to access any file stored in the file system.

Paths are a simple way to access files, but are not enough for the system to identify the location of a file. This is because in practice files are usually not stored in a single storage

space, but in several units or even in remote servers. Following a long path to find a file also takes time and consumes resources. For these reasons, locations are always identified by URLs (Uniform Resource Locator).

URLs are the most efficient way to identify the location of a resource (usually a file) in local or remote systems. They are just strings, like paths, but provide more information to the system and follow a convention that systems can take advantage of to manage and access files in a more efficient way. Foundation defines a class to work with URLs called `NSURL`. The following are some of the properties and methods provided by the class.

path—This property returns the path of a URL. It is a conditional of type `String`.

pathComponents—This property returns an array of `String` values that represent the components of a path extracted from a URL (the strings between forward slashes).

lastPathComponent—This property returns a string with the last component of a path extracted from a URL. It is usually used to get the file's name and extension.

pathExtension—This property returns a string with the extension of the path extracted from a URL. It is usually used to get the file's extension.

URLByAppendingPathComponent(component)—This method returns a new `NSURL` object with the URL of the original object plus the component specified by the **component** attribute. The attribute is a `String` value.

IMPORTANT: The `NSFileManager` class includes methods to work with both paths and URLs. For example, you can create a directory from a path using the `createDirectoryAtPath()` method or do it from a URL with the `createDirectoryAtURL()` method. The methods that work with paths take care of converting the path into a URL and are usually easier to implement, but you can use any method you need. For a complete list, visit our website and follow the links for this chapter.

Files and Directories

When the application is installed, iOS creates a group of standard directories that we can use. The most useful are the Documents directory, where we can store a user's files, and the Application Support directory, for files that our app needs to create during run time that are not directly generated by the user. The location of all the directories included in our app is not guaranteed, so we always have to ask iOS for the current URL that points to the location of the directory and files we want to access. To get the location of common

directories like Documents, the `NSFileManager` class includes the `URLsForDirectory()` method. The method requires two attributes. The first attribute is a value that represents the directory we want to access. There are several values available for the iOS and OSX operative systems, including `DocumentDirectory` to reference the Documents directory and `ApplicationSupportDirectory` to reference the Application Support directory. The second attribute indicates the domain where the directory is located. The system organizes directories and files in separate domains depending on their intended usage. The domain where our app files are stored is the User Domain, identified with the constant `UserDomainMask`. In consequence, to get the URL of any of the directories generated for our app and start creating files to store the user's data we have to get a reference to the `NSFileManager` object and then call the `URLsForDirectory()` method with the values mentioned above.

Files can be managed from anywhere in the code, but it is a common practice to prepare the files our app requires from the app's delegate object so they are ready when needed. The following example shows how to obtain the Documents directory's URL from the `AppDelegate` class.

Listing 21-7: Getting the Documents directory's URL

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        return true
    }
}
```

The `URLsForDirectory()` method returns an array of optional `NSURL` objects with all the possible locations of the directory. In Listing 21-7 we get the first result from the `first` property, unwrap it, and assign it to the `docURL` constant.

Now that we have the URL we can start adding other directories and files inside the Documents directory. The `NSFileManager` class includes the `createFileAtPath()` method to create new files. The method requires three values, the path where the file is going to be created (including the file's name and extension), a value with the content, and the file's attributes. The content and the attributes are optionals. If we declare these values as `nil`, the file is created empty and with attributes by default.

Listing 21-8: Creating a file inside Documents

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        let newFileURL = docURL.URLByAppendingPathComponent("mytext.txt")
        let path = newFileURL.path!
        manager.createFileAtPath(path, contents: nil, attributes: nil)
        return true
    }
}
```

The path must represent the whole root to the file, including its name and extension. The `NSURL` class includes the `URLByAppendingPathComponent()` method to extend a URL. The method adds a string to the end of the URL and takes care of including the necessary forward slashes to ensure the validity of the URL. In Listing 21-8 we use this method to get the URL for the new file, then get the path of the URL from the `path` property and finally call the `createFileAtPath()` method to create the file (if the file already exists, the method does nothing). After the execution of this code, the Documents directory of this application will contain an empty file called `mytext.txt`.

The same way we created a file, we can create a directory. The method to create a directory is called `createDirectoryAtPath()`. Again, the method takes three values: the path of the new directory (including its name), a Boolean value that indicates if we want to create all the directories included in the path that doesn't exist (in case we are declaring a long path with directories we haven't created yet), and the directory's attributes. This method throws an error if it can't complete the task, so we have to handle the error with the `try` keyword.

Listing 21-9: Creating a directory inside Documents

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
var window: UIWindow?
```

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject:
AnyObject]?) -> Bool {
    let manager = NSFileManager.defaultManager()
    let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
    let docURL = documents.first!
    let newDirectoryURL = docURL.URLByAppendingPathComponent("myfiles")
    let path = newDirectoryURL.path!
    do {
        try manager.createDirectoryAtPath(path, withIntermediateDirectories: false, attributes: nil)
    } catch {
        print("The directory already exists")
    }
    return true
}
```

The `NSFileManager` class also offers methods to list the content of a directory. We can use the `contentsOfDirectoryAtPath()` method to get an array of strings with the names of the files and directories in a specific path. In the following example, we create another method inside the `AppDelegate` class to list files and directories. The method receives the URL we want to search and then creates a `for in` loop to print the items found.

Listing 21-10: Listing the content of a directory

```
import UIKit
```

```
@UIApplicationMain
```

```
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
    var window: UIWindow?
```

```
    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject:
AnyObject]?) -> Bool {
```

```
        let manager = NSFileManager.defaultManager()
```

```
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
```

```
        let docURL = documents.first!
```

```
        listItems(docURL)
```

```
        return true
```

```
    }
```

```
    func listItems(directory: NSURL) {
```

```
        let manager = NSFileManager.defaultManager()
```

```
        let listOfItems = try? manager.contentsOfDirectoryAtPath(directory.path!)
```

```

if let list = listOfItems {
    if list.isEmpty {
        print("The directory is empty")
    } else {
        for item in list {
            print(item)
        }
    }
}
}

```

In the `application(application, didFinishLaunchingWithOptions: options)` method we get the Documents directory's URL and then call the `listItems()` method to list the files and directories inside Documents. The `listItems()` method has to get its own reference of the `NSFileManager` object and then use it to call the `contentsOfDirectoryAtPath()` method to get the list of items. Notice that this method throws, so we use the `try?` keyword to handle the errors. If the value returned is not `nil` and not empty, we print every item on the console.

Do It Yourself: Create a new Single View Application project. Update the `AppDelegate` class with the code of Listing 21-8 and run the application to create the `mytext.txt` file. Update the `AppDelegate` class again with the code of Listing 21-9 and run the application to create the `myfiles` directory. Update the `AppDelegate` class one more time with the code of Listing 21-10 and run the application. You should see the file and directory's names printed in the console.

In the example of Listing 21-8 we have created a file inside the Documents directory, but we can also create files inside the directories our app generates. For example, we can create a new file inside the `myfiles` directory created in Listing 21-9.

Listing 21-11: Creating files in a directory inside Documents

```

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
    }
}

```

```

let docURL = documents.first!

let mydirectoryURL = docURL.URLByAppendingPathComponent("myfiles")
let newFileURL = mydirectoryURL.URLByAppendingPathComponent("anotherfile.txt")

let path = newFileURL.path!
let created = manager.createFileAtPath(path, contents: nil, attributes: nil)
if !created {
    print("We couldn't create the file")
}
return true
}
}

```

The code of Listing 21-11 adds the myfiles component to the Documents directory's URL and then adds the anotherfile.txt component to the previous URL to get the final URL necessary to create the file inside the directory. The rest of the process is the same as before.

We can also move a file or directory from one location to another. All we have to do is to provide the paths for the origin and destination and then call the `moveItemAtPath()` method. For example, we can move the mytext.txt file to the myfiles directory created before.

Listing 21-12: Moving files

```

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
let originURL = docURL.URLByAppendingPathComponent("mytext.txt")
var destinationURL = docURL.URLByAppendingPathComponent("myfiles")
destinationURL = destinationURL.URLByAppendingPathComponent("mytext.txt")
        let originPath = originURL.path!
        let destinationPath = destinationURL.path!
        do {
try manager.moveItemAtPath(originPath, toPath: destinationPath)
        } catch {

```



```

    print("File was not moved")
}
return true
}
}

```

The same way we move, we can also copy. The following example makes a copy of the `anotherfile.txt` file created inside the `myfiles` in Listing 21-11 to the Documents directory.

Listing 21-13: Copying files

```

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        var originURL = docURL.URLByAppendingPathComponent("myfiles")
        originURL = originURL.URLByAppendingPathComponent("anotherfile.txt")
        let destinationURL = docURL.URLByAppendingPathComponent("anotherfile.txt")

        let originPath = originURL.path!
        let destinationPath = destinationURL.path!
        do {
            try manager.copyItemAtPath(originPath, toPath: destinationPath)
        } catch {
            print("File was not copied")
        }
        return true
    }
}

```

Removing files or directories also doesn't take too much work. We have to get the path to the file and call the `removeItemAtPath()` method.

Listing 21-14: Removing files

```

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        let fileURL = docURL.URLByAppendingPathComponent("anotherfile.txt")

        let path = fileURL.path!
        do {
            try manager.removeItemAtPath(path)
        } catch {
            print("File was not removed")
        }
        return true
    }
}

```

Do It Yourself: Update the **AppDelegate** class with the codes of Listings 21-11, 21-12, 21-13, and 21-14, and run the application every time. Incorporate the **listItems()** method of Listing 21-10 to the class so you can see the files being created and moved.

Files Attributes

Some applications need to know more than the name of the file. The **NSFileManager** class offers the **attributesOfItemAtPath()** method to get file attributes, such as the date of creation or the size. The method returns a dictionary with predefined keys to identify each value. There are several constants available we can use as keys. The most common are **NSFileCreationDate** (creation date), **NSFileModificationDate** (last time it was modified), **NSFileSize** (size), and **NSFileType** (type). The following code gets the attributes of the mytext.txt file created in previous examples.

Listing 21-15: Reading file's attributes

```

import UIKit

```

```

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        var fileURL = docURL.URLByAppendingPathComponent("myfiles")
        fileURL = fileURL.URLByAppendingPathComponent("mytext.txt")

        let filePath = fileURL.path!
        if let attributes = try? manager.attributesOfItemAtPath(filePath) {
            let type = attributes[NSFileType] as! String
            let size = attributes[NSFileSize] as! Int
            let date = attributes[NSFileCreationDate] as! NSDate
            if type != NSFileTypeDirectory {
                print("Name: \(fileURL.lastPathComponent)")
                print("Size: \(size)")
                print("Created: \(date)")
            }
        }
        return true
    }
}

```

The attributes are returned as **AnyObject** values, so we have to cast them to their corresponding types. The **NSFileType** key returns in turn another key to identify the type of resource. The key for files is **NSFileTypeRegular** and the key for directories is **NSFileTypeDirectory**. Comparing the attribute with those values we are able to know if the item is a file or a directory. Notice that the dictionary doesn't include any key for the name of the file, but we can read the **lastPathComponent** property of the URL to get it.

Files Content

Storage systems, like hard drives and flash memories, store information the only way a computer knows, as a series of ones and zeros. Therefore, the information we want to store in files has to be converted into a stream of bytes that can be later turned back into the original data. Foundation offers the **NSData** class for this purpose. The class provides properties and methods to process raw data, but it is a basic class and therefore we usually

have to rely on properties and methods of other classes to create the instances. For example, the **String** structure includes a method that turns a string into data and also an initializer that can get back the string from an **NSData** object.

String(data: data, encoding: encoding)—This initializer creates a **String** value with the text in the **NSData** object provided by the **data** attribute. The **encoding** attribute is a predefined constant that determines the type of encoding used to generate the string. The encoding usually depends on the language the text was written in. The most frequently used are **NSUTF8StringEncoding** and **NSASCIIStringEncoding**.

dataUsingEncoding(encoding)—This method returns an **NSData** object containing the string in the original **String** value. The **encoding** attribute is a predefined constant that determines the type of encoding used to generate the data. The most frequently used are **NSUTF8StringEncoding** and **NSASCIIStringEncoding**.

IMPORTANT: Several frameworks and classes offer methods to convert their values into **NSData** objects and vice versa. For example, the **UIKit** framework includes the functions **UIImageJPEGRepresentation()** and **UIImagePNGRepresentation()** to convert a **UIImage** object into data. For more information, visit our website and follow the links for this chapter.

The structures and classes capable of processing **NSData** objects and files also provide specific methods to write and read a file. For example, the **String** structure includes the following method to turn a string into data and store it in a file, all at once.

writeToURL(URL, atomically: value, encoding: encoding)—This method converts a string into an **NSData** object and store it in the file located at the **URL** specified by the **URL** attribute. The **value** attribute is a Boolean value that determines if we want the data to be stored in an auxiliary file first to ensure that the original file is not corrupted (the value **true** is recommended). The **encoding** attribute is a predefined constant that determines the type of encoding used to generate the data. The most frequently used are **NSUTF8StringEncoding** and **NSASCIIStringEncoding**.

With methods like this we can store the information generated by the user and always keep the file updated. For example, we can get input from the user through a **Text View** and persistently preserve in a file everything the user writes.



Figure 21-3: Interface to generate file content

The view controller for the view of Figure 21-3 has to perform two tasks: write the text inside the Text View in a file every time the user changes the content and read the file back every time the user loads the application to show the Text View as it was the last time.

Listing 21-16: Managing the file’s content

```
import UIKit

class ViewController: UIViewController, UITextViewDelegate {
    @IBOutlet weak var diaryText: UITextView!
    var fileURL: NSURL!

    override func viewDidLoad() {
        super.viewDidLoad()

        diaryText.delegate = self
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        fileURL = docURL.URLByAppendingPathComponent("userdata.txt")

        let filePath = fileURL.path!
        if manager.fileExistsAtPath(filePath) {
        if let content = manager.contentsAtPath(filePath) {
            diaryText.text = String(data: content, encoding: NSUTF8StringEncoding)
        }
        else {
            manager.createFileAtPath(filePath, contents: nil, attributes: nil)
        }
    }

    func textViewDidChange(textView: UITextView) {
        let text = diaryText.text
    }
}
```

```

do {
    try text.writeToURL(fileURL, atomically: true, encoding: NSUTF8StringEncoding)
} catch {
    print("Error")
}
}
}

```

The first thing we do in the **ViewController** class of Listing 21-16 is to check if the file where we are going to store the data already exists calling the **fileExistsAtPath()** method. If the file exists, we read its content with the **contentsAtPath()** method, convert the data into a string with the **String** initializer, and assign the string to the Text View. This way the user always sees the last text stored in the file as soon as the app is launched. If the file doesn't exist, however, we create it with the **createFileAtPath()** method. To keep the file updated with the text inserted by the user in the Text View, we declared the **ViewController** class as the delegate of the Text View and implemented the **textViewDidChange()** method (see Chapter 7). This method is called by the Text View every time its content changes. Inside the method we read the current text and call its **writeToURL()** method to store it in the file. Now everything the user writes in the Text View is persistently preserved and always available.

Do It Yourself: Create a new Single View Application project. Add a label and a Text View to the initial view to get the interface of Figure 21-3. Connect the Text View to the **ViewController** class with an Outlet called **diaryText**. Complete the **ViewController** class with the code of Listing 21-16. Run the application and insert a text in the Text View. Stop the application from Xcode and run it again. You should see on the screen the same text inserted before.

NSBundle

In previous examples we have worked with files created at run time, but sometimes we need to get access to files added to our project during development. This could be simple text files or even complex databases. But the app's files, code, and resources are not just stored in a single directory; they are encapsulated in a bundle. Bundles are directories assigned exclusively to each application. They create a hierarchical structure to organize all the app's files and resources. For this reason, to get access to the files in our project, we have to access the content of our app's bundle. To create and manage bundles, Foundation includes the **NSBundle** class. The class offers properties and methods to work with bundles and get their location, including a type method that returns a reference to the bundle created by default for our app.

mainBundle()—This type method returns a reference to the app’s bundle.

bundleURL—This property returns an `NSURL` object with the bundle’s URL.

bundlePath—This property returns the path of the bundle’s URL.

Because we are not able to determine the location of our app’s files and resources during development, every time we want to access these files from code we have to get their URLs from the `NSBundle` object. The class provides the following methods for this purpose.

URLForResource(name, withExtension: extension)—This method returns an `NSURL` object with the URL of a file or directory inside the bundle. The **name** attribute is a string that specifies the name of the file or directory, and the **extension** attribute is a string that specifies its extension (set as `nil` for resources that don’t have an extension).

pathForResource(name, ofType: extension)—This method returns the path of a file or directory inside the bundle. The **name** attribute is a string that specifies the name of the file or directory, and the **extension** attribute is a string that specifies its extension (set as `nil` for resources that don’t have an extension).

We frequently have to use the `NSBundle` object in professional applications to access files that are required for some services, like databases, for example, but we can also take advantage of this object to load data we need to populate the model when the app is launched for the first time or to restore an initial state. The following example loads a text file called `quote.txt` from the bundle as soon as the application is launched. The file must be added to the project during development by dragging the file from Finder to the Navigation Area (Figures 8-3 and 8-4).

Listing 21-17: Loading a project’s file

```
import UIKit
```

```
@UIApplicationMain
```

```
class AppDelegate: UIResponder, UIApplicationDelegate {
```

```
    var window: UIWindow?
```

```
    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
```

```
        let bundle = NSBundle.mainBundle()
```

```

let filePath = bundle.pathForResource("quote", ofType: "txt")
let manager = NSFileManager.defaultManager()
if let data = manager.contentsAtPath(filePath!) {
let message = String(data: data, encoding: NSUTF8StringEncoding)
print(message)
}
return true
}
}

```

The code of Listing 21-17 gets a reference to the app's bundle and then finds the path for the `quote.txt` file with its `pathForResource()` method. After we get the path, we can read the file as we did before. In this example we print the string in the console, but this information is usually stored in the model or assigned to other classes that need it.

Do It Yourself: Create a new Single View Application project and update its `AppDelegate` class with the code of Listing 21-17. Create a text file with a text editor, write some text in it, save it with the name `quote.txt`, and add it to your project (Figures 8-3 and 8-4). Run the application. You should see the file's text printed in the console.

IMPORTANT: Getting files from the bundle is necessary for non-conventional resources. Conventional resources, like images, provide their own mechanisms to load files, as we have seen in Chapter 8.

21.3 Archiving

The methods studied before to store data in files are enough for simple models but present some limitations. We can only work with single values and with classes that already provide a way to turn their content into data. Complex applications rely on more elaborated models that include collection of values and also custom objects. To give us more flexibility, Foundation offers the `NSCoder` class. This class can encode and decode values to `NSData` objects for storage purposes in a process called *Archiving* (`NSCoder` is also used for data distribution but archiving is its most important feature).

Archiving with `NSCoder` is a very powerful tool. An `NSCoder` object not only encodes an object but also the objects it is connected to, preserving the connections and the hierarchy. For example, we may have two objects with properties that reference the other object. Object 1 references Object 2 and Object 2 references Object 1. The structure built by the objects and their connections is called *Object Graph*. If we only store Object 1, we will have to recreate Object 2 and reconstruct the references, but storing the complete Object Graph we don't have to worry about it. Both objects are encoded, stored, and then decoded and connected again when we need them. This is how `NSCoder` works. We store one object and all the objects and connections in its Object Graph are stored along with it.

The objects are encoded and decoded internally, from the methods provided by the `NSCoder` class. The class includes the following methods to encode values.

encodeBool(value, forKey: key)—This method encodes a value of type `Bool`. The **value** attribute is the value we want to encode and the **key** attribute is a string to identify the value.

encodeInteger(value, forKey: key)—This method encodes a value of type `Int`. The **value** attribute is the value we want to encode and the **key** attribute is a string to identify the value.

encodeFloat(value, forKey: key)—This method encodes a value of type `Float`. The **value** attribute is the value we want to encode and the **key** attribute is a string to identify the value.

encodeDouble(value, forKey: key)—This method encodes a value of type `Double`. The **value** attribute is the value we want to encode and the **key** attribute is a string to identify the value.

encodeCGPoint(value, forKey: key)—This method encodes a value of type `CGPoint`. The **value** attribute is the value we want to encode and the **key** attribute is a string to identify the value.

encodeCGSize(value, forKey: key)—This method encodes a value of type `CGSize`. The **value** attribute is the value we want to encode and the **key** attribute is a string to

identify the value.

encodeCGRect(value, forKey: key)—This method encodes a value of type **CGRect**. The **value** attribute is the value we want to encode and the **key** attribute is a string to identify the value.

encodeObject(value, forKey: key)—This method encodes objects, arrays, strings, structures, and other types of values that don't have a specific method. The **value** attribute is the value we want to encode and the **key** attribute is a string to identify the value.

And the following are the methods to decode the same type of values.

decodeBoolForKey(key)—This method decodes a value of type **Bool**. The **key** attribute is a string with the key used to encode the value.

decodeIntegerForKey(key)—This method decodes a value of type **Integer**. The **key** attribute is a string with the key used to encode the value.

decodeFloatForKey(key)—This method decodes a value of type **Float**. The **key** attribute is a string with the key used to encode the value.

decodeDoubleForKey(key)—This method decodes a value of type **Double**. The **key** attribute is a string with the key used to encode the value.

decodeCGPointForKey(key)—This method decodes a value of type **CGPoint**. The **key** attribute is a string with the key used to encode the value.

decodeCGSizeForKey(key)—This method decodes a value of type **CGSize**. The **key** attribute is a string with the key used to encode the value.

decodeCGRectForKey(key)—This method decodes a value of type **CGRect**. The **key** attribute is a string with the key used to encode the value.

decodeObjectForKey(key)—This method decodes objects, arrays, strings, structures, and other types of values that don't have a specific method. The **key** attribute is a string with the key used to encode the value.

Every object that wants to be encoded and decoded has to adopt the **NSCoding** protocol and implement the following initializer and method.

init?(coder aDecoder: NSCoder)—This initializer uses the decoding methods provided by the **NSCoder** class to decode the values and recreate the original object.

encodeWithCoder(aCoder: NSCoder)—This method uses the encoding methods provided by the `NSCoder` class to encode the object's values.

Predefined classes already adopt the protocol and implement its initializer and method, but for custom classes we have to do it ourselves, as we will see later.

Encoding and Decoding

The `NSCoder` class provides all the methods necessary to encode and decode the values of an object, but all the work is done by instances of `NSCoder` subclasses called `NSKeyedArchiver` and `NSKeyedUnarchiver`. The `NSKeyedArchiver` class calls the method `encodeWithCoder()` on the objects to encode their values and stores the data in an `NSData` object or a file. The `NSKeyedUnarchiver` class, on the other hand, initializes the objects with the protocol's initializer and returns the original values.

The `NSKeyedArchiver` class offers the following type methods to encode an Object Graph and store it in a file or in an `NSData` object.

archiveRootObject(object, toFile: path)—This type method encodes the Object Graph of the object specified by the **object** attribute and stores the resulting `NSData` object in the file located at the path specified by the **path** attribute.

archivedDataWithRootObject(object)—This type method encodes the Object Graph of the object specified by the **object** attribute and returns an `NSData` object with the data.

The `NSKeyedUnarchiver` class offers the following type methods to decode an Object Graph from a file or an `NSData` object.

unarchiveObjectWithFile(path)—This type method decodes the data in the file located at the path specified by the **path** attribute and returns the original Object Graph.

unarchiveObjectWithData(data)—This type method decodes the data in the `NSData` object specified by the **data** attribute and returns the original Object Graph.

We can take different approaches to encode and decode values. Implementing the `NSKeyedArchiver` and `NSKeyedUnarchiver` methods we can generate `NSData` objects for processing or we can store and retrieve the data directly from a file. In the following

example we encode and decode a string to an `NSData` object and use `NSFileManager` methods to create and read the file.

Listing 21-18: Encoding and decoding data

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        let fileURL = docURL.URLByAppendingPathComponent("quotes.dat")
        let filePath = fileURL.path!

        if manager.fileExistsAtPath(filePath) {
            if let content = manager.contentsAtPath(filePath) {
                let result = NSKeyedUnarchiver.unarchiveObjectWithData(content) as! String
                print(result)
            }
        } else {
            let quote = "Fiction is the truth inside the lie"
            let fileData = NSKeyedArchiver.archivedDataWithRootObject(quote)
            manager.createFileAtPath(filePath, contents: fileData, attributes: nil)
        }
    }
}
```

Listing 21-18 presents a view controller with a simple example. A single string is encoded and stored in a file called `quotes.dat`. Since the object we are processing is a `String` value (and therefore it is convertible to an `NSString` object), it already adopts the `NSCoding` protocol and implements the `NSCoder` methods, so all we have to do is to call the `NSKeyedArchiver` and `NSKeyedUnarchiver` methods to encode and decode the value, respectively. As we did before, we check if the file exists with the `fileExistsAtPath()` method first and then proceed according to the result. If the file doesn't exist, we convert the string in the `quote` constant to an `NSData` value with the `archivedDataWithRootObject()` method and then create a file with this data as its content, but if the file already exists, we read its content with the `contentsAtPath()` method and decode the data with the `unarchiveObjectWithData()` method to get back the string. Notice that this method returns an optional object of type `AnyObject`, so we have to cast it to the original type.

The process may be simplified a little by implementing the other options provided by

the `NSKeyedArchiver` and `NSKeyedUnarchiver` classes. Instead of working with `NSData` values we store and read the data directly from the file with the `archiveRootObject()` and `unarchiveObjectWithFile()` methods.

Listing 21-19: Encoding and decoding data from file

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        let fileURL = docURL.URLByAppendingPathComponent("quotes.dat")
        let filePath = fileURL.path!

        if manager.fileExistsAtPath(filePath) {
            let result = NSKeyedUnarchiver.unarchiveObjectWithFile(filePath) as! String
            print(result)
        } else {
            let quote = "Fiction is the truth inside the lie"
            NSKeyedArchiver.archiveRootObject(quote, toFile: filePath)
        }
    }
}
```

Do It Yourself: Create a new Single View Application project. Update the `ViewController` class with the code of Listing 21-18. Run the application. The first time, the file is created and no message is printed on the console. Run the application again. Now you should see the quote on the console. Update the view controller with the code of Listing 21-19. Run the application one more time. The quote should be printed again in the console. No matter the changes we make in the code that processes the data, the file and its content are never erased.

Archiving was developed for Objective-C classes, so it only works with classes that inherit from the `NSObject` class (the elemental class in Foundation) and adopt the `NSCoding` protocol. Predefined classes are ready for archiving, but custom classes in Swift require some work. We have to make the class a subclass of `NSObject`, adopt the `NSCoding` protocol, implement its initializer and method, and encode and decode every property of the class with the `NSCoder` methods. As an example we have created a custom class called `Book` to

store the title, author, and edition of a book.

Listing 21-20: Encoding and decoding values in a custom class

```
import UIKit
```

```
class Book: NSObject, NSCoding {
    var title: String
    var author: String
    var edition: Int

    init(btitle: String, bauthor: String, bedition: Int) {
        title = btitle
        author = bauthor
        edition = bedition
    }

    func encodeWithCoder(aCoder: NSCoder) {
        aCoder.encodeObject(title, forKey: "title")
        aCoder.encodeObject(author, forKey: "author")
        aCoder.encodeInteger(edition, forKey: "edition")
    }

    required init?(coder aDecoder: NSCoder) {
        title = aDecoder.decodeObjectForKey("title") as! String
        author = aDecoder.decodeObjectForKey("author") as! String
        edition = aDecoder.decodeIntegerForKey("edition")
    }
}
```

The **Book** class introduced in Listing 21-20 defines three properties to store the title, author, and edition of a book and an initializer to set the initial values. To allow the objects created from this class to be encoded and decoded, the class inherits from the **NSObject** class and adopts the **NSCoding** protocol. The protocol method and initializer are in charge of encoding and decoding the object's properties. When a **Book** object is sent to any of the **NSKeyedArchiver** methods, the methods call the **encodeWithCoder(aCoder: NSCoder)** method on the object to encode its values. In this method we have to call the **NSCoder** methods to encode the values of the properties one at a time (we don't need to encode all the properties, only those necessary for the object to be completely functional when it is decoded later). The **NSCoder** methods take a property and a string that identifies the value. Depending on the data type of the property, we have to use one method or another. For example, the **title** and **author** properties are **String** values, and therefore they are internally converted to **NSString**, so they are considered objects and must be encoded with the **encodeObject()** method, but the **edition** property is a value of type **Int**, so we have to encode it with the **encodeInteger()** method.

On the other side of the process, when data containing a **Book** object is sent to any of the **NSKeyedUnarchiver** type methods, the methods initialize a new **Book** object calling the protocol's initializer. In this initializer we have to execute the **NSCoder** methods to decode every value. Again, every value must be decoded with the right method according to its type, but we also have to cast the values of objects because the **decodeObjectForKey()** method returns a conditional value of type **AnyObject**.

Once our custom class is ready, we can start creating instances and archiving them. The next example follows the process established before. When the main view is loaded, we check if the file already exists and read it or create a new one accordingly. To archive and unarchive the data we use the same methods as in the previous example, but this time we store an array of objects instead of a single one. This shows the versatility of the **NSKeyedArchiver** and **NSKeyedUnarchiver** classes and all the possible applications we can create with them.

Listing 21-21: Encoding and decoding arrays of custom objects

import UIKit

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        let fileURL = docURL.URLByAppendingPathComponent("userdata.dat")
        let filePath = fileURL.path!

        if manager.fileExistsAtPath(filePath) {
            let results = NSKeyedUnarchiver.unarchiveObjectWithFile(filePath) as! [Book]
            for book in results {
                print("\(book.title) - \(book.author) - \(book.edition)")
            }
        } else {
            let book1 = Book(btitle: "IT", bauthor: "Stephen King", bedition: 2)
            let book2 = Book(btitle: "Pet Sematary", bauthor: "Stephen King", bedition: 1)
            let book3 = Book(btitle: "The Shining", bauthor: "Stephen King", bedition: 1)
            let list = [book1, book2, book3]
            NSKeyedArchiver.archiveRootObject(list, toFile: filePath)
        }
    }
}
```

Do It Yourself: Create a new Single View Application project. Create a new Swift file for the class of Listing 21-20. Update the `ViewController` class with the code of Listing 21-21. Run the application. As it happened with the previous example, the file is created and nothing is printed on the console, but if you run the application again, the view controller decodes the content of the file created before and prints the values of every `Book` object in the console.

Real-Life Application

The possibility to encode and decode collections of objects, standard or custom, allows us to store the entire model of our app in files using archiving. In the following example we are going to build a small application to insert names in the model and list them in a table. The names are going to be managed by a model that archives all the values and stores them in a file for future reference, so every time the app is launched, the values in the model are recovered from the file.

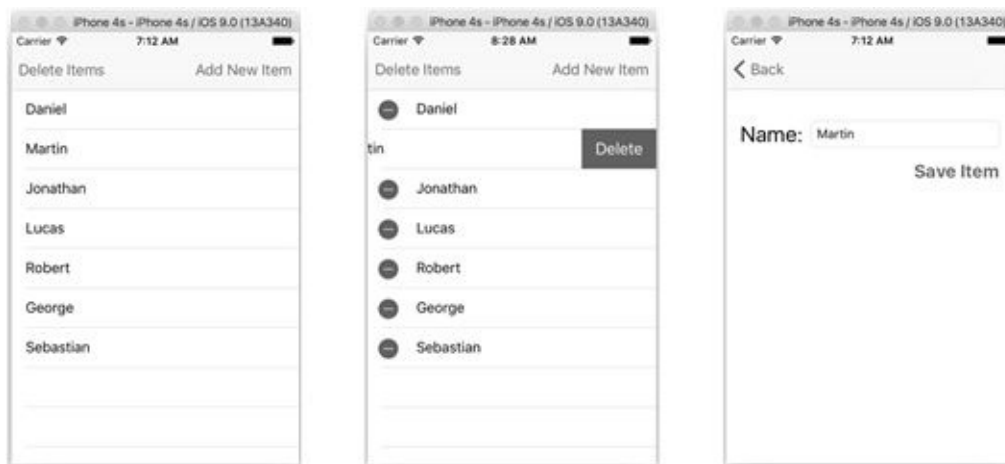


Figure 21-4: Application to store a list of names in file

Figure 21-5, next, shows the interface required for this example. We have to create a new Single View Application project, delete the initial view, add a Table View Controller to the now empty Storyboard, and embed it in a Navigation Controller. The Navigation Controller has to be declared as the initial view controller from the Attributes Inspector panel, and we also have to add a second view to let the user insert new names and update current values (Figure 21-5, right).

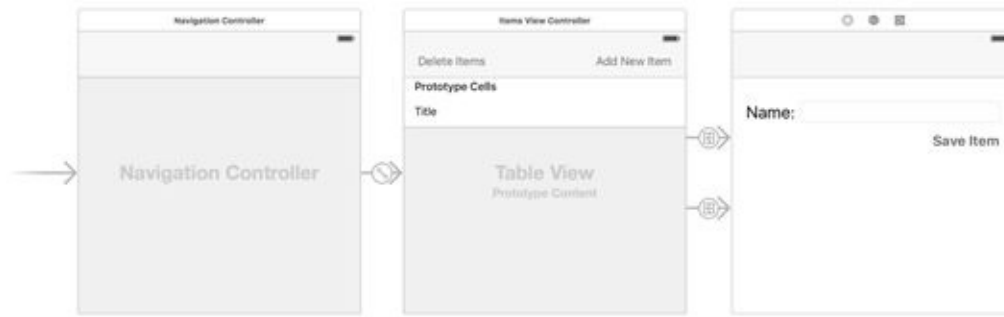


Figure 21-5: Interface to test a model with archiving

We have to configure the cell with the Basic style and the identifier “myCell”. The Table View Controller also needs two bar buttons, one to edit the table and another to open the second view to add new values. We called these buttons Delete Items and Add New Item (Figure 21-5, center). In the second view, we have to include a Text Field to allow the user to insert a name and a button to save it (Figure 21-5, right).

The second view is going to be opened two times: when the user wants to add a new name and when the user wants to update an existing one. To manage these situations, we have to create two Show Segues connected to the second view, one from the Add New Item button and another from the cell. After the Segues are created, the interface will look like Figure 21-5. To identify which Segue was triggered, we have to select the Segue that connects the cell to the second view and give it the identifier “editItem”.

The interface is ready; now it is time to create the model. We only need one property to store the list of names, but we want to store the data on file every time it is modified, so we decided to expand previous versions and create a more professional model that privatizes its properties and only provides access to the data from methods.

Listing 21-22: Archiving model’s data

```
import Foundation

struct ApplicationData {
    private var items: [String]
    private var filePath: String

    init() {
        items = []

        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        let fileURL = docURL.URLByAppendingPathComponent("items.dat")
        filePath = fileURL.path!

        if manager.fileExistsAtPath(filePath) {
```

```

    items = NSKeyedUnarchiver.unarchiveObjectWithFile(filePath) as! [String]
  } else {
    NSKeyedArchiver.archiveRootObject(items, toFile: filePath)
  }
}

func getItem(atIndex: Int) -> String {
    return items[atIndex]
}

func getTotal() -> Int {
    return items.count
}

mutating func addNewItem(name: String) {
    items.append(name)
    NSKeyedArchiver.archiveRootObject(items, toFile: filePath)
}

mutating func removeItem(index: Int) {
    items.removeAtIndex(index)
    NSKeyedArchiver.archiveRootObject(items, toFile: filePath)
}

mutating func updateItem(index: Int, name: String) {
    items[index] = name
    NSKeyedArchiver.archiveRootObject(items, toFile: filePath)
}
}

var AppData = ApplicationData()

```

When the model is initialized, we have to get the path to the file we are going to use to store the data. This is the same process we have followed in previous examples, but the difference this time is that we have to store the data in the file again every time it changes. For this purpose we have created three methods: **addNewItem()**, **removeItem()**, and **updateItem()**. The **addNewItem()** method takes the new name inserted by the user, adds it to the **items** array, and then archives the array again with the **archiveRootObject()** method. The **removeItem()** method performs a similar task, but instead of adding a new name, it removes the item at the index specified by the **index** attribute. Finally, the **updateItem()** method replaces the current value of the item at the index specified by the **index** attribute with the new value specified by the **name** attribute.

IMPORTANT: Notice that we have initialized the file with an empty array (**items = []**). This is not only allowed but also required because the methods provided by the **NSKeyedArchiver** and **NSKeyedUnarchiver** classes can't work with empty files.

Because the `items` property was declared as `private`, it won't be accessible from other structures and classes so we have to create methods to let other objects read its values and get the total amount of elements it contains. This is why we have also added the `getItem()` and `getTotal()` methods to the `AppData` structure. The `getItem()` method returns the value of the item at the index specified by the `atIndex` attribute, and the `getTotal()` method returns an integer with the total amount of elements in the array.

With the model ready, it is time to show the names in the Table View and let the user edit the values. Because we erased the initial main view and replaced it with a Table View Controller, we have to create a new file with a subclass of `UITableViewController` to work as the view controller of the table. We have called this subclass `ItemsViewController`.

Listing 21-23: Displaying the model's data

```
import UIKit

class ItemsViewController: UITableViewController {
    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        tableView.reloadData()
    }

    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return AppData.getTotal()
    }

    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
    UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier("myCell", forIndexPath: indexPath)
        cell.textLabel?.text = AppData.getItem(indexPath.row)
        return cell
    }

    override func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle,
    forRowAtIndexPath indexPath: NSIndexPath) {
        if editingStyle == .Delete {
            AppData.removeItem(indexPath.row)
            tableView.deleteRowsAtIndexPaths([indexPath], withRowAnimation: .Automatic)
        }
    }

    @IBAction func editItems(sender: UIBarButtonItem) {
        if tableView.editing {
            tableView.setEditing(false, animated: true)
        } else {
            tableView.setEditing(true, animated: true)
        }
    }
}
```

```

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "editItem" {
        let controller = segue.destinationViewController as! EditItemViewController
        if let indexPath = tableView.indexPathForSelectedRow {
            controller.index = indexPath.row
            controller.name = AppData.getItem(indexPath.row)
            tableView.deselectRowAtIndexPath(indexPath, animated: true)
        }
    }
}

```

Nothing has changed in the code of Listing 21-23 from previous examples of Table View Controllers (see Chapter 14). The model is in charge of archiving the data, so the only thing the view controller has to do is to call the model's methods to get the values it has to show inside the cells. Besides the protocol methods necessary for the Table View, we have also implemented an Action for the Delete Items button called `editItems()` to activate and deactivate the table's edition feature, and the `prepareForSegue()` method to send the value of the selected row to the second view when a row is selected.

For the second view we need a subclass of `UIViewController`. We called it **EditItemViewController**.

Listing 21-24: Adding and updating the model's values

```

import UIKit

class EditItemViewController: UIViewController {
    @IBOutlet weak var nameText: UITextField!
    var index: Int!
    var name: String!

    override func viewDidLoad() {
        super.viewDidLoad()
        if name != nil {
            nameText.text = name
        }
    }

    @IBAction func saveItem(sender: UIButton) {
        if index != nil && name != nil {
            let text = nameText.text!
            AppData.updateItem(index, name: text)
        } else {

```

```
let text = nameText.text!  
AppData.addItem(text)  
}  
navigationController?.popViewControllerAnimated(true)  
}  
}
```

This view controller has to contemplate two scenarios: users can open the view by selecting a row when they want to edit its value and also by pressing the Add New Item button in the Navigation Bar when they want to add a new name. We determine the situation by checking the content of the **index** and **name** properties. If their values are not **nil**, it means that the user selected a row to update its value, so we assign the value of the **name** property to the Text Field and call the **updateItem()** method in the model when the Save Item button is pressed to update the value. If, on the other hand, the values of **index** and **name** are **nil**, it means that the user pressed the Add New Item button to add a new name to the model, so we call the **addItem()** method instead.

Do It Yourself: Remember to connect the Outlets and Actions to their respective elements. Run the application and press the Add New Item button. Insert a new name and press the Save Item button, as shown in Figure 21-4. Stop the application from Xcode and run it again. The name you have inserted should be displayed in the Table View again.

Chapter 22

Core Data

22.1 Custom Object Graph

With archiving we can store not only objects but also their connections. As we mentioned in Chapter 21, this structure of objects and the connections between them is called Object Graph. Archiving is a good tool to store an Object Graph on file, but presents some limitations. The Object Graph we can store is difficult to expand and usually inflexible. The entire graph has to be stored in the file again after the smallest change, and it is hard to control the connections between objects and determine exactly which objects are going to be stored. The solution is called *Core Data*. Core Data is an Object Graph manager that defines and manages its own objects and connections. We can establish the composition of those objects and how they are going to relate to each other, and also store as many objects as we need. The system takes care of encoding and decoding the objects, preserving consistency, and storing the Object Graph on file in an efficient way.

Data Model

The structure of the Core Data's Object Graph is defined with a data model. This has nothing to do with the data model of the MVC pattern created in previous chapters, it is a definition of the type of objects the graph is going to contain (called *Entities*) and their connections (called *Relationships*).

A model can be created entirely by code, but Xcode offers a practical editor that we can use to define the structure of the graph. The model is stored in a file and then the file is compiled and included in the Core Data system created for our app. Xcode offers a template to create this file, among with other Core Data options. The file has to be created with the extension `xcdatamodeld` and any name we want.

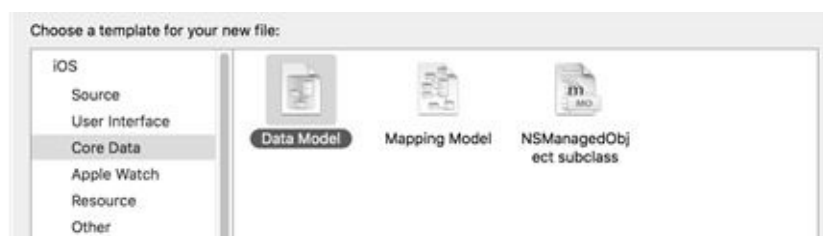


Figure 22-1: Template for a Core Data model

Once created, the file is included in our project. Clicking on it reveals the Xcode editor, illustrated by Figure 22-2, next.

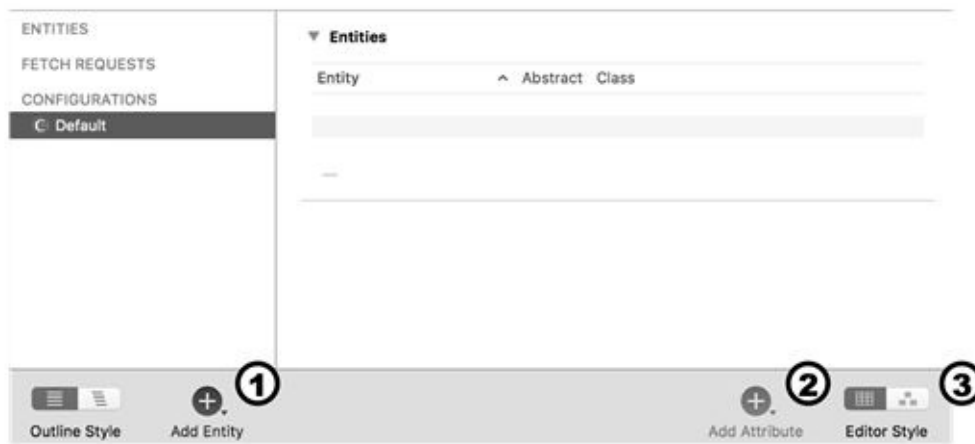


Figure 22-2: Model editor

The model contains three main components: Entities, Attributes, and Relationships. Entities are the descriptions of what the objects are going to contain, Attributes are the descriptions of the objects' properties, and Relationships are the connections we will be allowed to establish between objects. The first step is to add Entities to the model. Entities are created from the Add Entity button at the bottom of the editor (Figure 22-2, number 1). When we press this button, Xcode creates an entity with the generic name “Entity”.



Figure 22-3: New Entities

We can change the name of the newly created entity by double-clicking the name (Figure 22-3, number 1) or editing the Name field in the Data Model Inspector panel (Figure 22-3, number 2).

An entity defines the composition of objects that are going to be part of the Object Graph, so the next step is to declare the type of values those objects are going to manage. For this purpose, entities include Attributes. To add an attribute we have to select the entity and press the + button under the Attributes area (Figure 22-3, number 3) or press the Add Attribute button at the bottom of the editor (Figure 22-2, number 2). The attribute is added with the generic name “attribute” and the data type Undefined. Again, we can change the name of the attribute by double-clicking on it or editing the Name field in the Data Model Inspector panel. For our example, we called the entity *Books* and the first attribute *title* (Figure 22-4, number 1).



Figure 22-4: New Attributes

The Basics: The name of entities has to start with an upper-case letter and the names of attributes and relationships have to start with a lower-case letter.

Every attribute has to be associated with a data type for the objects to know what kind of values they can manage (Figure 22-4, number 2). Clicking on the attribute's type we can open a menu to select the right data type.

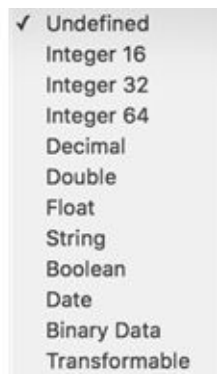


Figure 22-5: Attribute's data types

The Integer 16, 32, or 64 options are for **Int16**, **Int32**, and **Int64** values, **Decimal** is for **NSDecimalNumber** values, **Double** and **Float** are for **Double** and **Float** values, **String** is for **String** values, **Boolean** is for **Bool** values, **Date** is for **NSDate** values, **Binary Data** is for **NSData** values, and **Transformable** is for predefined and custom objects that adopt the **NSCoding** protocol and therefore include their own **NSCoder** methods to encode and decode the data, like the objects we have created in Chapter 21. The **Binary Data** and the **Transformable** options are frequently used to store images, but **Transformable** is more convenient because it takes care of archiving the **UIImage** object and turns it into an **NSData** value, while with the **Binary Data** option we have to do it ourselves.

Core Data was designed for Objective-C and therefore it works with objects and not primitive values or structures. All the numbers, including integers, floating-point, and Boolean values, are stored as **NSNumber** values, so we have to convert them to objects of this class or vice versa for processing.

An entity may contain as many attributes as our objects need. For example, we may add a few more attributes to complement the book's information.



Figure 22-6: Multiple Attributes

We could have also included another attribute for the author's name, but this is when we have to start thinking about the structure of the Object Graph and how we need the information to be stored. If we include a String type attribute for the author inside the Books entity, every time the user inserts a new book it will have to type the name of the author. This is error prone, time consuming, and when several books of the same author are available, it is impossible to make sure that all of them share the same exact name (one book could have the author's middle name and others don't, for example). Without the certainty of having the exact same name, we can never incorporate features in our app such as ordering the books by author or getting a list of books written by a particular author. Things get worse when, along with the name, we also decide to store other information about the author, like his or her age. A proper organization of this information demands the creation of separate objects and therefore we have to create new entities to represent them. Additional entities are added to the model the same way as the first one. Figure 22-7, next, shows our example with a new entity called *Authors* containing an attribute called *name*.



Figure 22-7: Multiple Entities

Entities are blueprints for objects, similar to what a class is, so we use them to define the characteristics of the objects we want to store. For instance, when we want to store a new book in our example, we create a new object based on the Books entity. That object will have four properties corresponding to the values of its title, year, cover, and thumbnail. The same happens when we want to store the information of an author. We create a new object based on the Authors entity and assign the name of the author to its **name** property. At the end, we will have two objects in the storage space, one for the book and another for the author. But if we want to retrieve these objects later, we need a way to

know what Books object is related to what Authors object. We have to connect these objects, so when we look for the Book's object we can also get the corresponding Author's object, otherwise we won't be able to show the name of the author along with the information of the book. To create this connection, the Core Data model includes Relationships.

Relationships are like properties in an object containing references to another object. They could have a reference to one object or a set of objects. For example, in the Books entity we can create a relationship that contains a reference to only one object of the Authors entity, because there could only be one author per book (for this example we are assuming that our app is storing books written only by one author). On the contrary, in the Authors entity we need to establish a relationship that contains references to multiple Books objects, because an author may have written several books, not just one. Core Data calls these relationships according to the number of objects they may reference. The names are To-One and To-Many and they are created pressing the + button in the Relationships area below the Attributes area. Figure 22-8 shows a relationship called *author* we have created for the Books entity of our example (we called it *author* because it is going to be used to retrieve the author of the book).

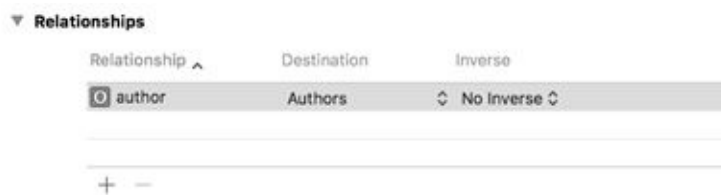


Figure 22-8: Relationship for the Books entity

A relationship only needs two values: its name (the name of the property) and the destination (the type of objects it is referencing), but it requires some parameters to be set. We have to tell the model if the relationship is going to be optional, define its type (To-One or To-Many), and determine what should happen to the destination object if the source object is deleted (the Delete Rule). All these options are available in the Data Model Inspector panel, as shown in Figure 22-9.



Figure 22-9: Relationship settings

By default, the relationship is set as Optional, which means that the source may be

connected to a destination object or not (a book can have an author or not), the Type of the relationship is set to To-One (a book can only have one author), and the Delete Rule is set to Nullify. The following are all the values available for this rule.

- **Deny** if there is at least one object at the destination, the source is not deleted (if there is an Authors object assigned to the Books object, the book is not deleted).
- **Nullify** the connection between objects are removed, but the objects at the destination are not deleted (if a Books object is deleted, the Authors object associated with that book loses the connection but is not deleted).
- **Cascade** the objects at the destination are deleted when the source is deleted (the Authors object is deleted if its book is deleted).
- **No Action** the object at the destinations are not deleted or modified (the connections are preserved, even when the source object doesn't exist anymore).

To find the right rule for a relationship we have to think in terms of the information we are manipulating. Is it right to delete the author if one of its books is deleted? In our case, the answer is simple. An author can have more than one book, so we can't delete the author when we delete a book because there could be other books that are connected to that same author. Therefore, the Nullify rule set by default is the right one for this relationship. But this could change when we create the opposite relationship, connecting the Authors entity to the Books entity. We need this second relationship to search for books that belong to an author. Figure 22-10 shows a relationship called *books* that we have created for the Authors entity.

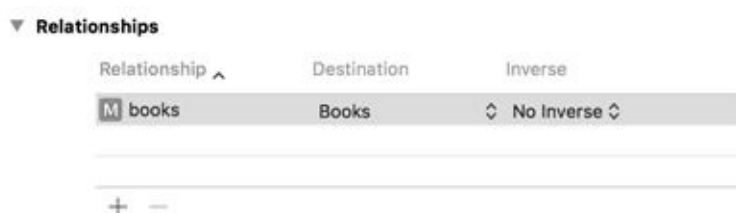


Figure 22-10: Relationship for the Authors entity

IMPORTANT: Relationships must always be bidirectional. If we set a relationship from entity A to entity B, we have to set the opposite relationship from entity B to entity A. Core Data offers another type of relationship called *Fetches Properties* to connect entities in only one direction. You can add a Fetched Property from the area below the Relationships area in the model's editor.

The new relationship added in Figure 22-10 is in the Authors entity, so every Authors object will have a property called `book` that we can use to retrieve the Books objects the author was assigned to. Because one author can have many books, the setting of this relationship is going to differ from the previous one. Now we have to set the Type of the relationship as To-Many (to many books) and modify the Delete Rule according to how we want our application to respond when an author is deleted. If we don't want to keep books that don't have an author assigned, we should select the Cascade option, so when an author is deleted all of his or her books are deleted too. But if we don't mind having books with no author around, then the option should be kept as Nullify.

The Basics: The Delete Rules are a way to ensure that the objects remaining in the Object Graph are those that our application and the user need. We can always set the rules as Nullify and take care of deleting all the objects that are not required anymore ourselves or to avoid deleting an object if we don't want to.

There is a third value for the relationship called *Inverse*. Once we set the relationships on both sides, it is highly recommended to set this value. It just tells the model what the name of the opposite relationship is. Core Data needs this to ensure the consistency of the Object Graph. Figure 22-11 shows the final setup for both relationships.



Figure 22-11: Inverse Relationships

Two relationships are simple to follow, but multiple relationships connecting several entities together can turn the model into an indecipherable mess. To help us identify every component of the model, Xcode offers an additional visualization style that displays the entities as boxes and the relationships as arrows connecting the boxes. The option, called *Editor Style*, is at the bottom of the screen (Figure 22-2, number 3). Figure 22-12, next, shows what our model looks like when we switch to this style (Notice that the To-Many relationship is represented by double arrows).

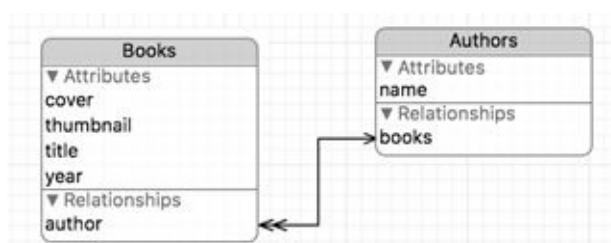


Figure 22-12: Graphic Style

Do It Yourself: Create a new Single View Application project. Open the File menu at the top of the screen, go to New and select the File... option to create a new file. Click on Core Data and select the option Data Model (Figure 22-1). Give the file the name you want but keep the extension `xcdatamodeld`. Click on the file to open the editor (Figure 22-2). Press the Add Entity button to create two new entities and give them the names Authors and Books. Create the attributes for these entities as illustrated in Figures 22-6 and 22-7. Create the relationships for every entity as shown in Figure 22-11. Set the books relationship as To-Many and keep the rest of the values by default. Click on the Editor Style button to see a graphic representation of the model (Figure 22-12).

Core Data Stack

The creation of the model is just the first step. Once we have all the entities, along with their attributes and relationships set up, we have to initialize Core Data for our application. Core Data is created from a group of objects that manage all the processes, from the organization of the Object Graph to the storage of the graph on file. There is an object that manages the model, an object that stores the data on file, and an object that intermediates between this persistent store and our own code. The scheme is called *stack*. Figure 22-13 illustrates a common Core Data stack.

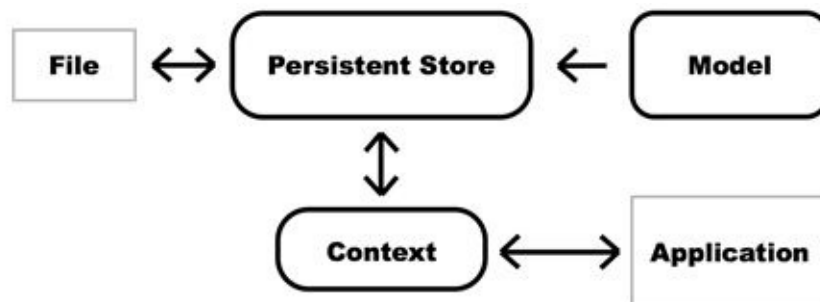


Figure 22-13: Core Data stack

The Persistent Store processes the Object Graph and stores it in the file. The Context asks the Persistent Store to read or add new objects to the graph, and our code interacts with the Context to manage the objects and their values.

The object that manages the model is created from the `NSManagedObjectContext` class. The class offers the following initializer to create this object based on the model designed in Xcode's editor.

NSManagedObjectModel(contentsOfURL: URL)—This initializer creates an **NSManagedObjectModel** object to manage the model. The **URL** attribute is the URL pointing to the model’s file in the bundle.

The persistent store is created from the **NSPersistentStore** class. Core Data allow us to create more than one Persistent Store per application. We can have a Persistent Store that stores the data in a file and also a Persistent Store that stores the data in memory, for example. The most common scenario only includes one Persistent Store associated with a file, but in case we need more Core Data works with a Persistent Store Coordinator created from the **NSPersistentStoreCoordinator** class. This coordinator not only manages the available Persistent Stores but also makes sure that the Object Graph is consistent with the model’s definitions. The class includes the following initializer and method to create the coordinator and add a Persistent Store to it.

NSPersistentStoreCoordinator(managedObjectModel: model)—This initializer creates an **NSPersistentStoreCoordinator** object and associates a model to it. The **model** attribute is a reference to the **NSManagedObjectModel** that manages the application’s model.

addPersistentStoreWithType(type, configuration: configuration, URL: URL, options: options)—This method creates an **NSPersistentStore** object and adds it to the Persistent Store Coordinator. The **type** attribute is a constant that represents the format that the Persistent Store is going to use to store the data on file. The possible values are **NSSQLiteStoreType** (stores the objects in an SQLite database), **NSXMLStoreType** (stores the objects in XML format), **NSBinaryStoreType** (stores the objects as binary data) and **NSInMemoryStoreType** (stores the objects in memory). The **configuration** attribute is the configuration of the model we want to use with the Persistent Store, the **URL** attribute is the URL pointing to the location of the storage file, and the **options** attribute is a dictionary with configuration options for the Persistent Store.

The context used by our app to interact with the Persistent Store is an object of the **NSManagedObjectContext** class. The class includes the following initializer.

NSManagedObjectContext(concurrencyType: type)—This initializer creates an **NSManagedObjectContext** object with the configuration set by the **type** attribute. The attribute is an enumeration of type **NSManagedObjectContextConcurrencyType** that determines the thread in which the operations will be performed. The possible values are **ConfinementConcurrencyType**, **PrivateQueueConcurrencyType**, and **MainQueueConcurrencyType**.

This class also includes the following properties and methods to manage the context and the objects in the Persistent Store.

hasChanges—This property returns a Boolean value that determines if the context contains changes that haven't been stored in the Persistent Store yet.

save()—This method saves the changes in the context to the Persistent Store.

reset()—This method resets the context to a basic state. All the `NSManagedObject` objects in the context are ignored.

deleteObject(object)—This method deletes an `NSManagedObject` object from the Persistent Store.

executeFetchRequest(request)—This method performs a fetch request on the Persistent Store to get the objects the request is asking for. The **request** attribute is an object of type `NSFetchRequest`, a Core Data class that allows us to configure a request.

countForFetchRequest(request, error: error)—This method returns the number of objects found in the Persistent Store for the request. The **request** attribute is an object of type `NSFetchRequest`, a Core Data class that allows us to configure a request. The **error** attribute is an optional pointer to an `NSError` value (usually specified as `nil`).

mergeChangesFromContextDidSaveNotification(notification)—This method merges the content of the Persistent Store with the values in the context. It was specifically designed to respond to changes in iCloud (we will study iCloud in Chapter 23).

There are different ways to create the Core Data stack for our application and also different ways to interact with it. The currently recommended approach is to create a custom class that configures all the components and then instantiate the object from the app's delegate. In the following example we create a class called `CoreDataStack` and configure the stack during initialization.

Listing 22-1: Creating the Core Data stack

```
import Foundation
import CoreData

class CoreDataStack: NSObject {
    var context: NSManagedObjectContext!

    override init() {
        super.init()
```

```

let bundle = NSBundle.mainBundle()
let modelURL = bundle.URLForResource("books", withExtension: "momd")
let manager = NSFileManager.defaultManager()
let urls = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
let url = urls.last!
let localURL = url.URLByAppendingPathComponent("store.sqlite")

let model = NSManagedObjectModel(contentsOfURL: modelURL!)
let coordinator = NSPersistentStoreCoordinator(managedObjectModel: model!)
try! coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: localURL,
options: nil)
context = NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)
context.persistentStoreCoordinator = coordinator
}
}

```

The class of Listing 22-1 defines a property called `context` to store a reference to the `NSManagedObjectContext` object required by our application and then initializes the stack. The first step is to get the URLs of the files the stack needs. We have to load the model's file from the bundle and create a new file to store the data. The URL of the model is obtained with the `URLForResource()` method of the bundle's object. The file's name is the name we have declared for the model's file when created, but the extension is different. The file created in Xcode with the `xcdatamodeld` extension includes information on how to present and organize the model on the screen. When the file is compiled, all this information is removed and the new file is assigned the extension `momd`. This is why in Listing 22-1 we get the URL for the file `books.momd` instead of `books.xcdatamodeld`. The file for the Persistent Store needs an extension according to the type of Persistent Store we are going to create. In this example, we decided to use SQLite (recommended in most cases), so the URL for the file has to contain the extension `sqlite`. We called it `store.sqlite`. With these two URLs ready, we can finally build the Core Data stack.

The first object we need is the `NSManagedObjectModel` object. We call the class initializer with the URL for the model's file and store the object in the `model` constant. Next, we use this reference to create the Persistent Store Coordinator (`NSPersistentStoreCoordinator`). To add the Persistent Store to the coordinator we have to specify what type of store we want to create and how it is going to be configured. The first attribute is the type. For this example we use an SQLite database (`NSSQLiteStoreType`). The second parameter is the configuration of the model. By default, Xcode creates a configuration called *Default*, but we can add more and prepare a version of the model for different Persistent Stores. To use the configuration by default (most common scenario) we can declare this value as `nil`. The last two parameters are the URL of the file where the data is going to be stored and a dictionary with configuration options. An initial Core Data stack usually doesn't need to specify any option and this value can be declared as `nil`, but there is an extensive list of constants available to define these options (we will study some of them later in this

chapter).

IMPORTANT: The `addPersistentStoreWithType()` method can fail and throw an error. How to handle this error depends on your application. You can use a `do catch` statement to report to the user that the Persistent Store is not available at this time and try to add it later, or you can use the `try!` keyword and let the app crash (as we did in Listing 22-1). When the app crashes and it is launched again, the system tries to build the Core Data stack one more time, so this may be a good alternative for small applications. Always remember that you have to provide the best experience for the users and crashing the app does not always achieve that goal.

The last object required is the `NSManagedObjectContext` object that our code is going to use to interact with the Persistent Store. There is only one parameter we have to specify to create this object called *concurrency type*. The concurrency type determines in which thread the operations are going to be performed. Assigning a time-consuming task to another thread lets the user work with the app while the process is being completed (see Chapter 20). Core Data only needs to work on a secondary thread when it has to perform processes that take too long to complete, but usually this is not the case, so we declare it as `MainQueueConcurrencyType` (main thread).

The Basics: Core Data was designed for Objective-C applications and require elements from other Objective-C frameworks to work. If you create the Core Data stack from a custom class, you have to make the class inherit from the Foundation's elemental class `NSObject` to make sure that everything works as expected.

The `CoreDataStack` class we have just defined creates the stack and stores a reference to the `NSManagedObjectContext` object in the `context` property. Because every view controller needs this reference to read or store objects in the Persistent Store, we have to create an instance of the `CoreDataStack` class in a place that is accessible for the rest of the code. Apple recommends to do it in the app's delegate, because this way we make sure that everything is created as soon as the app is launched. The following example initializes the stack and stores a reference in a property called `stack`.

Listing 22-2: Initializing the Core Data stack in the app's delegate

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?
    var stack: CoreDataStack!
```

```

func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject:
AnyObject]?) -> Bool {
    stack = CoreDataStack()
    return true
}
}

```

Now we can read the **stack** property of the app's delegate from our view controllers and get a reference to the **NSManagedObjectContext** object to add, fetch, and remove objects from the store. The following code illustrates how to get this reference from a view controller (notice that we assign the reference to another property called **context** to have a convenient reference we can use in other methods inside the class).

Listing 22-3: Accessing the context from a view controller

```

import UIKit
import CoreData

class ViewController: UIViewController {
    var context: NSManagedObjectContext!
    override func viewDidLoad() {
        super.viewDidLoad()

        let app = UIApplication.sharedApplication()
        let appDelegate = app.delegate as! AppDelegate
        context = appDelegate.stack.context
    }
}

```

Do It Yourself: Create a new Swift file called CoreDataStack.swift inside the project initiated before for the model and copy the code of Listing 22-1 in it. Update your **AppDelegate** class with the code of Listing 22-2. At this moment the app doesn't do anything other than creating the stack.

IMPORTANT: When the app is launched for the first time, the Core Data stack creates a Persistent Store based on the current model. If you later modify the model, adding new entities or deleting attributes, for example, the app will crash. To test the app with a new model you have to uninstall it first to erase the old Persistent Store. If you already shipped your app to the App Store or don't want to lose the information inserted for testing, you have to migrate from the old model to the new one. We will study migration later in this chapter.

NSManagedObject

Core Data doesn't store our custom objects, it defines a class called `NSManagedObject` to create generic container objects that we can associate with an Entity in the model and then use it to store the values we want. To create an `NSManagedObject` object the class includes the following initializer.

`NSManagedObject(entity: entity, insertIntoManagedObjectContext: context)`—This initializer creates an `NSManagedObject` object. The **`entity`** attribute is an object of type `NSEntityDescription` that represents the entity we want to associate to the object and the **`context`** attribute is a reference to the `NSManagedObjectContext` object from the stack.

An `NSManagedObject` object has to be associated to one of the model's entities for Core Data to know the names and type of values the object can manage. To represent entities, Core Data includes the `NSEntityDescription` class. The class defines two convenient type methods to work with entities and `NSManagedObject` objects.

`entityForName(entity, inManagedObjectContext: context)`—This type method returns a reference to the `NSEntityDescription` object that represents the entity specified by the **`entity`** attribute. The **`entity`** attribute is a string with the name of the entity we want to retrieve and the **`context`** attribute is a reference to the `NSManagedObjectContext` object from the stack.

`insertNewObjectForEntityForName(entity, inManagedObjectContext: context)`—This type method creates a new `NSManagedObject` associated with the entity defined by the **`entity`** attribute and inserts it in the context. The **`entity`** attribute is a string with the name of the entity we want to associate to the object, and the **`context`** attribute is a reference to the `NSManagedObjectContext` object from the stack.

Once the `NSManagedObject` object is created, we can assign and retrieve values from its properties using the `setValue()` and `valueForKey()` methods defined in the `NSKeyValueCoding` protocol (see Key/Value Observing in Chapter 18).

Managing Objects

All the interaction between our code and the Persistent Store is done through the context. When we want to access the objects already stored, add new ones, remove them or modify any of their values, we have to do it in the context and then move those changes from the context to the Persistent Store. To illustrate how this process works, we are going to create a simple example similar to the one we used before for Archiving in Chapter 21. Figure 22-14 shows the modifications we have to introduce to the interface for this example.

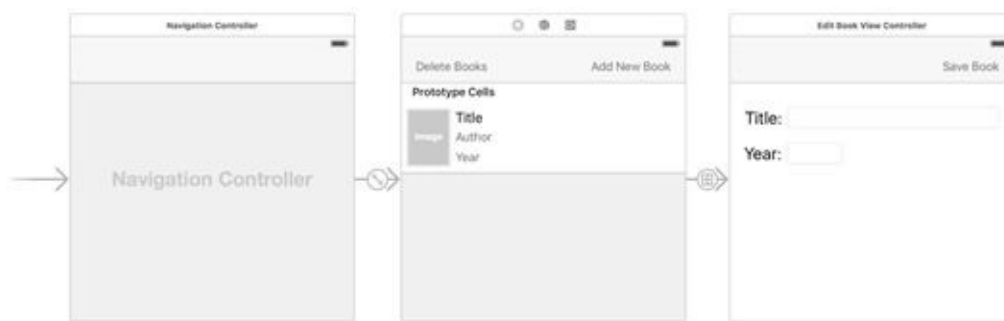


Figure 22-14: Interface to list and add books

We have replaced the initial view of the Single View Application project created before by a Table View Controller and a single view, both embedded in a Navigation Controller. The Navigation Bar in the Table View Controller includes two bar buttons: Delete Books to delete the books and Add New Book to add new ones. The Add New Book button is connected to the second view with a Show Segue. This view includes two Text Fields to let the user insert the title and year of a new book and a bar button called Save Book to save it. To show the values in the table we have designed a custom cell with four elements: an image view for the cover of the book and three labels for the title, author, and year. The following is the `UITableViewCell` subclass we need for this cell.

Listing 22-4: *Creating the Outlets for the elements in the cell*

```
import UIKit

class BooksCell: UITableViewCell {
    @IBOutlet weak var bookCover: UIImageView!
    @IBOutlet weak var bookTitle: UILabel!
    @IBOutlet weak var bookAuthor: UILabel!
    @IBOutlet weak var bookYear: UILabel!
}
```

Do It Yourself: Delete the initial view of the project created in the previous section of this chapter. Add a Table View Controller to the Storyboard and embed it in a Navigation Controller. Add two bar buttons to the Navigation Bar called Delete

Books and Add New Book. Add a new view and connect the Add New Book button to this view with a Show Segue. Add a Navigation Item to the Navigation Bar of the second view and a bar button called Save Book. Add two labels and two Text Fields to the second view, as illustrated in Figure 22-14. Select the prototype cell in the Table View, assign the value 100 to its height from the Size Inspector panel, and give it the identifier “booksCell” from the Attributes Inspector panel. Add to the cell an image view with Width and Height constraints of 60 and 80, respectively, and the labels “Title”, “Author”, and “Year”. Create a `UITableViewCell` subclass called `BooksCell` and assign it to the cell. Complete the class with the code of Listing 22-4. Connect the elements in the cell to their respective Outlets.

The purpose of this application is to show the list of objects already stored in the Persistent Store and create new ones. To create a new `NSManagedObject` object we need to get a reference to the entity the object is going to be associated with, create the object, and insert it in the context. There are two alternatives available to achieve this. We can obtain a reference to the entity from the `entityForName()` method provided by the `NSEntityDescription` class and then create the object and insert it into the context with the `NSManagedObjectContext` initializer, or we can use a convenient type method also included in the `NSEntityDescription` class called `insertNewObjectForEntityForName()`. This method creates an object for an entity and puts it in the context, all in just one single line of code. The process is introduced next in the `EditBookViewController` class we have created to control the second view.

Listing 22-5: Adding new objects to the Persistent Store

```
import UIKit
import CoreData

class EditBookViewController: UIViewController {
    @IBOutlet weak var bookTitle: UITextField!
    @IBOutlet weak var bookYear: UITextField!
    var context: NSManagedObjectContext!

    override func viewDidLoad() {
        super.viewDidLoad()

        bookTitle.becomeFirstResponder()
        let app = UIApplication.sharedApplication()
        let appDelegate = app.delegate as! AppDelegate
        context = appDelegate.stack.context
    }

    @IBAction func saveBook(sender: UIBarButtonItem) {
        let year = Int(bookYear.text!)
        let title = bookTitle.text!.stringByTrimmingCharactersInSet(.whitespaceAndNewlineCharacterSet())
        if title != "" && year != nil {
```

```

let newBook = NSEntityDescription.insertNewObjectForEntityForName("Books", inManagedObjectContext:
context)
newBook.setValue(title, forKey: "title")
newBook.setValue(NSNumber(integer: year!), forKey: "year")
newBook.setValue(nil, forKey: "author")

let cover = UIImage(named: "nocover")
newBook.setValue(cover, forKey: "cover")
let thumbnail = UIImage(named: "nothumbnail")
newBook.setValue(thumbnail, forKey: "thumbnail")

do {
try context.save()
} catch {
print("Error")
}
navigationController?.popViewControllerAnimated(true)
}
}

```

The first thing we always have to do in the view controllers that want to work with Core Data is to get a reference to the context and assign it to a property, so every method in the class have access to it. In the view controller of Listing 22-5 we call this property **context**. When the user presses the Save Book button, the **saveBook()** method gets the values form the Text Fields and uses this context reference to create and store a new object into the Persistent Store. The process begins with the **insertNewObjectForEntityForName()** method. This method creates a new **NSManagedObject** associated with the Books entity and inserts it into the context. After we get this object, we have to assign the values we want for its properties. An object associated with the Books entity contains four properties: **title**, **year**, **cover**, and **thumbnail**. Using the **setValue()** method on the object we assign the value of the first Text Field to the **title** property, the value of the second Text Field to the **year** property, the images **nocover.png** and **nothumbnail.png** to the **cover** and **thumbnail** properties, respectively (the images are available on our website), and the **nil** value to the **author** property (we still don't have an Authors object to associate with this book).

IMPORTANT: Core Data was designed for Objective-C and can't work with Swift primitive values. When we want to assign a Swift numeric type to a property of an **NSManagedObject** object, we have to convert the value into an **NSNumber** object before assigning it to the property, as we did in the example of Listing 22-5 (**NSNumber(integer: year!)**).

The `insertNewObjectForEntityForName()` method inserts the new object into the context, but this change is not permanent. If we close the app after the values are assigned to the properties, the object is lost. To persist the changes we have to save the context with the `save()` method. This method takes the information in the context and modifies the Persistent Store with it, so everything is stored on file.

Do It Yourself: Create a subclass of the `UIViewController` class called `EditBookViewController` and assign it to the second view. Connect the first Text Field with an Outlet called `bookTitle` and the second Text Field with an Outlet called `bookYear`. Connect the Save Book button in the Navigation Bar with an Action called `saveBook()`. Complete the class with the code of Listing 22-5.

IMPORTANT: Because the `save()` method can throw errors, we call it inside a `do catch` statement. In this example we just print a message if something goes wrong, but you should consider warning users with an Alert View and keeping the information safe to give them the chance to try again.

The process to get the objects back from the Persistent Store is the opposite. Instead of moving the changes in the context to the Persistent Store we have to get the objects from the Persistent Store and put them in the context. Once the objects are in the context we can read their properties, modify their values, or delete them. Core Data includes the `NSFetchRequest` class to configure a request for objects. The class includes the following initializer to create a request.

`NSFetchRequest(entityName: entity)`—This initializer creates an `NSFetchRequest` object that requests objects of the entity specified by the `entity` attribute. The attribute is a string with the name of the entity.

There are also a few properties defined in the `NSFetchRequest` class to configure the request.

`predicate`—This property sets or returns the predicate used to filter the objects fetched by the request. It is a value of type `NSPredicate`, a Core Data class used to establish logical conditions that describe objects in the Persistent Store.

`sortDescriptors`—This property sets or returns an array of sort descriptors that determine how the objects obtained by the request should be ordered. It is an array of values of type `NSSortDescriptor`, a Core Data class used to sort the objects according to the value of a property in ascending or descending order.

resultType—This property sets or returns a value that determines the type of data returned by the request. It is an enumeration of type `NSFetchRequestResultType` with the values `ManagedObjectResultType` (it returns values of type `NSManagedObject`), `ManagedObjectIDResultType` (it returns the identification values of the `NSManagedObject` objects instead of the objects themselves), `DictionaryResultType` (it returns a dictionary with the values of the properties), and `CountResultType` (it returns an integer value with the total of objects found).

propertiesToFetch—This property sets or returns an array of values that determine the properties we want to get (by default, all the properties of the `NSManagedObject` objects are returned). The properties of an entity (attributes) are represented by objects of the `NSPropertyDescription` class, or subclasses of it.

propertiesToGroupBy—This property sets or returns an array of `NSPropertyDescription` objects that determine the properties that are going to be used to group the `NSManagedObject` objects together. When objects are grouped together, only one value is returned for each group, so this is usually used with operations performed over the values, such as counting the number of books with the same title, for example (the property only works when the `resultType` property is set to `DictionaryResultType`). For an example, see Chapter 23.

By default, a request is configured to return an array of `NSManagedObject` objects with all their properties (it is actually an array of `AnyObject` objects that we can cast as `NSManagedObject` objects), and this is exactly the configuration we need in our example to get the list of all the books available and show their values in the table. The following is the view controller for the Table View. The code performs a request, stores the objects in an array, and then shows their values in the table.

Listing 22-6: Fetching values from the Persistent Store

```
import UIKit
import CoreData

class BooksViewController: UITableViewController {
    var context: NSManagedObjectContext!
    var listOfBooks: [NSManagedObject] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.rowHeight = 100
        let app = UIApplication.sharedApplication()
        let appDelegate = app.delegate as! AppDelegate
        context = appDelegate.stack.context
    }

    override func viewWillAppear(animated: Bool) {
```

```

super.viewWillAppear(animated)

let request = NSFetchRequest(entityName: "Books")
do {
    listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
} catch {
    print("Error")
}
tableView.reloadData()
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return listOfBooks.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("booksCell", forIndexPath: indexPath) as! BooksCell

    let book = listOfBooks[indexPath.row]
    cell.bookTitle.text = book.valueForKey("title") as? String
    cell.bookCover.image = book.valueForKey("thumbnail") as? UIImage
    let year = book.valueForKey("year") as? Int
    cell.bookYear.text = "Year: \(year!)"
    let author = book.valueForKey("author") as? NSManagedObject
    if author != nil {
        cell.bookAuthor.text = author!.valueForKey("name") as? String
    } else {
        cell.bookAuthor.text = "Not Defined"
    }
    return cell
}
}

```

There are different places from where we can perform a request, depending on the requirements of our application. In this example we do it from the `viewWillAppear()` method, so the objects are retrieved and the table is updated every time the user opens the view. To make a request, we first have to create an instance of the `NSFetchRequest` class associated with an entity and then call the `executeFetchRequest()` method in the context. The method returns an array of objects of type `AnyObject` that we have to cast to the `NSManagedObject` class and assign to a property to be able to access the values from other methods. For this example we called this property `listOfBooks`.

The Basics: You can concatenated `valueForKey()` methods to read the properties of a

relationship. For example, if you want to read the name of the author from the author relationship, as in the code of Listing 22-6, you can do it with just one line of code: `book valueForKey("author")?.valueForKey("name") as? String`.

Once we get the objects in an array, the rest of the process is similar to any other example we have seen before, with the exception that this time we are working with `NSManagedObject` objects and therefore we have to use the `valueForKey()` method to read their values. The method always returns an object of type `AnyObject`, so we have to cast the values to the right type. In the case of the `author` property, because we are not assigning authors to the books yet, we compare it to `nil` and then show the name of the author or a message indicating that the author was not defined. The result is illustrated in Figure 22-15.

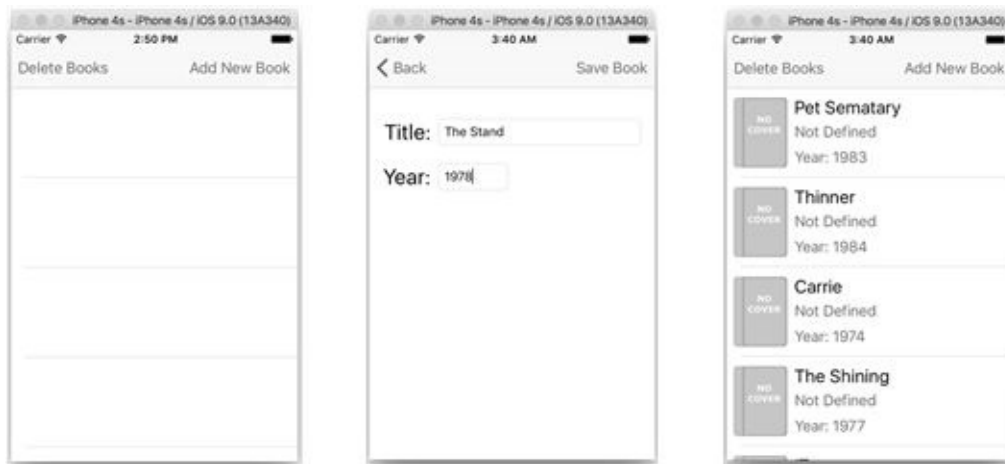


Figure 22-15: Adding books to a Persistent Store

Do It Yourself: Create a `UITableViewController` subclass called `BooksViewController` and assign it to the Table View Controller. Complete the class with the code of Listing 22-6. Download the images `nocover.png` and `nothumbnail.png` from our website and add them to the Assets Catalog. Run the application. The first screen will show an empty Table View (Figure 22-15, left). Press the Add New Book button and insert the values for the book (Figure 22-15, center). The table should look like the right picture of Figure 22-15.

Authors objects, like Books objects, are generated and stored as `NSManagedObject` objects, except they have to be associated to the Authors entity. This demands our application to provide new views where the user can select and add new objects. For our example we have decided to expand our interface with another Table View Controller to list the authors available and another single view to insert more. To provide access to these new views, we have added a button called Select Author to the `EditBookViewController`'s view, along with a label to show the name of the author.

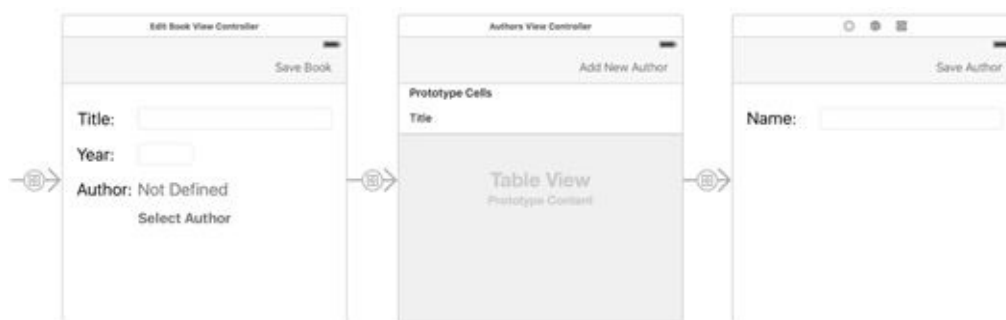


Figure 22-16: *Interface to list and add authors*

When the user presses the Add New Book button in the initial view (Figure 22-14, center), the next view now shows three input options: a Text Field to insert the title of the book, a Text Field to insert the year, and the Select Author button to select the author (Figure 22-16, left). This button is connected with a Show Segue to a Table View Controller that lists all the authors available (Figure 22-16, center). In addition, this Table View Controller includes a bar button called Add New Author to open a view that includes a Text Field to insert the name of a new author (Figure 22-16, right).

Do It Yourself: Add two labels and a button called Select Author to the `EditBookViewController`'s view (Figure 22-16, left). Add a Table View Controller to the Storyboard and connect the Select Author button to this view with a Show Segue (Figure 22-16, center). Select the cell in the new Table View and change its style to Basic and its identifier to "authorsCell". Add a bar button to this view called Add New Author. Add a new view to the Storyboard and connect the Add New Author button to this view with a Show Segue (Figure 22-16, right). Add a label, an input field, and a bar button called Save Author to the last view. Your interface should look like the combination of Figures 22-14 and 22-16.

Every time an author is selected or created, we have to get its `NSManagedObject` object and send it back to the `EditBookViewController` class to assign it to the book. There are different ways to do it, but for this example we are going to use Unwind Segues (see Chapter 13, Figure 13-19). We need two Unwind Segues, one that is triggered when the user selects an author from the Table View Controller, and another that is triggered when the user inserts a new author. When the Segues are triggered, we have to get the `NSManagedObject` of the author and update the view with the value of its `name` property. To receive and process this information, we have to modify the `EditBookViewController` class to include the Action for the Unwind Segues and properties to store and manage the Authors object.

Listing 22-7: *Adding an author to the book*

```

import UIKit
import CoreData

class EditBookViewController: UIViewController {
    @IBOutlet weak var bookTitle: UITextField!
    @IBOutlet weak var bookYear: UITextField!
    @IBOutlet weak var authorName: UILabel!
    var context: NSManagedObjectContext!
    var selectedAuthor: NSManagedObject!

    override func viewDidLoad() {
        super.viewDidLoad()
        bookTitle.becomeFirstResponder()
        let app = UIApplication.sharedApplication()
        let appDelegate = app.delegate as! AppDelegate
        context = appDelegate.stack.context
    }

    @IBAction func saveBook(sender: UIBarButtonItem) {
        let year = Int(bookYear.text!)
        let title = bookTitle.text!.stringByTrimmingCharactersInSet(.whitespaceAndNewlineCharacterSet())
        if title != "" && year != nil {
            let newBook = NSEntityDescription.insertNewObjectForEntityForName("Books", inManagedObjectContext: context)
            newBook.setValue(title, forKey: "title")
            newBook.setValue(NSNumber(integer: year!), forKey: "year")
            newBook.setValue(selectedAuthor, forKey: "author")

            let cover = UIImage(named: "nocover")
            newBook.setValue(cover, forKey: "cover")
            let thumbnail = UIImage(named: "nothumbnail")
            newBook.setValue(thumbnail, forKey: "thumbnail")
            do {
                try context.save()
            } catch {
                print("Error")
            }
            navigationController?.popViewControllerAnimated(true)
        }
    }

    @IBAction func backAuthor(segue: UIStoryboardSegue) {
        if segue.identifier == "backFromList" {
            let controller = segue.sourceViewController as! AuthorsViewController
            selectedAuthor = controller.selectedAuthor
        }
    }
}

```

```

    authorName.text = selectedAuthor.valueForKey("name") as? String
  } else if segue.identifier == "backFromNew" {
    let controller = segue.sourceViewController as! EditAuthorViewController
    selectedAuthor = controller.selectedAuthor
    authorName.text = selectedAuthor.valueForKey("name") as? String
  }
}
}
}

```

The view controller of Listing 22-7 manages the view that lets the user insert new books. When the user selects or inserts a new author, the `backAuthor()` method is executed (due to the Unwind Segues we will create later), and the code gets the `NSManagedObject` that represents the author, assign it to the `selectedAuthor` property to let other methods have access to it, and updates the label on the screen with the value of its `name` property (this process is done twice, one for each Unwind Segue).

The process to list and create new Authors objects is exactly the same we used for books. The following is the view controller for the Table View Controller added to the interface in Figure 22-16. We called it `AuthorsViewController`.

Listing 22-8: Listing authors

```

import UIKit
import CoreData

class AuthorsViewController: UITableViewController {
    var context: NSManagedObjectContext!
    var listOfAuthors: [NSManagedObject] = []
    var selectedAuthor: NSManagedObject!

    override func viewDidLoad() {
        super.viewDidLoad()
        let app = UIApplication.sharedApplication()
        let appDelegate = app.delegate as! AppDelegate
        context = appDelegate.stack.context
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        let request = NSFetchRequest(entityName: "Authors")
        do {
            listOfAuthors = try context.executeFetchRequest(request) as! [NSManagedObject]
        } catch {
            print("Error")
        }
    }
}

```

```

    }
    tableView.reloadData()
}
override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return listOfAuthors.count
}
override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("authorsCell", forIndexPath: indexPath)
    let author = listOfAuthors[indexPath.row]
    cell.textLabel?.text = author.valueForKey("name") as? String
    return cell
}
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    selectedAuthor = listOfAuthors[indexPath.row]
    performSegueWithIdentifier("backFromList", sender: self)
}
}

```

Other than changing the name for the array (**listOfAuthors**), the rest of the code, including the fetch request, is exactly the same we used to list books. This time we implement a cell with a Basic style, so all we have to do to show the authors in the Table View is to get the value of their **name** property and assign it to the label in the cell.

This code also implements the protocol method **tableView(tableView, didSelectRowAtIndexPath: indexPath)** to assign the selected author to the **selectedAuthor** property and trigger the Unwind Segue (the example assumes that we have created an Unwind Segue for the Table View Controller identified with the string "backFromList". Unwind Segues are explained in Chapter 13, Figures 13-16 and 13-19).

The view controller to add new authors is also very similar to the one we used to add new books. We have to take the text inserted by the user in the Text Field, create the new **NSManagedObject** object for the author, save the context, and finally trigger the Unwind Segue to move back to the **EditBookViewController**'s view (again, the example assumes that we have created an Unwind Segue for this view identified with the string "backFromNew").

Listing 22-9: Inserting new authors

```

import UIKit
import CoreData

class EditAuthorViewController: UIViewController {
    @IBOutlet weak var authorName: UITextField!

```



```

    var selectedAuthor: NSManagedObject!
var context: NSManagedObjectContext!
override func viewDidLoad() {
    super.viewDidLoad()
    authorName.becomeFirstResponder()
    let app = UIApplication.sharedApplication()
    let appDelegate = app.delegate as! AppDelegate
    context = appDelegate.stack.context
}
@IBAction func saveAuthor(sender: UIBarButtonItem) {
    let name = authorName.text!.stringByTrimmingCharactersInSet(.whitespaceAndNewlineCharacterSet())
    if name != "" {
        selectedAuthor = NSEntityDescription.insertNewObjectForEntityForName("Authors",
inManagedObjectContext: context)
        selectedAuthor.setValue(name, forKey: "name")
        do {
            try context.save()
            performSegueWithIdentifier("backFromNew", sender: self)
        } catch {
            print("Error")
        }
    }
}
}

```

With these additions our basic app is complete. Now authors can be assigned to new books. When we pressed the Select Author button, the app opens a Table View with all the authors we have available (Figure 22-17, center). If there are no authors yet or the one we want is not on the list, we can press the Add New Author button and insert a new one (Figure 22-17, right). Every time we select an author from the list or insert a new one, the app goes back to the view with the book's information and shows the name of the selected author on the screen (Figure 22-17, left). The `NSManagedObject` object that represents the author is assigned to the book's `author` property and therefore the name of the author is now shown in the list of books.

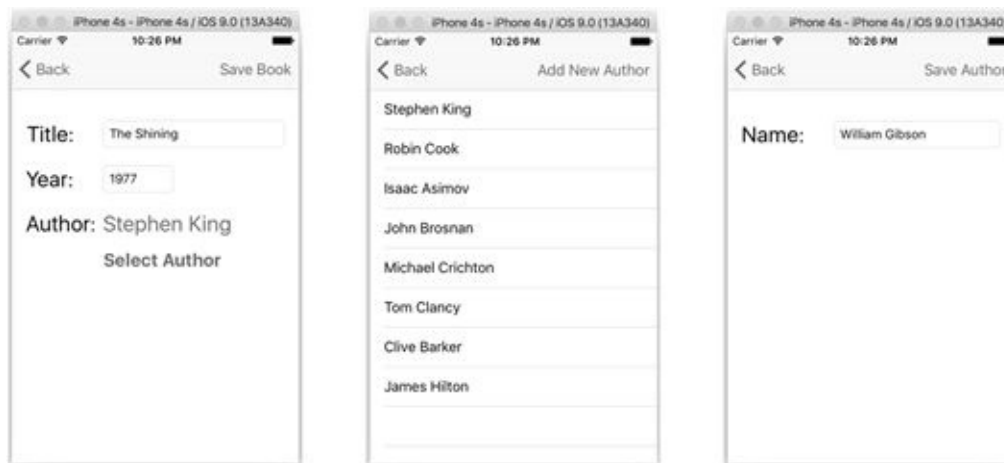


Figure 22-17: *Assigning an author to a book*

Do It Yourself: Update the `EditBookViewController` class with the code of Listing 22-7. Connect the label used to show the name of the author to the Outlet `authorName` in this class. Create a `UITableViewController` subclass called `AuthorsViewController` and assign it to the new Table View Controller (Figure 22-16, center). Complete the class with the code of Listing 22-8. Create an Unwind Segue for this view with the identifier “backFromList” (see Figure 13-19). Create a `UIViewController` subclass called `EditAuthorViewController` and assign it to the last view (Figure 22-16, right). Connect the Text Field in this view with an Outlet called `authorName` and complete the `EditAuthorViewController` class with the code of Listing 22-9. Create an Unwind Segue for this view with the identifier “backFromNew” (see Figure 13-19). Run the application and add a new book. Press the Select Author button and insert a new author. Save the book. You should see the list of books with their respective authors in the initial view.

Counting Objects

The `NSManagedObjectContext` class includes the `countForFetchRequest()` method we can use to count the number of objects associated with an entity. The method returns an integer with the number of objects we would get if we call the `executeFetchRequest()` method with the same request. The method doesn’t fetch the objects so we can call it without being afraid of using too much memory. For example, we can use it in the `EditBookViewController` class to get the number of authors already available in the Persistent Store.

Listing 22-10: Counting the authors available

```
override func viewDidLoad() {
    super.viewDidLoad()
```

```

bookTitle.becomeFirstResponder()
let app = UIApplication.sharedApplication()
let appDelegate = app.delegate as! AppDelegate
context = appDelegate.stack.context

let request = NSFetchRequest(entityName: "Authors")
let total = context.countForFetchRequest(request, error: nil)
print(total)
}

```

Do It Yourself: Replace the `viewDidLoad()` method of Listing 22-7 by the method of Listing 22-10. In this example we just print the result in the console, but you could add a new label to the view managed by the `EditBookViewController` class and show the value to the user.

Another alternative to count the number of `NSManagedObject` objects is to configure the request to count the objects and return an integer with the total. We can do this by assigning the value `CountResultType` to the `resultType` property.

Listing 22-11: Counting the authors available with CountResultType

```

override func viewDidLoad() {
    super.viewDidLoad()
    bookTitle.becomeFirstResponder()
    let app = UIApplication.sharedApplication()
    let appDelegate = app.delegate as! AppDelegate
    context = appDelegate.stack.context

    let request = NSFetchRequest(entityName: "Authors")
    request.resultType = .CountResultType
    do {
        let total = try context.executeFetchRequest(request) as! [Int]
        print(total.first!)
    } catch {
        print("Error")
    }
}

```

The request returns an array of integers with only one element containing the total amount of objects found. In Listing 22-11 we cast the results as an array of `Int` values and print the value of the first element in the console.

Do It Yourself: Replace the `viewDidLoad()` method of Listing 22-7 by the method of Listing 22-11. Run the application. You should see the number of Authors objects available printed in the console.

If what we want is to get the number of objects associated to a To-Many relationship, we can use the `valueForKey()` method. The difference between normal properties and those that represent relationships is that instead of an `AnyObject` value, the latter returns a Set of `AnyObject` values. To cast this value to a Set of `NSManagedObject` objects we have to use the Set's data type declaration `Set<NSManagedObject>`. For example, we can count the number of books of every author and show it along with the name in the table.

Listing 22-12: Counting the books of each author

```
override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {

    let cell = tableView.dequeueReusableCellWithIdentifier("authorsCell", forIndexPath: indexPath)

    let author = listOfAuthors[indexPath.row]
    let name = author.valueForKey("name") as? String
    let totalBooks = author.valueForKey("books") as! Set<NSManagedObject>
    cell.textLabel?.text = "\(name!) (\(totalBooks.count))"
    return cell
}
```

The method of Listing 22-12 replaces the same method in the `AuthorsViewController` class of Listing 22-8. Instead of assigning the name of the author directly to the cell's label, we get the books associated with each author, assign it to the `totalBooks` constant, and then create a string with the name plus the number of books available.

The Basics: The `NSKeyValueCoding` protocol defines other methods to work with keys. For example, the `valueForKeyPath()` method allows us to get objects in a deep relationship hierarchy using key paths (keys concatenated by dot notation). The topic is beyond the purpose of this book. For more information, visit our website and follow the links for this chapter.

Predicates

The requests performed in previous examples are getting all the objects associated to a

particular entity and the values of all their properties. Core Data takes advantage of a class called **NSPredicate** to define filters that allow us to fetch only the objects that comply to certain rules. For example, we could get only the books that were published in the year 1983. The class defines the following initializer to create a predicate with all the conditions we need.

NSPredicate(format: *format*, arguments)—This initializer creates an **NSPredicate** object with the conditions set by the **format** attribute. The **arguments** attribute is an optional list of values separated by comma that replace placeholders in the string assigned to the **format** attribute.

To filter values in a request, we have to create the **NSPredicate** object and assign it to the request's **predicate** property. The following example assigns an **NSPredicate** object to the **predicate** property of the request in the **BooksViewController** class to search for the books that has the value 1983 assigned to their **year** property.

Listing 22-13: Filtering books by year

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    let request = NSFetchRequest(entityName: "Books")
    request.predicate = NSPredicate(format: "year = 1983")
    do {
        listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
    } catch {
        print("Error")
    }
    tableView.reloadData()
}
```

If we are trying to search for a value in a relationship, we can concatenate the properties with dot notation in a construction called *key path*. For example, the following request searches for books written by Stephen King.

Listing 22-14: Filtering books by author

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    let request = NSFetchRequest(entityName: "Books")
    request.predicate = NSPredicate(format: "author.name = 'Stephen King'")
    do {
```

```

        listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
    } catch {
        print("Error")
    }
    tableView.reloadData()
}

```

The name of the author was inserted inside the string using single quotes, but we do not always know beforehand what value we have to search for. To incorporate the value of a variable inside the formatted string we can use placeholders. Placing the characters %@ inside the string, for example, indicates to the initializer that the value in the **arguments** attribute goes in that place.

***Listing 22-15:** Creating filters with placeholders*

```

override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    let search = "Stephen King"
    let request = NSFetchRequest(entityName: "Books")
    request.predicate = NSPredicate(format: "author.name = %@", search)
    do {
        listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
    } catch {
        print("Error")
    }
    tableView.reloadData()
}

```

We may include all the placeholders and arguments we need. The placeholders are replaced by the arguments one by one, in consecutive order, as with formatted strings (see String Structures in Chapter 3). For example, we can search for books of a particular author and year (notice that Core Data works with Objective-C objects and therefore we have to convert the Swift integer for the year into an **NSNumber** object).

***Listing 22-16:** Creating filters with multiple values*

```

override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    let search = "Stephen King"
    let year = 1983
    let request = NSFetchRequest(entityName: "Books")

```

```

request.predicate = NSPredicate(format: "author.name = %@ && year = %@", search, NSNumber(integer:
year))
do {
    listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
} catch {
    print("Error")
}
tableView.reloadData()
}

```

Do It Yourself: Replace the `viewWillAppear()` method in the `BooksViewController` class of Listing 22-6 by any of the methods presented above to see how predicates work. Add books with different authors and years to test the filters.

IMPORTANT: The placeholder `%@` is replaced by the value specified in the attributes between quotes. If you need to add the value to the predicate without the quotes, you have to use the placeholder `%K` instead. We will see an example in Chapter 23.

Predicates use the same comparison and logical operators offered by Swift. For example, we can compare values with the operators `==`, `!=`, `>`, `<`, `>=` and `<=`, and we can also concatenate conditions with the characters `&&` (or the word **AND**), `||` (or the word **OR**) and `!` (or the word **NOT**). Predicates also include keywords for a more meticulous search. The following are the most frequently used.

BEGINSWITH—The condition determined by this keyword is true when the expression on the left begins with the expression on the right.

CONTAINS—The condition determined by this keyword is true when the expression on the left contains the expression on the right.

ENDSWITH—The condition determined by this keyword is true when the expression on the left ends with the expression on the right.

LIKE—The condition determined by this keyword is true when the expression on the left is equal to the expression on the right.

IN—The condition determined by this keyword is true when the expression on the left is equal to any of the values included in the expression on the right. The values are provided as an array between parentheses.

These keywords may be accompanied by the characters **c** or **d** between square brackets to specify a case and diacritic insensitive search. For example, we may search for authors with a name beginning with the word “stephen”, without considering upper- or lower-case letters.

Listing 22-17: Filtering values with predicate keywords

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    let search = “stephen”
    let request = NSFetchRequest(entityName: “Books”)
    request.predicate = NSPredicate(format: “author.name BEGINSWITH[c] %@”, search)
    do {
        listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
    } catch {
        print(“Error”)
    }
    tableView.reloadData()
}
```

IMPORTANT: The list of keywords and options to create conditions for predicates is extensive. The topic is beyond the scope of this book. For more information, visit our website and follow the links for this chapter.

The Basics: Diacritics are the small marks used in certain languages to change the pronunciation of a letter, like the visible stress in some Spanish vowels. When we specify the character **d** between square brackets, the search ignores these marks and looks for the letter in its basic form. The **c** and **d** characters are usually implemented together, as in **[cd]**.

A practical application of predicates is to check for duplicates. The inclusion of duplicated values is something that every application must be prepared to prevent. For example, if in the application we have created so far we insert an author that already exists, two Authors objects with the same name are stored in the Persistent Store. To avoid this situation we can use a request with a predicate that searches for authors of the same name before creating a new **NSManagedObject** object. The following example modifies the **saveAuthor()** method in the **EditAuthorViewController** class to avoid duplicates.

Listing 22-18: Checking for duplicates

```
@IBAction func saveAuthor(sender: UIBarButtonItem) {
```



```

let name = authorName.text!.stringByTrimmingCharactersInSet(.whitespaceAndNewlineCharacterSet())
if name != "" {
    let request = NSFetchRequest(entityName: "Authors")
    request.predicate = NSPredicate(format: "name = %@", name)
    let total = context.countForFetchRequest(request, error: nil)

    if total == 0 {
        selectedAuthor = NSEntityDescription.insertNewObjectForEntityForName("Authors", inManagedObjectContext:
context)
        selectedAuthor.setValue(name, forKey: "name")
        do {
            try context.save()
            performSegueWithIdentifier("backFromNew", sender: self)
        } catch {
            print("Error")
        }
    }
}
}

```

The method of Listing 22-18 creates a request for the Authors entity and uses the value inserted in the Text Field by the user to create a predicate. The predicate searches for objects with the name equal to the value of the `name` constant. Using this request we call the `countForFetchRequest()` method in the context to get the total amount of objects that match the conditions in the predicate. If the value returned is 0, then we know that there are no authors in the Persistent Store with that name and we can create and add the new `NSManagedObject` object.

Do It Yourself: Replace the `saveAuthor()` method in the `EditAuthorViewController` class of Listing 22-9 by the method of Listing 22-18. Run the application and try to insert an author that already exists. Use the same filter in the `EditBookViewController` class to avoid duplicates when inserting new books. You can create a predicate that checks for books with the same title and author (the placeholder characters `%@` can take any kind of values and objects, including `NSManagedObject` objects).

Sort Descriptors

Objects returned from a request are usually in the order they were created, but this is not guaranteed. To sort the objects in a specific order, Core Data lets us associate the request with an object of type `NSSortDescriptor`. From this class we can create objects that specify an

order according to the values of a property. The sorting criteria is determined by methods provided by the values themselves. For example, data types like `NSDate` and `NSNumber` implement only the `compare()` methods to compare their values, but the `String` structure implements other methods that allow us to do things like order strings without differentiating between lower- and upper-case letters (see `NSString` in Chapter 4). By default, an `NSSortDescriptor` object uses the `compare()` method to compare any type of value, but we can specify others from the class initializer.

`NSSortDescriptor(key: property, ascending: value)`—This initializer creates an `NSSortDescriptor` object that orders the objects according to the value of the property specified by the **`property`** attribute. The **`property`** attribute is a string with the name of the property, and the **`value`** attribute is a Boolean value that determines if the objects will be sort in ascending or descending order. The object created by this initializer sorts the values with the `compare()` method.

`NSSortDescriptor(key: property, ascending: value, selector: method)`—This initializer creates an `NSSortDescriptor` object that orders the objects according to the value of the property specified by the **`property`** attribute. The **`property`** attribute is a string with the name of the property, the **`value`** attribute is a Boolean value that determines if the objects will be sorted in ascending or descending order, and the **`method`** attribute is the comparison method that determines the sorting criteria.

The `NSFetchRequest` class includes the `sortDescriptors` property to define the order of a request. All we have to do to order the objects is to create at least one `NSSortDescriptor` object and assign it to this property. The following example sorts the Books objects by title in ascending order.

Listing 22-19: Sorting the books by title

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    let request = NSFetchRequest(entityName: "Books")
    let sort = NSSortDescriptor(key: "title", ascending: true)
    request.sortDescriptors = [sort]
    do {
        listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
    } catch {
        print("Error")
    }
    tableView.reloadData()
}
```

The `sortDescriptors` property takes an array of `NSSortDescriptor` objects, so we can specify different conditions to sort the list. The final order is established according to the location of the `NSSortDescriptor` objects in the array. The following example sorts the objects by author first and then by year.

Listing 22-20: Sorting books by author and year

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    let request = NSFetchRequest(entityName: "Books")
    let sort1 = NSSortDescriptor(key: "author.name", ascending: true)
    let sort2 = NSSortDescriptor(key: "year", ascending: true)
    request.sortDescriptors = [sort1, sort2]
    do {
        listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
    } catch {
        print("Error")
    }
    tableView.reloadData()
}
```

When we compare string values, we can use other comparison methods than `compare()`. The method must be assigned to the `selector` attribute of the `NSSortDescriptor` initializer. The most frequently used method is `caseInsensitiveCompare()` because it compares values without differentiating between lower- and upper-case letters. The following example implements this method to sort the books by title.

Listing 22-21: Sorting books by title without differentiating between lower- and upper-case letters

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    let request = NSFetchRequest(entityName: "Books")
    let sort = NSSortDescriptor(key: "title", ascending: true, selector: "caseInsensitiveCompare:")
    request.sortDescriptors = [sort]
    do {
        listOfBooks = try context.executeFetchRequest(request) as! [NSManagedObject]
    } catch {
        print("Error")
    }
    tableView.reloadData()
}
```

Do It Yourself: Replace the `viewWillAppear()` method in the `BooksViewController` class of Listing 22-6 by any of the methods presented above to see how sort descriptors work. Change the value of the `ascending` attribute to see how the order changes in each case.

Delete Objects

In the initial interface presented in Figure 22-14 we included a bar button to activate the table's edition mode called `Delete Books`, but we haven't seen how to delete `NSManagedObject` objects yet. The `NSManagedObjectContext` offers the `deleteObject()` method for this purpose. The method takes a reference to the `NSManagedObject` object we want to delete and removes it from the context. As always, the change is only performed in the context, so we have to save the context or the object won't be deleted from the Persistent Store.

The following are the two methods we have to incorporate to the `BooksViewController` class of Listing 22-6 to activate the table's edition mode and delete the `NSManagedObject` object from the Persistent Store when the user presses the `Delete` button.

Listing 22-22: Deleting a book

```
@IBAction func editBooks(sender: UIBarButtonItem) {
    if tableView.editing {
        tableView.setEditing(false, animated: true)
    } else {
        tableView.setEditing(true, animated: true)
    }
}

override func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle,
forRowAtIndexPath indexPath: NSIndexPath) {
    if editingStyle == .Delete {
        let book = listOfBooks[indexPath.row]
        context.deleteObject(book)
        do {
            try context.save()
            listOfBooks.removeAtIndex(indexPath.row)
            tableView.deleteRowsAtIndexPaths([indexPath], withRowAnimation: .Automatic)
        } catch {
            print("Error")
        }
    }
}
```

```
}
```

The first method is the Action we have to associate to the Delete Books button. As always, this method activates or deactivates the table's edition mode depending on its current state. When the mode is activated and the user presses the Delete button in one of the rows, the table calls the `tableView(tableView, commitEditingStyle: editingStyle, forRowAtIndexPath: indexPath)` method on its delegate to perform the action. This method has to check if the style of the action is **Delete** and then erase the object from the context, remove it from the array, and save the context, in that order. In the code of Listing 22-22 we get the `NSManagedObject` object from the `listOfBooks` array corresponding to the row selected by the user and then call the `deleteObject()` method to remove it from the context. The context is saved and, if no error is found, the object is removed from the `listOfBooks` array and the row is deleted from the table.

Do It Yourself: Add the methods of Listing 22-22 to the `BooksViewController` class of Listing 22-6. Connect the Delete Books button on the Navigation Bar to the `editBooks()` method. Run the application, press the Delete Books button and remove the book. Run the application again to check that the book was removed from the Persistent Store.

The Basics: If we need to delete several objects in the same action, we can call the `save()` method at the end to improve performance. This also applies to other operations with the context.

22.2 Fetched Results Controller

Working with arrays to store the results returned by a request is not recommended most of the time. Tables are capable of managing thousands of items, and putting all those items in an array consumes too much memory. The solution is to get from the Persistent Store only the objects the table needs at a particular moment, but this is error prone and demands our code to keep track of the elements already loaded and request only those that we don't have. Programming an application to do this work efficiently is difficult. For this reason, Core Data offers the `NSFetchedResultsController` class. This class provides highly optimized code that intermediates between the table and the Persistent Store; taking care of fetching the objects the table needs and updating the list of objects available when some are updated, added, or removed. To create the controller, the class provides the following initializer.

`NSFetchedResultsController(fetchRequest: request, managedObjectContext: context, sectionNameKeyPath: section, cacheName: cache)`—This initializer creates an `NSFetchedResultsController` object that fetches `NSManagedObject` objects from the Persistent Store according to the request specified by the **request** attribute. The **request** attribute is an `NSFetchRequest` object with the request we want the controller to use to get the objects, the **context** attribute is a reference to the Core Data context, the **section** attribute is an optional string that identifies the name of the property used to create the table's sections, and the **cache** attribute is an optional string with the name of the file the controller uses to cache the objects returned by the request.

When we work with an `NSFetchedResultsController` object, we have to ask this object for any information about the request we need (the objects returned, their values, etc.). The class offers the following properties and methods to access its values.

fetchRequest—This property returns a reference to the `NSFetchRequest` object that represents the request used by the controller.

fetchedObjects—This property returns an array with the `NSManagedObjects` currently fetched by the controller.

sections—This property returns an array of objects with information about every section of the table. The objects implement the properties defined in the `NSFetchedResultsSectionInfo` protocol to provide this information.

sectionIndexTitles—This property returns an array of strings with the titles for the index of every section.

performFetch()—This method executes the fetch request set for the controller. The controller doesn't return any value until this method is called.

objectAtIndexPath(path)—This method returns the `NSManagedObject` object at the index path specified by the **path** attribute. The attribute is an `NSIndexPath` object.

indexPathForObject(object)—This method returns the index path of the `NSManagedObject` object specified by the **object** attribute. The value returned is an `NSIndexPath` object.

After the `NSFetchedResultsController` object is initialized, we have to call its `performFetch()` method to execute the request and get the `NSManagedObject` objects we need. The results produced by the request are stored in a temporal container and automatically updated by the `NSFetchedResultsController` object every time a modification is introduced to the context (an object is modified, deleted, moved, or new objects are added). Because of the close relationship between this controller and the Table View, it is important to make sure that all the changes are immediately reflected in the table. Core Data simplifies this task with the addition of the `NSFetchedResultsControllerDelegate` protocol. The protocol defines methods that are called every time a change occurred in the objects managed by the `NSFetchedResultsController` object, and therefore we can use them to update the Table View.

controllerWillChangeContent(controller)—This method is called on the delegate when the controller is going to start processing changes.

controller(controller, didChangeObject: anObject, atIndexPath: indexPath, forChangeType: type, newIndexPath: newIndexPath)—This method is called on the delegate when an object managed by the controller changed. The **object** attribute is a reference to the `NSManagedObject` object that changed, the **indexPath** and **newIndexPath** attributes are `NSIndexPath` objects that represent the old and the new path of the object, and the **type** attribute is a value of the `NSFetchedResultsControllerChangeType` enumeration that indicates the type of the change. The possible values are `Insert`, `Delete`, `Move`, and `Update`.

controller(controller, didChangeSection: sectionInfo, atIndex: sectionIndex, forChangeType: type)—This method is called on the delegate when a section changed. The **sectionInfo** attribute is an object that adopts the `NSFetchedResultsControllerSectionInfo` protocol to provide information about the section, the **sectionIndex** attribute is an integer with the index of the section, and the **type** attribute is an enumeration of type `NSFetchedResultsControllerChangeType` that indicates the type of change. The possible values are `Insert`, `Delete`, `Move`, and `Update`.

controllerDidChangeContent(controller)—This method is called on the delegate when the controller finished processing changes.

controller(controller, sectionIndexTitleForSectionName: sectionName)—

This method is called on the delegate to get the title for the index of a section. The **sectionName** attribute is a string with the name of the section. If the method is not implemented, the controller returns the capitalized first letter of the section's name.

A Table View associated with an **NSFetchedResultsController** object works exactly like any other table; with the exception that now the information is provided by the object instead of an array. In the following example we modify the **BooksViewController** class introduced in Listing 22-6 to fetch books using an **NSFetchedResultsController** object.

Listing 22-23: *Fetching objects with an NSFetchedResultsController object*

```
import UIKit
import CoreData

class BooksViewController: UITableViewController, NSFetchedResultsControllerDelegate {
    var context: NSManagedObjectContext!

    var fetchedController: NSFetchedResultsController!

    override func viewDidLoad() {
        super.viewDidLoad()

        tableView.rowHeight = 100

        let app = UIApplication.sharedApplication()
        let appDelegate = app.delegate as! AppDelegate
        context = appDelegate.stack.context

        let request = NSFetchedRequest(entityName: "Books")
        let sort = NSSortDescriptor(key: "title", ascending: true, selector: "caseInsensitiveCompare:")
        request.sortDescriptors = [sort]

        fetchedController = NSFetchedResultsController(fetchRequest: request, managedObjectContext: context,
        sectionNameKeyPath: nil, cacheName: nil)
        fetchedController.delegate = self
        do {
            try fetchedController.performFetch()
        } catch {
            print("Error")
        }
    }

    override func tableView(tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
        if let sections = fetchedController.sections {
            let sectionInfo = sections[section]
            return sectionInfo.numberOfObjects
        }
    }
}
```



```

    }
    return 0
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("booksCell", forIndexPath: indexPath) as! BooksCell
    updateCell(cell, path: indexPath)
    return cell
}

func updateCell(cell: BooksCell, path: NSIndexPath) {
    let book = fetchedResultsController.objectAtIndexPath(path)
    cell.bookTitle.text = book.valueForKey("title") as? String
    cell.bookCover.image = book.valueForKey("thumbnail") as? UIImage
    let year = book.valueForKey("year") as? Int
    cell.bookYear.text = "Year: \ \(year!)"

    let authorName = book.valueForKey("author")?.valueForKey("name") as? String
    if authorName != nil {
        cell.bookAuthor.text = authorName
    } else {
        cell.bookAuthor.text = "Not Defined"
    }
}
}

```

The first thing we need to do as soon as the view is loaded is to initialize the **NSFetchedResultsController** object and call its **performFetch()** method to execute the request. The request must include a sort descriptor for the controller to know the order in which the **NSManagedObject** objects are going to be offered to the table, so in Listing 22-23 we create the **NSFetchRequest** object for the Books entity and assign it a sort descriptor that orders the books by title. Besides the request, the **NSFetchedResultsController** initializer requires three more values: a reference to the context, the name of the property that is going to be used to create the sections for the table, and the name of the file used to cache the information for better performance. In this example we are not going to use the last two parameters, so we declared them as **nil**.

The next step is to tell the table how many rows it has to create per section from the **tableView(tableView, numberOfRowsInSection: section)** method. The **NSFetchedResultsController** class was designed to work with sections and defines objects that implement properties of the **NSFetchedResultsSectionInfo** protocol to provide information for every section, including the number of objects they contain. The properties defined by this protocol are the following.

numberOfObjects—This property returns an integer with the number of objects

designated to the section.

objects—This property returns an array with references to the objects in the section.

name—This property returns the name of the section.

indexTitle—This property returns the index's title for the section (by default, the index's title is the capitalized first letter of the name of the section).

In the `tableView(tableView, numberOfRowsInSection: section)` method of Listing 22-23 we read the `sections` property of the `NSFetchedResultsController` object to get the array of objects that represent the sections, get the object corresponding to the current section, and return the value of its `numberOfObjects` property to tell the table how many objects are in the section.

The last step to configure the table is to get the values for every cell. For this example we configure the cell in a separate method called `updateCell()` because, as we will see later, other protocol methods may need to update the values of the cell and we want to avoid code duplication. To access the `NSManagedObject` objects returned by the request and get the values for the cells, the `NSFetchedResultsController` object offers the `objectAtIndexPath()` method. Using the cell's index path as the method's attribute we get back the `NSManagedObject` corresponding to the cell and read its values as we did in previous examples.

Do It Yourself: Replace the `BooksViewController` class in your project with the class of Listing 22-23 (this example doesn't include the Action for the Delete Books button, so you should delete the connection before executing the app). Run the application. The Table View Controller should list the books as it did before.

At this point the view controller can show the books available, but if we introduce new books or try to delete them, the application won't work as before or will even return an error because the information managed by the `NSFetchedResultsController` object doesn't match the information contained by the table. To keep the Table View updated, we have to adopt the `NSFetchedResultsControllerDelegate` protocol and implement its methods. In the example of Listing 22-23 we have already adopted the `NSFetchedResultsControllerDelegate` protocol and declare the view controller as the delegate of the `NSFetchedResultsController` object, so all that is left is to add the methods to the class.

Listing 22-24: *Implementing the `NSFetchedResultsControllerDelegate` protocol methods*

```
func controller(controller: NSFetchedResultsController, didChangeObject anObject: AnyObject, atIndexPath indexPath:
NSIndexPath?, forChangeType type: NSFetchedResultsChangeType, newIndexPath: NSIndexPath?) {
    switch type {
    case .Delete:
        if let path = indexPath {
```

```

tableView.deleteRowsAtIndexPaths([path], withRowAnimation: .Fade)
}
case .Insert:
if let path = newIndexPath {
tableView.insertRowsAtIndexPaths([path], withRowAnimation: .Fade)
}
case .Update:
if let path = indexPath {
    let cell = tableView.cellForRowAtIndexPath(path) as! BooksCell
    updateCell(cell, path: path)
}
default:
break
}
}

```

Because our table just has one section so far, the only protocol method required to keep it updated is the **controller(controller: didChangeObject: anObject, atIndexPath: indexPath, forChangeType: type, newIndexPath: newIndexPath)** method. This method receives a value of type **NSFetchedResultsControllerChangeType** that we can use to know the type of change performed and update the table accordingly. In this example, we check if an object has been deleted (**Delete**), added (**Insert**), or modified (**Update**) with a **switch** statement. If an object was deleted, we delete its row, if it was inserted, we insert a new row (notice that we have to use the value of the **newIndexPath** attribute this time), and if the object was updated, we get a reference to its cell and call the **updateCell()** method to update its values.

Do It Yourself: Add the method of Listing 22-24 to the **BooksViewController** class of Listing 22-23. Run the application and add a new book. The Table View should update its rows to show the new object.

If we perform multiple changes at the same time, we can tell the table to update all of its rows at the end by calling the **beginUpdates()** method when the process starts and the **endUpdates()** method when it is over. The **NSFetchedResultsControllerDelegate** protocol defines the methods **controllerWillChangeContent()** and **controllerDidChangeContent()** for this purpose.

Listing 22-25: Updating the table all at once

```

func controllerWillChangeContent(controller: NSFetchedResultsController) {
    tableView.beginUpdates()
}
func controllerDidChangeContent(controller: NSFetchedResultsController) {

```

```
tableView.endUpdates()
}
```

IMPORTANT: The `BooksViewController` class we have implemented with these examples doesn't include the methods necessary to activate the table's edition mode and delete books, like those introduced in Listing 22-22. You can incorporate those methods to your class, just remember to remove the call to the Table View's `deleteRowsAtIndexPaths()` method, because now the table is updated from the `NSFetchedResultsControllerDelegate` protocol method.

The Basics: If you incorporate the methods of Listing 22-25 to the `BooksViewController` class of Listing 22-23, the table will wait until all the changes to the context have been made to update its rows. If we only perform one change at a time, like adding a new book, these methods are not necessary.

Sections

As we mentioned before, the `NSFetchedResultsController` object was designed to work with sections. It can take a property of the `NSManagedObject` objects and generate sections with their values. For example, we can create one section for every author's name. The `NSFetchedResultsController` object divides the information in sections, with every section containing the books of a particular author, and automatically assigns the name of the author to the section's title. All we have to do is to declare the property we want to use as the value of the `sectionNameKeyPath` attribute of the object's initializer, as shown in the following example.

Listing 22-26: Organizing the `NSManagedObject` objects in sections

```
override func viewDidLoad() {
    super.viewDidLoad()

    tableView.rowHeight = 100
    let app = UIApplication.sharedApplication()
    let appDelegate = app.delegate as! AppDelegate
    context = appDelegate.stack.context

    let request = NSFetchedRequest(entityName: "Books")
    let sort1 = NSSortDescriptor(key: "author.name", ascending: true, selector: "caseInsensitiveCompare:")
    let sort2 = NSSortDescriptor(key: "title", ascending: true, selector: "caseInsensitiveCompare:")
    request.sortDescriptors = [sort1, sort2]
```

```

fetchResultsController = NSFetchedResultsController(fetchRequest: request, managedObjectContext: context,
sectionNameKeyPath: "author.name", cacheName: nil)

fetchResultsController.delegate = self

do {
    try fetchResultsController.performFetch()
} catch {
    print("Error")
}
}

```

The value of the `sectionNameKeyPath` attribute only determines the property we want to use to separate the objects in sections (`author.name` in this case), but we also have to sort the objects to coincide with this configuration. We can assign to the request as many sort descriptors as we want, but the first one has to order the objects by the value of the same property we used to generate the sections. In the example of Listing 22-26 we sort the sections by the value of the `name` property of the `author` relationship first and then add another `NSSortDescriptor` object to sort the objects inside each section by the value of the `title` property.

Of course, this is not enough to get the table to present the information in sections. We also have to implement the necessary protocol methods, including the `UITableViewDataSource` protocol methods to let the table know the number of sections available and the title for each section, and an additional `NSFetchedResultsControllerDelegate` protocol method to update the sections.

Listing 22-27: Implementing the necessary protocol methods to work with sections

```

override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    if let sections = fetchResultsController.sections {
        return sections.count
    }
    return 1
}

override func tableView(tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
    if let sections = fetchResultsController.sections {
        let sectionInfo = sections[section]
        return sectionInfo.name
    }
    return nil
}

func controller(controller: NSFetchedResultsController, didChangeSection sectionInfo: NSFetchedResultsSectionInfo,
atIndex sectionIndex: Int, forChangeType type: NSFetchedResultsChangeType) {
    switch type {

```

```

case .Insert:
let index = NSIndexPath(index: sectionIndex)
tableView.insertSections(index, withRowAnimation: .Fade)
case .Delete:
let index = NSIndexPath(index: sectionIndex)
tableView.deleteSections(index, withRowAnimation: .Fade)
default:
break
}
}

```

To return the number of sections available we just count the values in the `sections` property of the `NSFetchedResultsController` object. For the name of the section we need to get the object that represents the section and return its `name` property. Finally, to update the sections in the table after every change, we execute the Table View's `insertSections()` and `deleteSections()` methods to insert or remove a section depending on the type of change reported by the `NSFetchedResultsController` object.

Sections are identified with an object of the `NSIndexPath` class. This object manages a collection of integers that represent indexes (in this case, the indexes of the sections). The Table View's methods require these values to recognize the section to be affected. That is why, in Listing 22-27, we create these objects before modifying the section.

With the inclusion of these methods, now the table can present sections organized according to the name of the authors and ordered by title. The result is shown in Figure 22-18.

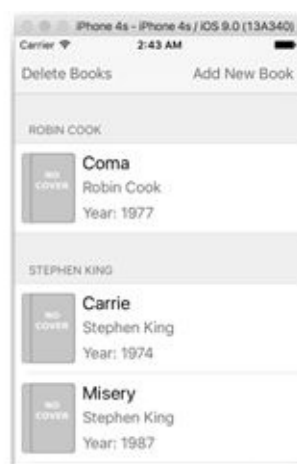


Figure 22-18: Organizing the books in sections

Do It Yourself: Replace the `viewDidLoad()` method of the `BooksViewController` class of Listing 22-23 with the method of Listing 22-26. Add the methods in Listing 22-27 to the class. Select the Table View in the Storyboard and change the style to Grouped

from the Attributes Inspector panel. Run the application. You should see something similar to Figure 22-18.

If we want to incorporate an index for the sections, as we did in the examples of Chapter 14 (Listing 14-7), we have to implement the `UITableViewDataSource` protocol method `sectionIndexTitlesForTableView(tableView)` and return the array we get from the `sectionIndexTitles` property, as in the following example (the property returns an array with the initial letters of the sections' names capitalized).

Listing 22-28: Generating an index

```
override func sectionIndexTitlesForTableView(tableView: UITableView) -> [String]? {  
    return fetchedController.sectionIndexTitles  
}
```

Search

The process to let users search for values in a Table View associated with an `NSFetchedResultsController` object doesn't differ from a normal Table View. We have to add the search bar to the Storyboard (as we did in Chapter 14, Figure 14-31), connect the bar to an Outlet, adopt the `UISearchBarDelegate` protocol, declare the view controller as the bar's delegate (as we did in Listing 14-30), and implement the protocol methods. In the `searchBar(searchBar, textDidChange: searchText)` method we have to modify the predicate of the request assigned to the `NSFetchedResultsController` object to search for the value inserted by the user. The following example presents the protocol methods necessary for our `BooksViewController` class to search for books the titles of which contain the text in the search bar.

Listing 22-29: Searching books

```
func searchBar(searchBar: UISearchBar, textDidChange searchText: String) {  
    let request = fetchedController.fetchRequest  
    request.predicate = NSPredicate(format: "title CONTAINS[cd] %@", searchText)  
    do {  
        try fetchedController.performFetch()  
    } catch {  
        print("Error")  
    }  
    tableView.reloadData()  
}
```

```

func searchBarCancelButtonClicked(searchBar: UISearchBar) {
    searchBooks.text = ""
    searchBooks.setShowsCancelButton(false, animated: true)
    searchBooks.resignFirstResponder()

    let request = fetchedController.fetchRequest
    request.predicate = nil
    do {
        try fetchedController.performFetch()
    } catch {
        print("Error")
    }
    tableView.reloadData()
}

func searchBarTextDidBeginEditing(searchBar: UISearchBar) {
    searchBooks.setShowsCancelButton(true, animated: true)
}

```

To perform a search we don't have to replace the request or create a new **NSFetchedResultsController** object, all we have to do is to assign a new predicate to the request with the conditions we need and call the **performFetch()** method to execute the request again. In Listing 22-29 we modify the predicate every time the text in the search bar changes, and assign the original predicate back when the search is cancelled (in this case there was no original predicate so we declare it as **nil**).

Do It Yourself: Add a Search Bar between the Table View and the Navigation Bar of the view managed by the **BooksViewController** class, as we did in Figure 14-31. Connect the bar to an Outlet called **searchBooks**, make the view controller adopt the **UISearchBarDelegate** protocol, and assign the view controller as the delegate, as we did in Listing 14-30. Add the methods of Listing 22-29 to the **BooksViewController** class. If you have implemented all the protocol methods introduced in the previous section of this chapter, the rows and sections should appear or disappear according to the term inserted in the search bar.

22.3 Migration

The Persistent Stores Coordinator creates the Persistent Stores based on the Core Data model defined for the application. All the examples in this chapter work on the Core Data model created at the beginning. We didn't introduce any modification to the entities or their attributes. But if we had introduced a change, no matter how insignificant, the app would have crashed, because the new model wouldn't have been compatible with the one used to create the Persistent Store. If we want to update the model, adding or modifying entities, attributes, or relationships, we have to generate a new version of the model. This is a process called *migration*, and it involves the creation of a new model based on the old one. The option to add the new version of the model is available in the Editor menu at the top of the screen. When we click on the Add Model Version... option on this menu, a pop-up window asks for the name we want to assign to the new model and the model it is based on, as shown in Figure 22-19.



Figure 22-19: *New model version*

After we click on Finish, the new version is created and added to the models available in the Navigation Area. We can click on the arrow at the left side of the xcdatamodeld file to see the list.

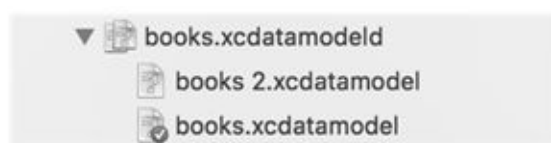


Figure 22-20: *Model versions available*

There is a green badge indicating the current active model. To start using the new version, we have to select it from the File Inspector panel in the Utilities Area. The option is called Model Version, as shown in Figure 22-21.



Figure 22-21: *Selecting model version*

Once we select the new version from this panel, the green badge is assigned to it in the Navigation Area. Now, when the `xcdatamodeld` file is selected, the new model is shown in the Editor Area. All the modifications must be introduced in the new version; the old version is only preserved as reference.

The process we just described provides our app with a new model we can update, but it doesn't tell the app that it has to migrate the Persistent Store from the old model to the new one. For this purpose we have to modify the Core Data stack to initialize the Persistent Store with two options: `NSMigratePersistentStoresAutomaticallyOption` and `NSInferMappingModelAutomaticallyOption`, both with the value `true`. This tells Core Data that it has to try to migrate to a new model version when available. The options are assigned to the Persistent Store when it is initialized by the coordinator with the `addPersistentStoreWithType()` method, as in the following example.

Listing 22-30: *Migrating to a new model version*

```
import Foundation
import CoreData

class CoreDataStack: NSObject {
    var context: NSManagedObjectContext!
    override init() {
        super.init()

        let bundle = NSBundle.mainBundle()
        let modelURL = bundle.URLForResource("books", withExtension: "momd")
        let manager = NSFileManager.defaultManager()
        let urls = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let url = urls.last!
        let localURL = url.URLByAppendingPathComponent("store.sqlite")
        let model = NSManagedObjectModel(contentsOfURL: modelURL!)

        let options = [NSMigratePersistentStoresAutomaticallyOption: true,
            NSInferMappingModelAutomaticallyOption: true]
        let coordinator = NSPersistentStoreCoordinator(managedObjectModel: model!)
        try! coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: localURL,
            options: options)
        context = NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)
        context.persistentStoreCoordinator = coordinator
    }
}
```

```
}  
}
```

Do It Yourself: Select the `xcdatamodeld` file in the Navigation Area. Open the Editor menu at the top of the screen and select the option `Add Model Version...` Insert a name for the new version (different than the previous name) and click `Finish`. With the model selected, go to the Utilities Area, open the File Inspector panel (first button at the top), and change the value in the `Model Version` option to the new version you just created (Figure 22-19). Select the model again to make sure that the model you see in the Editor Area is the new one. Add a new attribute to the `Authors` entity, such as the age of the author. Update the `CoreDataStack` class with the code of Listing 22-30. Run the application. Everything should work as expected with the difference that now you have a new property in the `Authors` objects that you can use to insert the age of the author.

IMPORTANT: You don't have to add these options only after a new version of the model is introduced; you can assign them to the initial Core Data stack if you want. Once the options are added, always keep them there, even when future versions of the app use the same model. You never know when the users are going to update your app on their devices.

Chapter 23

iCloud

23.1 Data in the Cloud

These days, users have more than one device available. If we create an application that works on multiple devices, we have to provide a way for the users to share their data, otherwise they will have to insert the same information on every device they own. The only way to do it effectively is through a server. The data from one device is stored in a server and then retrieved by the other device. But setting up a server running this kind of system is complicated and costly. To provide a standard solution that every developer can easily implement in his or her applications without any prior investment, Apple created a system called iCloud.

iCloud is a complete system that allows applications to synchronize data across devices. It provides three basic services: Key/Value Storage, to store small pieces of information, Document Storage, to store files and data produced by file-based systems like Core Data, and CloudKit Storage, an elemental system that provides more control over the data moved to and from iCloud.

Enabling iCloud

The iCloud service must be enabled for each application we create. We have to create entitlements to authorize our app to use the service and also a container in iCloud where our app's data will be stored. Xcode can set up everything for us by just turning on a switch. The option is available in the Capabilities panel inside the app's settings window (Figure 5-3, number 5). When we open this panel, we see all the external services provided by Apple, including iCloud at the top of the list, as shown in Figure 23-1.

IMPORTANT: iCloud services are only available for developers that are members of the Apple Developer Program. At the time of this writing, the membership costs \$99 US Dollars per year. You also have to register your account with Xcode, as we explained in Chapter 5 (see Figures 5-7 and 5-8).



Figure 23-1: Activating iCloud for the app

When we turn on the switch on the right, Xcode asks for the account we want to use. These are the accounts registered in Xcode (see Chapter 5, Figures 5-7 and 5-8). For a single developer there is usually only one account available. Whatever is the case, the account selected must be part of the Apple Developer Program.

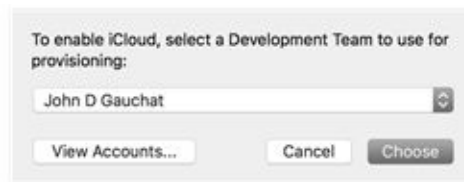


Figure 23-2: *Selecting a developer account*

After we select an account, the iCloud option is expanded to show the services available. We have to select the services we want for our app and Xcode takes care of creating the entitlements and the container if necessary. Figure 23-3 shows the panel with the services we need for the examples of this chapter activated (Key/Value storage and Documents storage).



Figure 23-3: *Activating iCloud services*

Do It Yourself: Create a new Single View Application project. Click on the app’s settings option at the top of the Navigation Area (Figure 5-3, number 5) and open the Capabilities panel (Figure 23-1). Turn on the switch for the iCloud option, select your account in the popup window, and activate the options Key-value storage and iCloud Documents. After a few seconds, Xcode creates the entitlements and the iCloud container, and iCloud services become available for the application.

Testing Devices

The best way to test an iCloud application is running the app in two different devices, but Apple has made iCloud services available in the simulator as well. Thanks to this feature, we can now synchronize data between a device and the simulator to test our app.

For the simulator to be able to access iCloud services, we have to register our iCloud account in the Settings app. The simulators behave like a real device. To access the Settings app, we have to select the option Home in the Hardware menu at the top of the screen, and then click on the Settings app's icon (Figure 23-4, left). Inside the app, we have to select the iCloud option (Figure 23-4, center) and complete the information requested (Figure 23-4, right).

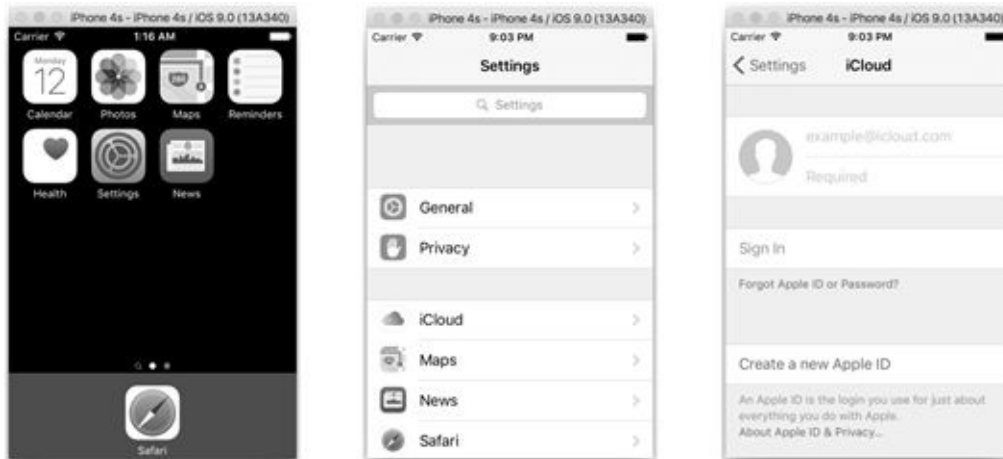


Figure 23-4: iCloud account in simulator

Do It Yourself: Run the application. When the simulator finishes launching your app, click on the Hardware menu at the top of the screen and select the option Home. Navigate the screen to find the Settings app's icon. Select the iCloud option and insert the email and password corresponding to your iCloud account. iCloud services are now available in this simulator, but you have to repeat the process for every type of simulator you use.

23.2 Key/Value Storage

The Key/Value storage is the `NSUserDefaults` system for iCloud. It works exactly like the user defaults system studied in Chapter 21, and even shares some of its methods to assign and retrieve values, but all the data is stored in iCloud servers instead of the device's storage space. We can use it to store the app's preferences or current state, for example, or any other value that has to be set automatically in every device.

Foundation defines the `NSUbiquitousKeyValueStore` class to provide access to this system. The class includes the following methods to store and retrieve values.

setBool(value, forKey: key)—This method stores a value of type `Bool`.

setDouble(value, forKey: key)—This method stores a value of type `Double`.

setLongLong(value, forKey: key)—This method stores a value of type `Int64` (long long is the way 64 bits integers are described in C and Objective-C).

setString(value, forKey: key)—This method stores a value of type `String`.

setArray(value, forKey: key)—This method stores an array.

setDictionary(value, forKey: key)—This method stores a dictionary.

setData(value, forKey: key)—This method stores a value of type `NSData`.

setObject(value, forKey: key)—This method stores an object. The data types allowed are restricted to the Property List values studied in Chapter 18 (`NSNumber`, `NSString`, `NSDate`, `NSArray`, `NSDictionary`, `NSData`, and their equivalents in Swift)

boolForKey(key)—This method retrieves a value of type `Bool`.

doubleForKey(key)—This method retrieves a value of type `Double`.

longLongForKey(key)—This method retrieves a value of type `Int64`.

stringForKey(key)—This method retrieves a value of type `String`.

arrayForKey(key)—This method retrieves an array.

dictionaryForKey(key)—This method retrieves a dictionary.

dataForKey(key)—This method retrieves a value of type `NSData`.

objectForKey(key)—This method retrieves an object.

To store or access a value we have to initialize an `NSUbiquitousKeyValueStore` object and then call its methods. The object takes care of establishing the connection with iCloud and downloading or uploading the values. As we already mentioned, the system is used to storing discrete values that represent the user's preferences or the app's latest status. For example, we may have a Stepper that lets the user set a limit to the number of items the application can manage.



Figure 23-5: Simple interface to test Key/Value storage

If the user decides to set the limit to 5 in one device, the same limit should be established when the application is executed on a different device. To achieve this, we have to read the value from iCloud when the view is loaded and store it back when the user sets a new one. The following is the view controller necessary for this application.

Listing 23-1: Storing settings in iCloud

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var stepper: UIStepper!
    @IBOutlet weak var counter: UILabel!
    var kvStorage: NSUbiquitousKeyValueStore!

    override func viewDidLoad() {
        super.viewDidLoad()
        kvStorage = NSUbiquitousKeyValueStore()
        let control = kvStorage.doubleForKey("control")
        stepper.value = control
        counter.text = String(control)
    }

    @IBAction func changeValue(sender: UIStepper) {
        let current = stepper.value
        counter.text = String(current)
        kvStorage.setDouble(current, forKey: "control")
        kvStorage.synchronize()
    }
}
```

The example of Listing 23-1 creates the `NSUbiquitousKeyValueStore` object as soon as the view is loaded and reads a value of type `Double` identified with the key “control” from iCloud. Like it happens with some of the methods included in the `NSUserDefaults` class, the `doubleForKey()` method used in this example returns 0 if the value is not found, so the first time the application is executed, the value assigned to the label will be 0. To update the value, we have created an Action for the Stepper called `changeValue()`. In this method we get the current Stepper’s value, assign it to the label to reflect the change on the screen, and then store the value in iCloud with the `setDouble()` method and the “control” key. If the user opens the application in a different device, the process starts again by reading the value associated with the “control” key from iCloud, but this time it will find the last number stored by the `setDouble()` method.

Notice that we have to execute a method called `synchronize()` after storing a new value. This is a method provided by the `NSUbiquitousKeyValueStore` class to force the system to send the new value to iCloud right away. It is not always necessary, only when we want to make sure that the value is there as soon as possible.

Do It Yourself: Create a new Single View Application project. Add a Stepper and a label to the initial view, as illustrated in Figure 23-5. Connect the Stepper to the `ViewController` class with an Outlet called `stepper` and the label with an Outlet called `counter`. Connect the Stepper with an Action called `changeValue()` and complete the `ViewController` class with the code of Listing 23-1. Run the application in a simulator. Press the Stepper’s button to change the value to 5. Stop the application and run it again. The label should contain the value 5 this time. If you run the app on a device or a different simulator, you should see the value 5 again on the screen (remember to activate the same iCloud account in any simulator or device you try).

The previous example stores in iCloud a value associated with the key “control”, and every time the app is executed the value is retrieved from iCloud servers and shown on the screen. The problem is that at this moment the application doesn’t know when the value is modified from another device. If we launch the app on an iPhone, set the value to five, and then launch it again on an iPad, we will see the value 5 on the iPad’s screen, but modifying the value after this won’t produce any effect on the opposite device. To report the changes, Foundation defines a notification for the `NSUbiquitousKeyValueStore` class called `NSUbiquitousKeyValueStoreDidChangeExternallyNotification`. This notification is posted to the Notification Center when the system detects a change in the values on the Key/Value storage. We have to listen to this notification so we can update the interface every time a new value is received.

Listing 23-2: Updating the interface when a new value is received

```
import UIKit
```

```

class ViewController: UIViewController {
    @IBOutlet weak var stepper: UIStepper!
    @IBOutlet weak var counter: UILabel!
    var kvStorage: NSUbiquitousKeyValueStore!
    override func viewDidLoad() {
        super.viewDidLoad()

        kvStorage = NSUbiquitousKeyValueStore()
        let control = kvStorage.doubleForKey("control")
        stepper.value = control
        counter.text = String(control)

        let center = NSNotificationCenter.defaultCenter()
        center.addObserver(self, selector: "valueReceived:", name:
NSUbiquitousKeyValueStoreDidChangeExternallyNotification, object: kvStorage)
    }
    @IBAction func changeValue(sender: UIStepper) {
        let current = stepper.value
        counter.text = String(current)
        kvStorage.setDouble(current, forKey: "control")
        kvStorage.synchronize()
    }
    func valueReceived(notification: NSNotification) {
        let control = kvStorage.doubleForKey("control")
        stepper.value = control
        counter.text = String(control)
    }
}

```

The notification is registered in the `viewDidLoad()` method, so every device that is running the application will get a notification when the value is modified from another device. The observer executes the `valueReceived()` method. In this method we read the new value and update the interface. Now we can have the application running simultaneously on different devices and the user will see the value automatically changing on the screen when it is modified from another device.

Do It Yourself: Update the `ViewController` class or your project with the code of Listing 23-2. Run the application simultaneously in a simulator and a device. Modify the value on the simulator and check the device to see how the label changes.

IMPORTANT: The simulator doesn't update the information automatically. Most of the time you have to force the update. If you modify the value on your device and don't see it changing on the simulator, open the Debug menu at the top of the screen and select the option Trigger iCloud Sync. This will synchronize the application with iCloud and update the values right away.

23.3 UIDocument

The Key/Value storage system was developed to store simple values. The purpose is to allow users to set preferences or configuration parameters across devices. Although very useful, it presents some limitations, especially on the amount of data we can store (currently no more than 1 megabyte). Therefore, if we need more flexibility and process a large amount of data, we have to work with files.

We already studied how to create and read files in Chapter 21, but iCloud introduces some challenges that the `NSFileManager` class can't overcome. The most important is coordination. Because of the unreliability of network connections, at any moment iCloud may find different versions of the same file. Modifications that were introduced to the file from one device may not have reached iCloud and therefore may later be in conflict with updates introduced from another device. The application has to decide which version of the file to preserve or what data is more valuable when the file is edited from two different devices at the same time. These issues are not easy to solve and can turn development into a nightmare. Considering the problems a developer has to face, Apple introduced a class called `UIDocument`, designed specifically to manage files for iCloud. The class includes capabilities to coordinate and synchronize files of any size, and also features that simplify the manipulation of documents in iOS devices, like progression reports, automatic thumbnail generation, undo manager, and others.

Like other UIKit classes, the `UIDocument` class was not designed to be implemented directly in our code; it is more like an interface between the app's data and the files we use to store it. To take advantage of this class we have to create a subclass and overwrite some of its methods. Once we have the subclass, we can create the object with the following initializer.

`init(fileURL: URL)`—This initializer creates a new `UIDocument` object. The initializer must be called from a subclass of `UIDocument`. The `URL` attribute is an `NSURL` object with the location of the file in iCloud.

The following are the methods we need to overwrite in the subclass to provide the data for the file and to retrieve it later.

`contentsForType(type)`—This method is called when the `UIDocument` object needs to store the content of the document on file. The method must return an object with the document's data (usually an `NSData` object). The `type` attribute is a string that identifies the type of the file (by default it is determined from the file's extension).

`loadFromContents(content, ofType: type)`—This method is called when the `UIDocument` object loads the content of the document from the file. The `content` attribute

is an object with the content of the file (usually an `NSData` object) and the **type** attribute is a string that identifies the type of the file (by default it is determined from the file's extension).

The `UIDocument` class also offers methods to manage the file, including the following to open and save it.

openWithCompletionHandler(handler)—This method asks the `UIDocument` object to open the file and load its content. The **handler** attribute is a closure with the statements to execute after the process is completed.

saveToURL(url, forSaveOperation: operation, completionHandler: handler)—This method asks the `UIDocument` object to save the content of the document on file. The **operation** attribute is an enumeration of type `UIDocumentSaveOperation` to indicate the type of operation to perform. The possible values are `ForCreating` (to save the file for the first time) and `ForOverwriting` (to overwrite the file's current version). The **handler** attribute is a closure with the statements to execute after the process is completed.

Getting access to the files is also complicated in iCloud. We can't just get a list of files with methods like `contentsOfDirectoryAtPath()` from the `NSFileManager` class because there could be some that haven't been downloaded yet to the device. What we can do instead is to get the information pertaining to the files. This data is called *metadata*, and refers to all the information associated with a particular file, such as its name, the date it was created, etc. To get the metadata, Foundation defines the `NSMetadataQuery` class. This class provides the properties and methods necessary to retrieve the information and also keep watching for updates.

predicate—This property sets or returns the predicate for the query. It is of type `NSPredicate`.

sortDescriptors—This property sets or returns the sort descriptors for the query. It is an array of `NSSortDescriptor` objects.

searchScopes—This property sets or returns a value that indicates the scope of the query. It is a dictionary with constants that represent a predefined scope. The constants available for iOS are `NSMetadataQueryUbiquitousDocumentsScope` (searches all files in the Documents directory of the iCloud's container) and `NSMetadataQueryUbiquitousDataScope` (searches all files that are not in the Documents directory of the iCloud's container).

results—This property returns an array with the query's results. By default, the array contains `NSMetadataItem` objects with the metadata of every file found.

resultCount—This property returns an integer with the number of results produced by the query.

startQuery()—This method initiates the query.

stopQuery()—This method stops the query.

enableUpdates()—This method enables query updates.

disableUpdates()—This method disables query updates.

The `NSMetadataQuery` class also includes some notifications to report when new data is available. The following are the most frequently used.

NSMetadataQueryDidUpdateNotification—This notification is posted when the results of the query change.

NSMetadataQueryDidFinishGatheringNotification—This notification is posted when the query finishes getting all the information.

The results of a query are communicated through the `results` property in the form of an array of `NSMetadataItem` objects. This is a simple class created to contain the attributes of a file. The class provides the following method to retrieve the values.

valueForAttribute(key)—This method returns the value of the file's attribute determined by the **key** attribute. The `NSMetadataItem` class defines a list of constants to represent the attributes. The constants available are `NSMetadataItemFSNameKey` (file's name), `NSMetadataItemDisplayNameKey` (document's name), `NSMetadataItemURLKey` (file's URL), `NSMetadataItemPathKey` (file's path), `NSMetadataItemFSSizeKey` (file's size), `NSMetadataItemFSCreationDateKey` (date of creation), and `NSMetadataItemFSContentChangeDateKey` (date the file was last modified).

The interface for an application capable of processing documents has to include a way to select the document the user wants to see and the tools to edit its content. For the following example we work with only one document to keep it simple. The initial view includes a button with the title Edit My Document connected to a second view containing a Text View to edit the document's content.

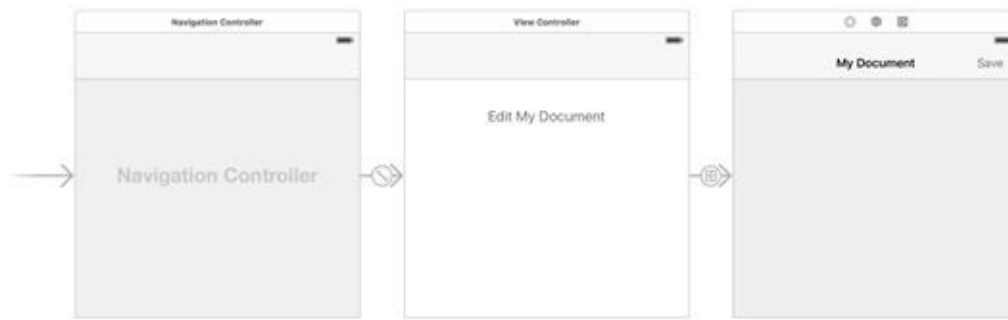


Figure 23-6: *Interface to edit documents*

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add the Edit My Document button to the initial view. Add a second view to the Storyboard. Connect the button with a Show Segue to the second view. Add a Navigation Item and then a bar button with the title Save to the Navigation Bar of the second view. Add a Text View to the second view and pin it to the edges of the view. Click on the first item in the Navigation Area to open the app’s Settings. Select the Capabilities panel, turn the iCloud switch on, and enable the services Key-value storage and iCloud documents.

The first step is to create the `UIDocument` subclass that is going to be in charge of managing our documents (in this example only one). As always, it is recommended to create a separate file for the subclass. We just have to follow the steps to create a Cocoa Touch class file, as we did for other UIKit subclasses before, and select the `UIDocument` class as the superclass. For this example, we call it **MyDocument**.

Listing 23-3: *Creating the document*

```
import UIKit

class MyDocument: UIDocument {
    var fileContent: String = ""

    override func contentsForType(typeName: String) throws -> AnyObject {
        let data = NSKeyedArchiver.archivedDataWithRootObject(fileContent)
        return data
    }

    override func loadFromContents(contents: AnyObject, ofType typeName: String?) throws {
        if let text = contents as? NSData {
            if text.length > 0 {
                fileContent = NSKeyedUnarchiver.unarchiveObjectWithData(text) as! String
            }
        }
    }
}
```

```
}
```

When the `UIDocument` object is asked to store or load the data in file, it calls its `contentsForType()` and `loadFromContents()` methods, respectively. In these methods we have to prepare the application's data to be stored or extract it from the content of the file. For this purpose, we use the methods provided by the `NSKeyedArchiver` and `NSKeyedUnarchiver` classes. The document of our example contains only a string, so we turn the string into an `NSData` object to provide the content for the file in the `contentsForType()` method and get the string back when the file is loaded in the `loadFromContents()` method. Notice that the `unarchiveObjectWithData()` method can't work with empty values, so we check the size of the `NSData` object first with a property provided by the `NSData` class called `length`. If the value returned by this property is greater than 0, it means that there is information encoded that we can decode.

The next step is to create the view controllers to manage the document. In this example we work with only one file, so all the logic is managed by the second view (all the functionality for the initial view at this time is provided by the Segue connected to the button). We call the view controller for the second view `EditionViewController`. Here we have to check the files available, get the one we are interested in, and create the `MyDocument` object to represent it. The following example introduces the properties and the initial statements for this class.

Listing 23-4: Editing the document

```
import UIKit
```

```
class EditionViewController: UIViewController {
    @IBOutlet weak var mycontent: UITextView!
    var document: MyDocument!
    var iCloudURL: NSURL!
    var metaData: NSMetadataQuery!
    override func viewDidLoad() {
        super.viewDidLoad()

        let center = NSNotificationCenter.defaultCenter()
        center.addObserver(self, selector: "metadataReceived:", name: NSMetadataQueryDidFinishGatheringNotification,
            object: nil)

        metaData = NSMetadataQuery()
        metaData.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope]
        metaData.startQuery()
    }
}
```

Because of the logic of our application, we need a property to store the reference to the **MyDocument** object we are going to use to access the file, a property to store the file's URL in iCloud, and a property to reference the **NSMetadataQuery** object that will let us search for the files available. In Listing 23-4, we call these properties **document**, **iCloudURL**, and **metaData**.

The first thing we have to do is to check whether the file exists or not and get its metadata. In Listing 23-4, we initialize the **NSMetadataQuery** object as soon as the view is loaded and start the process with the **startQuery()** method. The query was configured to look into the iCloud's Documents directory because this is the recommended directory for our files, but adding other constants to the array assigned to the **searchScopes** property, we could expand the search.

Once the query begins, the results are reported by the **NSMetadataQuery** object through notifications. Because these notifications can be posted right after the process begins, we have to register the observers before the **startQuery()** method is called. The only notification we care about in our example is the **NSMetadataQueryDidFinishGatheringNotification**, posted when the process of gathering data is over. In the method assigned to this notification, we have to create the **MyDocument** object and connect it to the file (an existing one or a new one if no file was found). The following is the implementation of the **metadataReceived()** method assigned to the observer in our example.

Listing 23-5: Processing query results

```
func metadataReceived(notification: NSNotification) {  
    metaData.stopQuery()  
  
    let center = NSNotificationCenter.defaultCenter()  
    center.removeObserver(self, name: NSMetadataQueryDidFinishGatheringNotification, object: nil)  
  
    if metaData.resultCount > 0 {  
        let results = metaData.results as! [NSMetadataItem]  
        let file = results[0]  
        let fileURL = file.valueForAttribute(NSMetadataItemURLKey) as! NSURL  
        document = MyDocument(fileURL: fileURL)  
        document.openWithCompletionHandler({ (success: Bool) in  
            if success {  
                self.mycontent.text = self.document.fileContent  
                self.iCloudURL = fileURL  
            }  
        })  
    } else {  
        let manager = NSFileManager.defaultManager()  
        if let fileURL = manager.URLForUbiquityContainerIdentifier(nil) {  
            iCloudURL = fileURL.URLByAppendingPathComponent("Documents")  
        }  
    }  
}
```

```

iCloudURL = iCloudURL.URLByAppendingPathComponent("myfile.dat")
document = MyDocument(fileURL: iCloudURL)
document.saveToURL(iCloudURL, forSaveOperation: .ForCreating, completionHandler: nil)
}
}
}

```

The **NSMetadataQuery** object generates a live process. It keeps querying iCloud for new information until we tell it to make a pause or stop. In our example we just want to know if our file exists, so we stop the query all together with the **stopQuery()** method. Because we are not going to listen for more data, it doesn't make sense to keep observing the **NSMetadataQuery** notification, so we remove the observer as well. The next step is to load the file if it already exists or create a new one. For better performance, it is recommended to read the **resultCount** property of the query instead of counting the elements of the **results** array. If this property returns a value greater than 0, we get the first element in the **results** array (we are only interested in one file) and create the **MyDocument** object with its URL. This tells the **MyDocument** object what is the URL of its file, but we have to call the **openWithCompletionHandler()** method next to open it. This method asks to the instance of **MyDocument** to load the content of the file. The instance then executes its **loadFromContents()** method and assigns the content to its **fileContent** property. In the completion handler we assign the value of this property to the Text View and the text is shown on the screen.

On the other hand, if the file doesn't exist, we have to generate the URL and create a new one. In this case we take advantage of a method provided by the **NSFileManager** class to get iCloud locations called **URLForUbiquityContainerIdentifier()**. The method returns the URL of the location of our app's iCloud container. Because an app can have several containers, the method takes an attribute to specify the container's identifier (declared as **nil** to use the container by default). The URL returned by this method is the root directory, so we have to add the Documents directory and the file's name to create the file. With the final URL, we create a **MyDocument** object and call its **saveToURL()** method to create the file. Internally, the **MyDocument** instance calls its **contentsForType()** method to get the content and then it creates the file.

The codes added so far to the **EditionViewController** class load or create a file and show its content on the screen, but we still have to store the changes when the document is saved. The following is the Action method for the Save button. In this method we get the text from the Text View, assign it to the **fileContent** property of the **MyDocument** object, and then call its **saveToURL()** method to save it on file. The operation assigned to the method is **ForOverwriting** because at this point we already know that there is a file available and we just need to update its content.

Listing 23-6: Saving the document

```

@IBAction func saveDocument(sender: UIBarButtonItem) {

```

```

document.fileContent = mycontent.text
document.saveToURL(iCloudURL, forSaveOperation: .ForOverwriting, completionHandler: nil)
navigationController?.popViewControllerAnimated(true)
}

```

Do It Yourself: Create a new file with a subclass of `UIDocument` called `MyDocument` and complete the class with the code of Listing 23-3. Create a subclass of `UIViewController` called `EditionViewController` and assign it to the second view. Connect the Text View with an Outlet called `mycontent`. Complete the class with the code of Listing 23-4 and the method of Listing 23-5. Connect the Save button with an action called `saveDocument()` and complete the method with the code of Listing 23-6. Run the application in the simulator. Press the Edit My Document button, insert some text in the Text View and press the Save button. Run the app on a device and press the Edit My Document button again. You should see on the screen the text inserted before in the simulator. Remember that the updates from iCloud take time to download and that most of the time you have to select the Trigger iCloud Sync from the Debug menu to synchronize the simulator.

In the previous example we searched for every file in the iCloud’s container, but we can make the query more selective using predicates. For example, if we want to make sure that the query returns the file `myfile.dat` used in our application, we can assign an `NSPredicate` object to the `NSMetadataQuery` object, as shown in the following example.

Listing 23-7: Filtering the query

```

override func viewDidLoad() {
    super.viewDidLoad()

    let center = NSNotificationCenter.defaultCenter()
    center.addObserver(self, selector: "metadataReceived:", name: NSMetadataQueryDidFinishGatheringNotification,
        object: nil)

    metaData = NSMetadataQuery()
    metaData.predicate = NSPredicate(format: "%K == %@", NSMetadataItemFSNameKey, "myfile.dat")
    metaData.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope]
    metaData.startQuery()
}

```

The format of the predicate for a query requires comparing the value of a file’s attribute with the string we are looking for. The attribute is represented by the constants defined in the `NSMetadataItem` class (described above). In Listing 23-7 we use the `NSMetadataItemFSNameKey` constant to filter the files by name (the value of the constant has

to be introduced to the string with the %K placeholder to avoid the quotes). The query now returns only files with the name myfile.dat.

Do It Yourself: Update the `viewDidLoad()` method of the `EditionViewController` class with the code of Listing 23-7. Run the application. The Text View should show the content of the myfile.dat file. Try to change the name of the file in the predicate and run the application again.

Multiple Documents

The initial view of our previous example offers a single button to open a document and show the content on the screen, however most applications allow the user to create and manage all the documents they need. As we mentioned before, an `NSMetadataQuery` object generates a live query. The query keeps looking for documents until we tell it to stop, and this is how we work with multiple documents in iCloud. We have to create a query and keep it running to update the list of documents available every time new information comes in.

Multiple documents conform a list of items, and the best way to present this type of information is with a table or a collection view. To illustrate how the process works, we replaced the initial view with a Table View Controller, as illustrated by Figure 23-7.

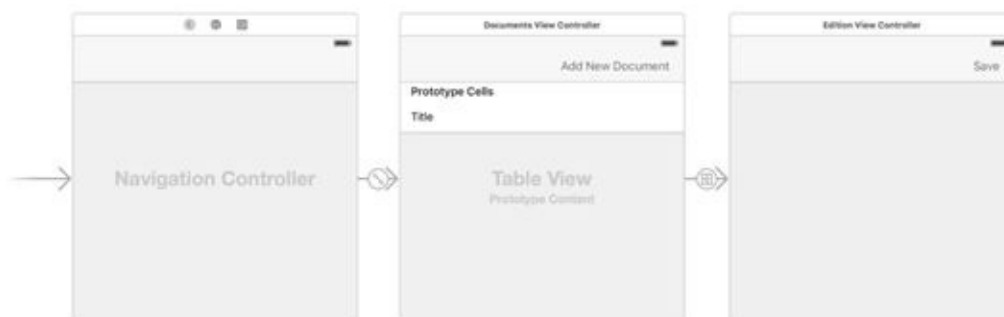


Figure 23-7: Interface to work with multiple documents

Do It Yourself: Select and delete the initial view of the interface introduced in Figure 23-6. Add a Table View Controller in its place. Connect the Navigation Controller to the Table View Controller with a Root View Controller Segue. Connect the cell to the second view with a Show Segue and give it the identifier “showDocument”. Add a bar button to the Table View Controller with the title Add New Document. Select the cell, change its style to Basic and give it the identifier “documentCell”.

The Table View Controller includes a bar button to add new documents and a Basic cell to display the name of the documents. The view controller for this view has to create a query to get the documents available and then show their names in the table. We called this class **DocumentsViewController**. The following example defines the properties we need to manage the data and configures the query.

Listing 23-8: Initiating the query

```
import UIKit

class DocumentsViewController: UITableViewController {
    var documentsList: [String] = []
    var metaData: NSMetadataQuery!
    override func viewDidLoad() {
        super.viewDidLoad()

        let center = NSNotificationCenter.defaultCenter()
        center.addObserver(self, selector: "updateList:", name: NSMetadataQueryDidFinishGatheringNotification,
object: nil)
        center.addObserver(self, selector: "updateList:", name: NSMetadataQueryDidUpdateNotification, object: nil)
        metaData = NSMetadataQuery()
        metaData.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope]
        metaData.startQuery()
    }
}
```

Because we are not only getting the list of documents available but also monitoring iCloud for changes, the view controller for this view has to register observers for two notifications: the **NSMetadataQueryDidFinishGatheringNotification** to know when the process of gathering the initial data is over, and the **NSMetadataQueryDidUpdateNotification** to know when there are updates available. In Listing 23-8 we register these observers and assign them the **updateList()** method. The method is in charge of getting the results returned by the query, updating the list of values, and refreshing the table.

Listing 23-9: Keeping the table updated

```
func updateList(notification: NSNotification) {
    metaData.disableUpdates()
    for item in metaData.results {
        let name = item.valueForAttribute(NSMetadataItemFSNameKey) as! String
        if documentsList.indexOf(name) == nil {
            documentsList.append(name)
        }
    }
}
```

```

}
documentsList.sortInPlace({ (value1, value2) in value1 < value2 })
tableView.reloadData()
metaData.enableUpdates()
}

```

Every time the query gets information, it posts notifications to let our code know what happened. To avoid conflicts with the data we are processing and new data received, we have to pause the query until the process is over. This is done by the **disableUpdates()** and **enableUpdates()** methods of the **NSMetadataQuery** class. In the method of Listing 23-9, we call this method at the beginning and the end to be sure that the **updateList()** method is not executed again until we get the new information and refresh the table.

Usually an application like this gets the values it needs from the results returned by the query and stores them in a place the rest of the class can read, such as a property, a model, or a more complex storage system like Core Data. For this example we defined a simple property called **documentsList** to contain an array with the names of the documents. Because the **updateList()** method is called when the query finishes gathering the data and also when it finds changes, we have to check for duplicates before introducing a new name into this array. The code in Listing 23-9 creates a loop to go through every new value, gets the attribute for the name, and checks if it already exists with the **indexOf()** method. Comparing the value returned by this method to **nil** we can be sure that there won't be any duplicates on the list. After the list is updated, the table is refreshed. The Table View methods required to show this information are simple, we just have to get the values from the **documentsList** and create the cells.

Listing 23-10: Populating the table

```

override func tableView(tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {
    return documentsList.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("documentCell", forIndexPath: indexPath)
    cell.textLabel?.text = documentsList[indexPath.row]
    return cell
}

```

We have connected the prototype cell on the table to the second view with a Show Segue. This is so every time the user selects a cell, the content of the document is shown on the screen. For this to work, we need to implement the **prepareForSegue()** to tell the second view which is the selected document. The following example assigns the document's name to a property in the **EditionViewController** class called **selected** (we will define

this property later).

Listing 23-11: Sending the name of the selected document to the second view

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?){
    if segue.identifier == "showDocument" {
        let controller = segue.destinationViewController as! EditionViewController
        if let path = tableView.indexPathForSelectedRow {
            controller.selected = documentsList[path.row]
        }
    }
}
```

The `DocumentsViewController` class is not complete until we include the code necessary for the user to create new documents. To simplify this example, we decided to create an Alert View with a Text Field to insert the name of the document. The following example defines the Action for the Add New Document button and a method called `saveFile()` to create the new document and store it in iCloud.

Listing 23-12: Creating new documents

```
@IBAction func addDocument(sender: UIBarButtonItem) {
    let alert = UIAlertController(title: "New File", message: nil, preferredStyle: .Alert)
    let cancel = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
    alert.addAction(cancel)
    let action = UIAlertAction(title: "Create", style: .Default, handler: { (action) in
        if let fields = alert.textFields {
            let file = fields[0].text
            self.saveFile(file!)
        }
    })
    alert.addAction(action)
    alert.addTextFieldWithConfigurationHandler({ (textField) in
        textField.placeholder = "Insert name and extension"
    })
    presentViewController(alert, animated: true, completion: nil)
}

func saveFile(name: String) {
    if documentsList.indexOf(name) == nil {
        let manager = NSFileManager.defaultManager()
        if let fileURL = manager.URLForUbiquityContainerIdentifier(nil) {
            var iCloudURL = fileURL.URLByAppendingPathComponent("Documents")
```

```

iCloudURL = iCloudURL.URLByAppendingPathComponent(name)
let document = MyDocument(fileURL: iCloudURL)
document.saveToURL(iCloudURL, forSaveOperation: .ForCreating, completionHandler: nil)
}
}
}

```

The code of Listing 23-12 creates an Alert View with a Text Field and a button with the title Create. When this button is pressed, the value of the Text Field is assigned to the **file** constant and sent to the **saveFile()** method. This method takes the new name, checks if it already exists in the **documentsList** array, and if it doesn't, it creates the new document.

The **MyDocument** object we create in this method is just a temporary object we use to save the file in iCloud. The **DocumentsViewController** class doesn't really manage any document, all it does is get the list of documents available and show them on the screen, but creating a new **MyDocument** object we get the file stored in iCloud and therefore the query finds it right away and the table is updated with the new value.

Do It Yourself: Create a subclass of **UITableViewController** called **DocumentsViewController** and assign it to the Table View Controller added to replace the initial view. Update the class with the code of Listing 23-8. Add the methods of Listings 23-9, 23-10, 23-11 and 23-12 to the class. Connect the Add New Document button to the Action **addDocument()**.

The code for the **EditionViewController** class has been simplified. We don't have to create a new document from this view controller anymore, we just have to read the selected document, show it on the screen, and save the changes introduced by the user.

Listing 23-13: *Sending the name of the selected document to the second view*

```

import UIKit

class EditionViewController: UIViewController {
    @IBOutlet weak var mycontent: UITextView!
    var document: MyDocument!
    var iCloudURL: NSURL!
    var metaData: NSMetadataQuery!
    var selected: String!
    override func viewDidLoad() {
        super.viewDidLoad()

        let center = NSNotificationCenter.defaultCenter()
    }
}

```

```

        center.addObserver(self, selector: "metadataReceived:", name: NSMetadataQueryDidFinishGatheringNotification,
object: nil)

        metaData = NSMetadataQuery()
        metaData.predicate = NSPredicate(format: "%K == %@", NSMetadataItemFSNameKey, selected)
        metaData.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope]
        metaData.startQuery()
    }

    func metadataReceived(notification: NSNotification) {
        metaData.stopQuery()

        let center = NSNotificationCenter.defaultCenter()
        center.removeObserver(self, name: NSMetadataQueryDidFinishGatheringNotification, object: nil)

        if metaData.resultCount > 0 {
            let results = metaData.results as! [NSMetadataItem]
            let file = results[0]
            let fileURL = file.valueForAttribute(NSMetadataItemURLKey) as! NSURL
            document = MyDocument(fileURL: fileURL)
            document.openWithCompletionHandler({ (success: Bool) in
                if success {
                    self.mycontent.text = self.document.fileContent
                    self.iCloudURL = fileURL
                }
            })
        }
    }

    @IBAction func saveDocument(sender: UIBarButtonItem) {
        document.fileContent = mycontent.text
        document.saveToURL(iCloudURL, forSaveOperation: .ForOverwriting, completionHandler: nil)
        navigationController?.pushViewControllerAnimated(true)
    }
}

```

The query for this view controller now contains a predicate to search the selected document, but the rest of the code is the same as before. The only difference is that now instead of working with a unique file, we can process every file on the list.

Do It Yourself: Update the `EditionViewController` class with the code of Listing 23-13. Run the application in the simulator. Press the Add New Document button. Insert the name and extension of the document you want to create and press Create. After a moment, the new document should be added to the table. Run the application on a device. The document you just created in the simulator should appear on the device's

screen. Add another document in the simulator and select the Trigger iCloud Sync option in the Debug menu at the top of the screen to make sure the simulator is synchronized with iCloud. The document you just added should appear on the device's screen after a few seconds. Select a document in the simulator, insert some text, and press the Save button. After a moment, the document should contain the same text on the device.

IMPORTANT: In these examples we have stored only a single string in the documents, but we can archive all the values we want, including custom classes, as we did in the examples of Chapter 21. When working with documents that manage content of several files, like text files and image files, instead of encoding all together in one file it is recommended to use a file wrapper for better performance. The wrapper is created with objects of the `NSFileWrapper` class. If the content of the document is created with a wrapper, iCloud is able to download and upload only the files inside the wrapper that present changes, reducing bandwidth usage and improving speed. For more information, visit our website and follow the links for this chapter.

22.4 Core Data in iCloud

Core Data uses the iCloud Documents service to upload the Persistent Store to iCloud and make the information available on all of user's devices. The two systems are highly integrated, and only a few changes are necessary to get them working together. In fact, the connection is established by simply assigning the following option to the Persistent Store.

NSPersistentStoreUbiquitousContentNameKey—This option assigns a name to the Persistent Store in iCloud. It requires a `String` value with the name that the Persistent Store will have in the iCloud's container.

Creating the Persistent Store with this option automatically connects our Core Data stack with iCloud and everything we do in the Persistent Store from that moment on will be available in any of the user's devices running the application. However, there are still a few problems we have to solve. If the user logs out of iCloud or changes accounts, Core Data uses a temporal local Persistent Store to store the objects until iCloud is available again, which could create inconsistencies between the Persistent Store and the context. Also, when new values are added to Core Data from iCloud they are assigned to the Persistent Store but not to the context we are working with, making it necessary to merge the data in our context with the data in the store every time a change occurs. To report these states and let our code respond accordingly, the Persistent Store Coordinator posts the following notifications.

NSPersistentStoreCoordinatorStoresWillChangeNotification—This notification is posted when the new Persistent Store is about to be created, either because the application is executed for the first time or the user changed his or her iCloud account.

NSPersistentStoreCoordinatorStoresDidChangeNotification—This notification is posted when the process of migrating the Persistent Store from the old iCloud account to the new one is over.

NSPersistentStoreDidImportUbiquitousContentChangesNotification—This notification is posted when changes are detected in the iCloud's Persistent Store and our app needs to merge them with the data in the context.

The following example defines a Core Data stack that works with iCloud. The implementation is almost the same as before, except this time the class includes the option to give the Persistent Store a name in the iCloud's container and observers to respond to the three iCloud's notifications.

Listing 23-14: Defining a Core Data stack for iCloud

```
import Foundation
import CoreData

class CoreDataStack: NSObject {
    var context: NSManagedObjectContext!

    override init() {
        super.init()

        let bundle = NSBundle.mainBundle()
        let modelURL = bundle.URLForResource("books", withExtension: "momd")
        let manager = NSFileManager.defaultManager()
        let urls = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let url = urls.last!
        let localURL = url.URLByAppendingPathComponent("store.sqlite")
        let model = NSManagedObjectModel(contentsOfURL: modelURL!)
        let coordinator = NSPersistentStoreCoordinator(managedObjectModel: model!)
        let center = NSNotificationCenter.defaultCenter()
        center.addObserver(self, selector: "storeWillChange:", name:
NSPersistentStoreCoordinatorStoresWillChangeNotification, object: coordinator)
        center.addObserver(self, selector: "storeDidChange:", name:
NSPersistentStoreCoordinatorStoresDidChangeNotification, object: coordinator)
        center.addObserver(self, selector: "contentChange:", name:
NSPersistentStoreDidImportUbiquitousContentChangesNotification, object: coordinator)
        let options = [NSPersistentStoreUbiquitousContentNameKey: "BooksStore"]
        try! coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: localURL,
options: options)

        context = NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)
        context.persistentStoreCoordinator = coordinator
    }
}
```

The stack in Listing 23-14 is ready to work with iCloud, but we still have to define the methods specified for the observers. These methods have specific tasks to perform. The `storeWillChange()` method has the responsibility to make sure that any change in the context is moved to the Persistent Store before the system removes the current store to create a new one. The `storeDidChange()` method has to report to the rest of the application that there is a new store available, and the `contentChange()` method has to merge the context with the changes introduced to the Persistent Store in iCloud. The following example shows possible implementations.

Listing 23-15: Responding to iCloud’s notifications

```
func storeWillChange(notification: NSNotification) {
    context.performBlock({
        if self.context.hasChanges {
            do {
                try self.context.save()
            } catch {
                print("Error")
            }
        } else {
            self.context.reset()
        }
    })
}

func storeDidChange(notification: NSNotification) {
    updateUI()
}

func contentChange(notification: NSNotification) {
    context.performBlock({
        self.context.mergeChangesFromContextDidSaveNotification(notification)
        self.updateUI()
    })
}

func updateUI() {
    let mainQueue = NSOperationQueue.mainQueue()
    mainQueue.addOperationWithBlock({
        let center = NSNotificationCenter.defaultCenter()
        center.postNotificationName("updateUI", object: nil)
    })
}
```

The methods in Listing 23-15 present a standard implementation. This is what Apple recommends to do when these methods are called. The first one, `storeWillChange()`, checks if there are changes available in the context with the `hasChanges` method and saves the context with the `save()` method or resets it with the `reset()` method depending on the case. Because the situation in which this method is executed is critical and can compromise the integrity of the Persistent Store, we have to define the statements inside a closure and execute the closure with a method provided by the `NSManagedObjectContext` class called `performBlock()`. The `performBlock()` method makes sure that the closure is executed in the same queue where the context was created, avoiding any possible conflict between the context and the

current data.

The `storeDidChange()` method has a simpler responsibility. This method is called when the process of creating or migrating the store is over, so it is used to update or activate the user’s interface. In this example we call the `updateUI()` method to post a notification and let the rest of the application know that the Persistent Store is ready.

Finally, the `contentChange()` method uses the `performBlock()` method again to call the context’s method `mergeChangesFromContextDidSaveNotification()` to merge the changes in the Persistent Store with the context.

The `updateUI()` method at the end is just a custom method that posts a notification that other parts of the application can read to know when they have to update their own data and the interface. This method is called from the `storeDidChange()` method to update the interface after the store is created or the migration is over, and also from the `contentChange()` method after the data in the context and the Persistent Store are merged to reflect the changes in the interface immediately.

Because the `contentChange()` method calls the `updateUI()` method from a closure executed by the `performBlock()` method, we have created a new operation to make sure the statements in `updateUI()` are executed in the main queue (see Operation Queues in Chapter 20).

Do It Yourself: Create a new Single View Application project. Create a Swift file called `CoreDataStack.swift` for the `CoreDataStack` class of Listing 23-14. Add the methods of Listing 23-15 at the end of the class.

As an example, we have created an application that stores and lists books using Core Data and iCloud. The interface needs a Table View with a Basic cell embedded in a Navigation Controller. The Table View Controller includes a button called Add New Book to add new books.

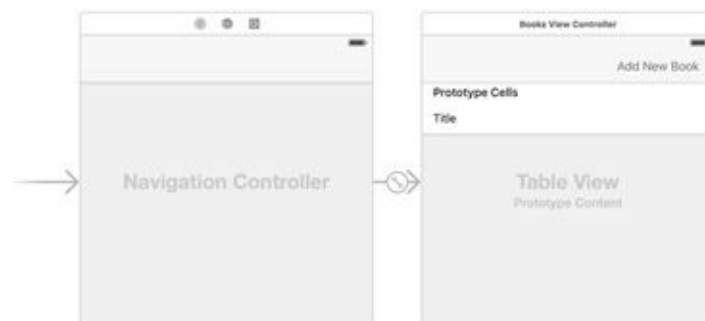


Figure 23-8: Interface to display books stored using Core Data and iCloud

We also need a Core Data model with an Entity called “Books” that contains an attribute of type `String` called “title”, as illustrated in Figure 23-9.



Figure 23-9: Simple Core Data model to store books

Do It Yourself: Select the option at the top of the Navigation Area to open the app’s settings. Open the Capabilities panel, turn on the iCloud switch and select the options Key-value storage and iCloud Documents to activate iCloud for your app (Figures 23-1 and 23-3). Remove the initial view in your project, add a Table View Controller, and embed it in a Navigation Controller. Select the cell, change its style to Basic, and give it the identifier “booksCell”. Add a bar button to the Table View Controller with the title Add New Book (Figure 23-8). Open the File menu at the top of the screen and select the option New / File... to create a new file. Select Core Data and double click the Data Model icon. Name the file books.xcdatamodeld. Select this new file and in the Editor Area add an entity called “Books” with an attribute of type **String** called “title”, as shown in Figure 23-9.

As always, it is recommended to create the stack’s instance in the app’s delegate. We should also implement the delegate’s **applicationWillTerminate(application)** method to save the context before the application is terminated. This is a good practice when working with iCloud because of the multiple circumstances in which the context may be modified without our app being notified or having enough time to respond. The implementation may look like the following.

Listing 23-16: Preparing the app’s delegate to work with iCloud

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    var stack: CoreDataStack!

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        stack = CoreDataStack()
        return true
    }

    func applicationWillTerminate(application: UIApplication) {
```

```

if stack.context.hasChanges {
do {
try stack.context.save()
} catch {
print("Error")
}
}
}
}
}

```

The view controller for the Table View Controller is very similar to what we have created before for other applications. We have to get the list of books and create the cells for the table with their titles. We called this view controller **BooksViewController**. The class includes an Action that presents an Alert View to let the user insert new books and adds an observer to listen to the **updateUI** notification posted by the Core Data stack when the interface needs updating (see the **updateUI()** method in Listing 23-15).

Listing 23-17: Listing books and updating the interface

```

import UIKit
import CoreData

class BooksViewController: UITableViewController {
    var context: NSManagedObjectContext!
    var listOfBooks: [NSManagedObject] = []

    override func viewDidLoad() {
        super.viewDidLoad()

        let app = UIApplication.sharedApplication()
        let appDelegate = app.delegate as! AppDelegate
        context = appDelegate.stack.context
        let center = NSNotificationCenter.defaultCenter()
        center.addObserver(self, selector: "fetchData", name: "updateUI", object: nil)
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        fetchData()
    }

    func fetchData() {
        let request = NSFetchRequest(entityName: "Books")
        let sort = NSSortDescriptor(key: "title", ascending: true, selector: "caseInsensitiveCompare:")
        request.sortDescriptors = [sort]
    }
}

```

```

    if let results = try? context.executeFetchRequest(request) {
        listOfBooks = results as! [NSObject]
        tableView.reloadData()
    }
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return listOfBooks.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("booksCell", forIndexPath: indexPath)
    let book = listOfBooks[indexPath.row]
    cell.textLabel?.text = book.valueForKey("title") as? String
    return cell
}

@IBAction func addBook(sender: UIBarButtonItem) {
    let alert = UIAlertController(title: "New Book", message: nil, preferredStyle: .Alert)
    let cancel = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
    alert.addAction(cancel)
    let action = UIAlertAction(title: "Save", style: .Default, handler: { (action) in
        if let fields = alert.textFields {
            let title = fields[0].text
            self.saveBook(title!)
        }
    })
    alert.addAction(action)
    alert.addTextFieldWithConfigurationHandler({ (textField) in
        textField.placeholder = "Insert title"
    })
    presentViewController(alert, animated: true, completion: nil)
}

func saveBook(title: String) {
    let newBook = NSEntityDescription.insertNewObjectForEntityForName("Books", inManagedObjectContext:
context)
    newBook.setValue(title, forKey: "title")
    do {
        try context.save()
    } catch {
        print("Error")
    }
}

```

}

The observer for the `updateUI` notification calls the `fetchData()` method every time the notification is posted. In this method we update the `listOfBooks` array we use to store all the existent `NSManagedObject` objects and refresh the table. The method is also called when the view is about to appear (`viewWillAppear()`) and when a new object is added to the Persistent Store (`saveBook()`), therefore the table is updated when local or remote changes are introduced.

With this last code, the application is ready. To test it, we need to run it in two different devices or in a device and the simulator with iCloud activated, as we explained before (Figure 23-4). When the application is executed in any of the devices, the system creates a temporary local store and posts the message “Using local storage: 1” in the Xcode’s console. Next, it makes an attempt to connect to iCloud and move the store to the cloud. If successful, it posts the message “Using local storage: 0” in the console. This last message means that the store is now on iCloud and every change we make can be read by instances of our application running in other devices (in our example, we can only add new books). Figure 23-10 shows what the application looks like in the simulator.

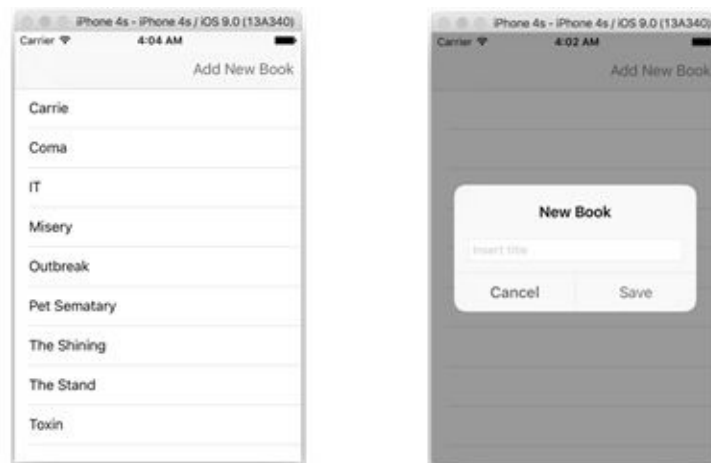


Figure 23-10: Application to store books in iCloud

Do It Yourself: Update the `AppDelegate` class with the code of Listing 23-16. Create a `UITableViewController` subclass called `BooksViewController`, assign it to the Table View Controller, and complete the class with the code of Listing 23-17. Connect the Add New Book button to the `addBook()` Action. Run the application in the simulator. You should see the two messages in the console while the app is connecting to iCloud. Add a new book. Run the application again on your device. You should see on the screen the book you just inserted. Try to add more books in the simulator. You should see them appear on the device’s screen after a few seconds. Remember to select the option Trigger iCloud Sync in the simulator’s menu every time the system doesn’t respond to a change.

Seeding the Store

The action of storing initial data in the store is called *seeding*. If we want to provide basic information for the user that will be available as soon as the application is installed, we have to seed the Persistent Store. There are different ways to do it: we can load the information from a file, create a Persistent Store during development, and then migrate it to the user's local or iCloud store, or we can just create the content one object at a time from code, as we do with any other `NSManagedObject` object. The method we choose will depend on the characteristics of our application and the amount of data we want to seed, but what is a constant among implementations is that we have to do it only once. Once the data is loaded into the Persistent Store we have to store a value somewhere that the app can read to not do it again otherwise we will create new objects every time the app is executed. In local storage we usually used User Settings (see Chapter 21) to store this value, but with iCloud we have to do it using the Key/Value Storage system, so every instance of our app can read the value and decide what to do.

In the following example we read a Boolean value from the Key/Value Storage system after the Core Data stack is initialized to know if we have to load the initial data or it was already loaded. The code checks whether a value identified with the key “seeded” exists and, if it doesn't, it calls the `seedCoreData()` method to load the initial information into the Persistent Store. After the process is over, the value of “seeded” is changed to `true` so the next time the app is launched the data won't be loaded again.

Listing 23-18: Seeding the store

```
override init() {  
    super.init()  
  
    let bundle = NSBundle.mainBundle()  
    let modelURL = bundle.URLForResource("books", withExtension: "momd")  
    let manager = NSFileManager.defaultManager()  
    let urls = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)  
    let url = urls.last!  
    let localURL = url.URLByAppendingPathComponent("store.sqlite")  
  
    let model = NSManagedObjectModel(contentsOfURL: modelURL!)  
    let coordinator = NSPersistentStoreCoordinator(managedObjectModel: model!)  
    let center = NSNotificationCenter.defaultCenter()  
    center.addObserver(self, selector: "storeWillChange:", name:  
        NSPersistentStoreCoordinatorStoresWillChangeNotification, object: coordinator)  
    center.addObserver(self, selector: "storeDidChange:", name:  
        NSPersistentStoreCoordinatorStoresDidChangeNotification, object: coordinator)
```

```

center.addObserver(self, selector: "contentChange:", name:
NSPersistentStoreDidImportUbiquitousContentChangesNotification, object: coordinator)

let options = [NSPersistentStoreUbiquitousContentNameKey: "BooksStore"]

try! coordinator.addPersistentStoreWithType(NSSQLiteStoreType, configuration: nil, URL: localURL, options:
options)

context = NSManagedObjectContext(concurrencyType: .MainQueueConcurrencyType)

context.persistentStoreCoordinator = coordinator


let kvStorage = NSUbiquitousKeyValueStore()
let seeded = kvStorage.boolForKey("seeded")
if !seeded {
    seedCoreData()
}
}

func seedCoreData() {
    let values = ["Carrie", "Coma", "The Shining", "Pet Sematary", "IT"]
    for value in values {
        let newBook = NSEntityDescription.insertNewObjectForEntityForName("Books", inManagedObjectContext:
context)
        newBook.setValue(value, forKey: "title")
    }
    do {
        try context.save()
        let kvStorage = NSUbiquitousKeyValueStore()
        kvStorage.setBool(true, forKey: "seeded")
    } catch {
        print("Error")
    }
}
}

```

The `seedCoreData()` method generates an array with the values we want to provide to the user and then creates the `NSManagedObjectContext` objects one by one in a `for in` loop.

Do It Yourself: Replace the `init()` method in the `CoreDataStack` class created in Listings 23-14 and 23-15 with the method of Listing 23-18, and add the `seedCoreData()` method of Listing 23-18 at the end of the class. Run the application in the simulator. You will see the values inserted before along with the values generated by the `seedCoreData()` method. Stop the app and run it again. The value of the "seeded" value will be `true` this time, so you won't see any new value added to the list.

The Basics: iCloud applications store their information online. If you delete the

application in the device or the simulator, the data is not deleted and is loaded from the cloud when the application is installed again. If you want to delete everything and start from scratch, you have to remove the information on iCloud. The easiest way to do it is to open the Apple menu at the top-left corner of your computer's screen (the Apple icon) and select the option System Preferences... Click on the iCloud icon and then click on the Manage... button at the bottom of the window. You will see a list of the applications that are using iCloud at the moment. Select your application and press the Delete Documents and Data button to delete all the information. Wait a few minutes for the changes to propagate and install the application again (remember to uninstall the application in every device before following these steps).

Duplicates

When working with a system like iCloud it is normal to find data inconsistencies. The most common scenario is the creation of duplicates. This may occur when the user creates objects with the same information in two different devices and at least one of them is offline. In cases like this, the application can't detect if the object already exists and creates a new one. When the device is back online, the application uploads the object to iCloud and the Persistent Store ends up containing two objects with the same information (two objects representing the same book, for example). This inconsistency can't be detected by Core Data and therefore it is our code's responsibility to solve it.

Detecting duplicates demands certain preparation. We have to find an attribute in the entity that can help us identify the duplicated data. For example, if we want to remove duplicates associated with an entity used to store information of employees, we can use their Social Security Numbers. These numbers are unique, so if we find two employees with the same number, we can be sure that one of them is a duplicate and not a real employee. We can do the same with emails, for example. If we have an entity to store users accounts and our application doesn't allow the creation of two accounts with the same email, we can compare emails to know when an object is a duplicate. But not every entity contains an attribute that has a unique value. For example, two books may be written by the same author, have the same title, but be published in a different year (two different editions). This means that we can't compare these values alone because more than one book may have the same title, author, or year. In cases like this we need to add a specific attribute to the entity to be able to store a value that comprises all this information (e.g., a string containing the title, the author, and the year) and use that value to compare objects.

Once we decide what is the attribute we are going to compare to detect duplicates, we have to create a method that counts the number of objects that contain the same value for that attribute and erase all the objects except the one we establish to be the original. There are several steps we have to follow to achieve this. First we have to create an expression to count the objects inside the Persistent Store. An expression is a side of the condition

established by a predicate. When a predicate is created, every part of the condition is represented by an expression. For instance, a predicate with the format `name = "John"` is translated as two expressions, one for the `name` property and another for the value `"John"`. The format of a predicate is actually a shortcut for the creation of expressions, but we can create expressions independently using a class defined by Foundation called `NSExpression`. The class cannot only create simple expressions to represent values and properties, but also perform operations, and that is what makes it useful. The following is the initializer to create an `NSExpression` object.

`NSExpression(format: format, arguments)`—This initializer creates an `NSExpression` object with the configuration specified by the **`format`** attribute. The **`arguments`** attribute is an optional list of values separated by comma that replace placeholders in the string assigned to the **`format`** attribute.

As we mentioned, besides representing properties and values, expressions can also be used to perform operations. The class accepts the functions `sum`, `count`, `min`, `max`, and `average`. Taking advantage of these functions, we can ask the request to compare or calculate values of a specific property and return the result. For example, we may have `NSManagedObject` objects that contain a property to store the salary of every employee. Using the `max` function we can retrieve the highest salary. To find duplicates, we use the `count` function to count the number of objects that contain the same value.

The attributes of an entity are represented internally by an object of the `NSPropertyDescription` class, so we can combine expressions with other types of values to perform a request. But expressions can't be used directly. To include expressions in a request, we have to prepare a description that will tell the request the name that it has to use to identify the result produced by the expression and the type of value it returns. Foundation defines a subclass of the `NSPropertyDescription` class called `NSExpressionDescription` to create this description. The class includes the following properties.

`name`—This property sets or returns a string used to identify the values returned by the expression.

`expression`—This property sets or returns the `NSExpression` object used to create the expression.

`expressionResultType`—This property sets or returns a value that determines the type of value returned by the expression. It is an enumeration of type `NSAttributeType` with values that correspond to every data type available in Core Data: `Integer16AttributeType`, `Integer32AttributeType`, `Integer64AttributeType`, `DecimalAttributeType`, `DoubleAttributeType`, `FloatAttributeType`, `StringAttributeType`, `BooleanAttributeType`, `DateAttributeType`, `BinaryDataAttributeType`, `TransformableAttributeType`, and `ObjectIDAttributeType`.

The following example introduces the first part of a custom method that finds and removes duplicates in the Persistent Store. We call it `removeDuplicates()`. The method creates an expression that counts the number of objects using the `title` property. The expression is later added to a description that gives the expression the name “counter” and sets the type of the value returned to `Int64`.

Listing 23-19: Creating an expression to count objects

```
func removeDuplicates() {  
    let expression = NSEExpression(format: “count:(title)”)  
    let counter = NSEExpressionDescription()  
    counter.name = “counter”  
    counter.expression = expression  
    counter.expressionResultType = .Integer64AttributeType  
}
```

Now that we have an expression and its description we can create a request to find duplicates. This time we don’t have to get back a list of `NSManagedObject` objects, all we need are two values: the value of the property we are using to detect duplicates (in our example the title of the book) and the number of objects that contain that value. For example, if we have two objects with the title “The Shining”, the request has to return a dictionary with an item containing the values “The Shining” and 2. The following example adds and configures a request to count how many objects contain the same title and return a dictionary with all the coincidences found (one item per coincidence).

Listing 23-20: Creating a request with an expression

```
let entity = NSEntityDescription.entityForName(“Books”, inManagedObjectContext: context)  
let attributes = entity!.attributesByName  
let titleAttribute = attributes[“title”]  
  
let request = NSFetchedRequest(entityName: “Books”)  
request.resultType = .DictionaryResultType  
request.propertiesToFetch = [titleAttribute!, counter]  
request.propertiesToGroupBy = [titleAttribute!]
```

The code of Listing 23-20 is a continuation of the `removeDuplicates()` method. In Listing 23-19 we created the description for an expression that will return the number of coincidences, so now we have to get a reference to the `title` attribute of the Books entity to be able to get the book’s title as well. The `NSEntityDescription` object contains a property called `attributesByName` that returns a dictionary with references to the objects that represent

the entity's attributes. From this property we get the reference to the `title` attribute and assign it to the `titleAttribute` constant to use it later to configure the request.

The request is initialized as always, but this time we have to change the configuration by default to get the values we need. The first thing we do is to assign the value `DictionaryResultType` to the `resultType` property. This tells the request that we want a dictionary back. Next, we assign an array with the expression and the `title` attribute to the `propertiesToFetch` property. This determines that we want the dictionary to contain the titles of the books and the amount of objects found per value. And finally, we group the duplicates together. This is to avoid getting a dictionary with items for every object. For example, if we have two objects with the title "The Shining", a normal request will return a dictionary with two items with the values "The Shining" and 2, when we only need one. Assigning the reference of the `title` attribute to the `propertiesToGroupBy` property, we tell the request to get a dictionary with only one item per coincidence.

The Basics: The objects that represent the attributes are also instantiated from a subclass of `NSPropertyDescription` (called `NSAttributeDescription`), so we can use them along with the expression to construct the request. Besides the subclasses studied in this chapter, there are others to represent additional Core Data elements. For more information, visit our website and follow the links for this chapter.

The dictionary returned by our request contains items for every title found in the Persistent Store, including those that are found only once. The next step is to execute the request and create a new array containing only the titles that are present two times or more, because those are the duplicated titles we have to eliminate. In the following example we call this array `titleList`.

Listing 23-21: Getting only the duplicated titles

```
var titleList: [String] = []
do {
    let results = try context.executeFetchRequest(request)
    for dict in results {
        let total = dict["counter"] as! Int
        let title = dict["title"] as! String
        if total > 1 {
            titleList.append(title)
        }
    }
} catch {
    print("Error")
}
```

The code of Listing 23-21 executes the request and processes the results with a **for in** loop. The values in the dictionary are identified with the keys “counter” (the expression’s name) and “title” (the attribute’s name). If the value of counter is greater than 1, we add the value of title to the **titleList** array to get a list with only the duplicated titles. At the end of this process, the **titleList** array contains strings that represent the titles that are duplicated in the Persistent Store. Using this list we can now get the **NSManagedObject** objects related to those titles to eliminate the ones that are found to be the duplicates.

Listing 23-22: Getting the duplicated NSManagedObject objects

```
var results: [NSManagedObject] = []
if titleList.count > 0 {
    let request = NSFetchRequest(entityName: “Books”)
    request.predicate = NSPredicate(format: “title IN (%@)”, titleList)
    let sort = NSSortDescriptor(key: “title”, ascending: true)
    request.sortDescriptors = [sort]
    do {
        results = try context.executeFetchRequest(request) as! [NSManagedObject]
    } catch {
        print(“Error”)
    }
}
```

Listing 23-22 creates a new request with a predicate that uses the IN operator to get all the **NSManagedObject** objects with a title that matches any of the strings in the **titleList** array. The objects returned are stored in a new array called **results**. The values are ordered by title, so we can create a loop to decide which are the duplicates and which the originals.

Listing 23-23: Eliminating duplicates

```
var previous: NSManagedObject!
for duplicate in results {
    if previous != nil {
        let duplicateTitle = duplicate.valueForKey(“title”) as! String
        let previousTitle = previous.valueForKey(“title”) as! String
        if duplicateTitle == previousTitle {
            context.deleteObject(previous)
        }
        previous = duplicate
    } else {
        previous = duplicate
    }
}
```

```

    }
}
if context.hasChanges {
    do {
        try context.save()
    } catch {
        print("Error")
    }
}
}

```

Listing 23-23 presents the last piece of code necessary for the `removeDuplicates()` method to be able to remove the duplicates in the Persistent Store. The code uses a `for in` loop to iterate through the `results` array and compare the values of their `title` properties. We store the first object in a variable called `previous` and then compare it with the next object returned by the loop. If the values are equal, we delete the previous object, assign the current object to the previous variable and move on to the next.

Do It Yourself: Add the `removeDuplicates()` method to the `CoreDataStack` class created in the previous section of this chapter. Complete the method with the codes of Listings 23-20, 23-21, 23-22, and 23-23, in that order.

The Basics: In this example we don't compare other values to decide which object to delete, we just delete the previous one, but you should incorporate more attributes to the entity to help you make the decision (e.g., the date the object was created), or compare other values (e.g., the author of the book). When you delete an object, always remember that the object may be connected to others through relationships. If you have a complex model, you have to consider not only what objects to delete but also how deleting the objects will affect your Object Graph.

After we incorporate the `removeDuplicates()` method to the `CoreDataStack` class, we have to call it from the methods that could generate duplicates. These are the `seedCoreData()` method, where an error could cause our app to seed the Persistent Store twice, and the `contentChange()` method, where changes in the iCloud's Persistent Store are downloaded and merged with the context. The following is the new `seedCoreData()` method for our example.

Listing 23-24: Eliminating duplicates after seeding the Persistent Store

```

func seedCoreData() {
    let values = ["Carrie", "Coma", "The Shining", "Pet Sematary", "IT"]
    for value in values {

```

```

    let newBook = NSEntityDescription.insertNewObjectForEntityForName("Books", inManagedObjectContext:
context)
    newBook.setValue(value, forKey: "title")
}
do {
    try context.save()
    let kvStorage = NSUbiquitousKeyValueStore()
    kvStorage.setBool(true, forKey: "seeded")
    removeDuplicates()
} catch {
    print("Error")
}
}

```

And the following is the new **contentChange()** method.

Listing 23-25: Eliminating duplicates after merging the Persistent Store with the context

```

func contentChange(notification: NSNotification) {
    context.performBlock({
        self.context.mergeChangesFromContextDidSaveNotification(notification)
        self.removeDuplicates()
        self.updateUI()
    })
}

```

Do It Yourself: Replace the **seedCoreData()** method in the **CoreDataStack** class with the code of Listing 23-24 and the **contentChange()** method with the code of Listing 23-25. Run the application and insert duplicated values. Our app doesn't control the duplicates locally, but you will see that every time changes are received from iCloud the duplicates are eliminated. This is a didactic example, but always remember that a real application must control duplicates not only coming from iCloud but also when new values are inserted, as we did in the example of Listing 23-17.

Chapter 24

Graphics

24.1 Core Graphics

All the views we have used so far are `UIView` subclasses that include additional code to draw content on the screen. A label is a view that can draw text, a button is a view that can draw its title, a Slider is a view that can draw the horizontal bar and the indicator the user moves to set a new value, and other more complex subclasses like Table Views can draw multiple graphics to represent large pieces of information. All those graphics are created by predefined code included in these subclasses. We can provide some values for configuration, like the color for the text or the shape of the box around a Text Field, for example, but if we want to show custom graphics, we have to define our own subclasses and create the code to draw the graphics ourselves. For this purpose, Apple offers the Core Graphics framework.

We have introduced Core Graphics in Chapter 4. This is an old framework programmed in the C language and developed to provide a two-dimensional drawing engine for Apple systems. In Chapter 4, we presented its data types, like `CGFloat` and `CGRect`, but the framework also includes a powerful low-level drawing API for the creation of graphics known as Quartz 2D. Using the functions provided by Quartz 2D we can generate graphics and send them to a destination like the screen or a printer.

Destination

Quartz 2D can generate graphics for multiple destinations including printers, PDF files, or windows, but in iOS it is often used to create the graphics for `UIView` subclasses or images. The `UIView` subclass is created in a new file and then associated with the view in the Storyboard where the graphics are going to be shown. Figure 24-1 shows the initial view of a Single View Application project with a view inside associated to a `UIView` subclass called `DrawingView`.



Figure 24-1: View associated with a subclass

The `UIView` class includes a method called `drawRect()` that is executed every time the

system needs to draw the view's content on the screen. Overwriting this method in a `UIView` subclass we can produce our own graphics for the view. The following example shows this method in the `DrawingView` subclass we have created for the view of Figure 24-1.

Listing 24-1: *Creating a `UIView` subclass*

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
    }
}
```

Do It Yourself: Create a new Single View Application project. Add an empty view from the Object Library to the initial view (Figure 5-35). Create a new file with a subclass of `UIView` called `DrawingView` and assign it to the view from the Identity Inspector panel, as shown in Figure 24-1. Replace the code in this class by the code of Listing 24-1.

The Basics: The `drawRect()` method receives a `CGRect` value that represents the portion of the view that needs updating. Sometimes this value may be equal to the view's boundaries, but that is not always the case. If you need to know the size of the view to calculate the coordinates for your graphics, you can get the width and height from the view's `bounds` property (e.g., `self.bounds.size.height`).

Context

Because Quartz 2D can draw on multiple surfaces, the first thing we have to do to create our own graphics is to define the context in which the graphics are going to be drawn. The context is an object of type `CGContext` (also known as `CGContextRef`) that contains the drawing parameters and all the information required to draw the graphics. Although Quartz 2D offers several functions to create and manage contexts, they are usually not necessary. `UIView` objects create their context before calling the `drawRect()` method, so all we have to do to start drawing is to get a reference to the current context. The UIKit framework offers the following function for this purpose.

`UIGraphicsGetCurrentContext()`—This function gets a reference to the current graphic context.

Paths

The graphics in Quartz 2D are defined as paths. We have to create a path first and then send it to the context to be drawn. The following functions are provided to indicate the beginning and the ending of a path.

CGContextBeginPath(context)—This function indicates the beginning of a path in the context referenced by the **context** attribute.

CGContextClosePath(context)—This function closes the current path in the context referenced by the **context** attribute. If the path is not a closed path, the function adds a line between the pen's current position and the initial position to close the path.

The path is created as a combination of straight lines and curves. The strokes move from one point to another in the view's coordinates, like following the movement of a pen. Quartz 2D offers a set of functions to define the position of the pen and generate the path; the following are the most frequently used.

CGContextMoveToPoint(context, x, y)—This function moves the pen to the coordinates indicated by the **x** and **y** attributes.

CGContextAddLineToPoint(context, x, y)—This function adds a straight line to the path that goes from the pen's current position to the coordinates indicated by the **x** and **y** attributes.

CGContextAddLines(context, points, number)—This function adds multiple lines to the path. The **points** attribute is an array of **CGPoint** values with the coordinates for the lines (the lines are added in sequence), and the **number** attribute is the total amount of points in the array.

CGContextAddArc(context, x, y, radius, startAngle, endAngle, clockwise)—This function adds an arc to the path. The **x** and **y** attributes are the coordinates of the center of the circle formed by the arc, the **radius** attribute is the length of the circle's radius, the **startAngle** attribute is the angle in which the arc starts (expressed in radians), the **endAngle** attribute is the angle in which the arc ends (expressed in radians), and the **clockwise** attribute specifies the orientation in which the arc is calculated (1 for clockwise and 0 for counterclockwise).

CGContextAddArcToPoint(context, x1, y1, x2, y2, radius)—This function adds an arc to the path using tangent points. The **x1** and **y1** attributes define the

coordinates of the end of the first tangent line, the **x2** and **y2** attributes define the coordinates of the end of the second tangent line, and the **radius** attribute is the length of the circle's radius.

CGContextAddRect(context, rectangle)—This function adds a rectangle to the path. The **rectangle** attribute is a **CGRect** value with the coordinates and size of the rectangle.

CGContextAddEllipseInRect(context, rectangle)—This function adds an ellipse to the path. The **rectangle** attribute is a **CGRect** value that determines the area occupied by the ellipse. If the rectangle is a square, the ellipse will be a circle.

CGContextAddQuadCurveToPoint(context, cpx, cpy, x, y)—This function adds a quadratic Bezier curve to the path with only one control point. The **cpx** and **cpy** attributes define the coordinates of the control point of the curve, and the **x** and **y** coordinates define the coordinates of the ending point.

CGContextAddCurveToPoint(context, cp1x, cp1y, cp2x, cp2y, x, y)—This function adds a cubic Bezier curve to the path with two control points. The **cp1x** and **cp1y** attributes define the coordinates of the first control point of the curve, the **cp2x** and **cp2y** attributes define the coordinates of the second control point of the curve, and the **x** and **y** coordinates define the coordinates of the ending point.

A path defines the line to trace but it doesn't draw anything. To draw the path in the context, and therefore show it in the view, Quartz 2D offers two convenient functions.

CGContextStrokePath(context)—This function draws a line that follows the current path.

CGContextFillPath(context)—This function paints the area within the current path.

To create a path and show it on the screen, we have to provide the functions with a reference to the context in which we want the path to be drawn and declare them in order, following the line of the pen. The following example gets a reference to the current context and creates a path with the shape of a triangle.

Listing 24-2: Drawing a triangle

```
import UIKit
```

```
class DrawingView: UIView {  
    override func drawRect(rect: CGRect) {  
        let context = UIGraphicsGetCurrentContext()
```

```

CGContextBeginPath(context)
CGContextMoveToPoint(context, 10, 10)
CGContextAddLineToPoint(context, 100, 10)
CGContextAddLineToPoint(context, 10, 100)
CGContextClosePath(context)
CGContextStrokePath(context)
}
}

```

By default, the pen's initial position is the coordinate 0, 0 (top-left corner of the view). To set the initial position for our graphic, we call the `CGContextMoveToPoint()` function. In Listing 24-2 we indicate the beginning of a new path with the `CGContextBeginPath()` function and then move the pen to the coordinates 10, 10 before adding the first line. The lines or curves added to the path are generated from the current position of the pen to the coordinates indicated by the function (every line or curve starts from the point the previous one ended). For instance, the next function in our example creates a line from the initial point 10, 10 to the point 100, 10. The next line starts at 100, 10 and ends at 10, 100. Figure 24-2 shows the path created by this example.

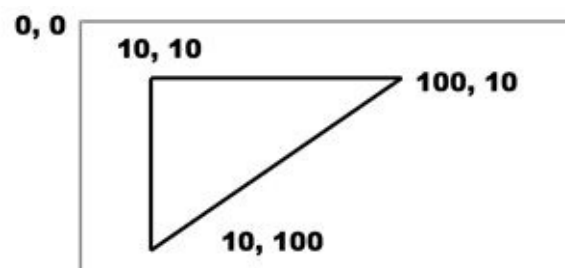


Figure 24-2: Representation of a path

Notice that in our example we only created two lines. The line that goes from the point 10, 100 to the point 10, 10 is created by the `CGContextClosePath()` function to close the path. If we want to create an open path, we can ignore this function.

Do It Yourself: Update the `DrawingView` class with the code of Listing 24-2 and run the application. You should see a shape similar to Figure 24-2 on the screen. Use the same project to test the following examples.

The code of Listing 24-2 uses the `CGContextStrokePath()` function to draw the path in the context and generate an empty figure, but we can call the `CGContextFillPath()` function instead if we want the path to be solid. When we use this function, we don't need to close the path.

Listing 24-3: Painting a path

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        CGContextBeginPath(context)
        CGContextMoveToPoint(context, 10, 10)
        CGContextAddLineToPoint(context, 100, 10)
        CGContextAddLineToPoint(context, 10, 100)
        CGContextFillPath(context)
    }
}
```

Combining different functions, we can create complex paths. The following path creates a graphic combining two lines and an arc.

Listing 24-4: Complex paths

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        CGContextBeginPath(context)
        CGContextMoveToPoint(context, 10, 10)
        CGContextAddLineToPoint(context, 100, 10)

        let startAngle = CGFloat(270 * M_PI / 180)
        let endAngle = CGFloat(90 * M_PI / 180)
        CGContextAddArc(context, 100, 30, 20, startAngle, endAngle, 0)
        CGContextAddLineToPoint(context, 10, 50)

        CGContextStrokePath(context)
    }
}
```

The angle for arcs and ellipses has to be expressed in radians. Radian is the standard unit to measure angles, but we can easily convert degrees into radians with the formula **degrees * M_PI / 180**, where **degrees** is the value of the angle in degrees and **M_PI** is a constant that returns the value of PI (see Chapter 4). Because arcs are calculated based on the

coordinates of the center of the circle and its radius, we have to consider these two values to connect the arc with the previous stroke. If the starting coordinates of the arc don't coincide with the current position of the pen, a line is created between these two points to connect the path. Figure 24-3 illustrates the path we get with the example of Listing 24-4 (left) and what we will see if we move the center of the circle 10 points to the top (right).

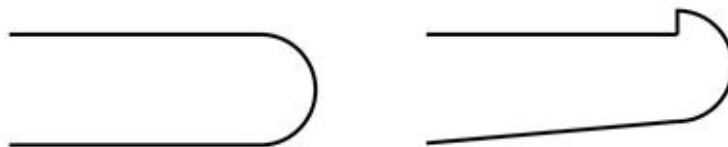


Figure 24-3: Path combining lines and arcs

The `CGContextAddRect()` and `CGContextAddEllipseInRect()` functions allow us to add rectangles and circles to the path. The functions add the figures to the current path, but they move the pen to the position indicated by the rectangle, so it can be considered as an independent shape.

Listing 24-5: Adding circles to the path

import UIKit

```
class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        CGContextBeginPath(context)
        CGContextMoveToPoint(context, 10, 10)
        CGContextAddLineToPoint(context, 100, 10)
        let frame = CGRect(x: 100, y: 0, width: 20, height: 20)
        CGContextAddEllipseInRect(context, frame)
        CGContextStrokePath(context)
    }
}
```

In this case no line is generated between the current position of the pen and the circle if they are not connected. Figure 24-4 illustrates the path we get with the example of Listing 24-5 (left) and what we will see if we move the area of the circle 10 points to the right (right).



Figure 24-4: Path with straight lines and circles

Every time we need to create a shape we have to initialize a path, add all the strokes necessary for the shape, and draw the path in the context. Because simple geometric shapes like rectangles and circles are very common, Quartz 2D offers a group of functions that draw directly into the context, significantly reducing the amount of code we have to write.

CGContextStrokeRect(context, rectangle)—This function draws an empty rectangle on the context. The **rectangle** attribute is a **CGRect** value with the coordinates and size of the rectangle.

CGContextFillRect(context, rectangle)—This function draws a solid rectangle in the context. The **rectangle** attribute is a **CGRect** value with the coordinates and size of the rectangle.

CGContextClearRect(context, rectangle)—This function clears a rectangular area in the context. The **rectangle** attribute is a **CGRect** value with the coordinates and size of the area.

CGContextStrokeEllipseInRect(context, rectangle)—This function draws an empty ellipse in the context. The **rectangle** attribute is a **CGRect** value with the coordinates and size of the area.

CGContextFillEllipseInRect(context, rectangle)—This function draws a solid ellipse in the context. The **rectangle** attribute is a **CGRect** value with the coordinates and size of the area.

These figures create their own path and draw it into the context all at once. The following example draws a few circles inside a rectangle to simulate a face. Notice that we don't have to open and close the path anymore.

Listing 24-6: Drawing basic shapes

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()

        var frame = CGRect(x: 10, y: 10, width: 100, height: 100)
        CGContextStrokeRect(context, frame)
        frame = CGRect(x: 25, y: 30, width: 20, height: 20)
```

```

CGContextFillEllipseInRect(context, frame)
frame = CGRect(x: 75, y: 30, width: 20, height: 20)
CGContextFillEllipseInRect(context, frame)
frame = CGRect(x: 40, y: 60, width: 40, height: 40)
CGContextStrokeEllipseInRect(context, frame)
}
}

```



Figure 24-5: Shapes drawn directly into the context

Colors

Core Graphics is programmed in the C Language and therefore it doesn't work with high-level classes like `UIColor`. The framework defines its own data type to generate colors called `CGColor` (also known as `CGColorRef`). Along with `CGColor`, Core Graphics defines another data type called `CGColorSpace` (also known as `CGColorSpaceRef`) to specify the type of colors we are working with. The framework includes the following functions to create a color space and a color.

CGColorSpaceCreateDeviceCMYK()—This function creates a color space based on the CMYK color model. The model uses the basic colors cyan, magenta, yellow, and black to create all the colors in the spectrum.

CGColorSpaceCreateDeviceGray()—This function creates a color space composed by a gray scale.

CGColorSpaceCreateDeviceRGB()—This function creates a color space based on the RGB model. The model uses the basic colors red, green, and blue to create all the colors in the spectrum.

CGColorCreate(space, components)—This function creates a color using the color space specified by the **space** attribute and the values specified by the **components** attribute. The **components** attribute is an array with the values of the colors. The types of values depend on the color space selected. For RGB colors, for example, the array requires four `CGFloat` values between 0.0 and 1.0 to define the level of red, green, blue,

and transparency the color will have.

By default, the color assigned to the path is black, but we can assign a different color before the path is drawn in the context using the following functions.

CGContextSetStrokeColorWithColor(context, color)—This function defines the color for the stroke of the path.

CGContextSetFillColorWithColor(context, color)—This function defines the color for the interior of the path.

Only one color per path can be defined (one for the stroke and another for the filling). If we want our graphic to have more than one color we have to create different paths and change the color before the path is drawn in the context. The following example shows how to define a color for a single path.

Listing 24-7: Changing the color of the path

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        let space = CGColorSpaceCreateDeviceRGB()
        let components: [CGFloat] = [0.8, 0.8, 0.8, 1]
        let color = CGColorCreate(space, components)

        CGContextSetStrokeColorWithColor(context, color)
        CGContextBeginPath(context)
        CGContextMoveToPoint(context, 10, 10)
        CGContextAddLineToPoint(context, 100, 100)
        CGContextStrokePath(context)
    }
}
```

Although Core Graphics can't work with `UIColor` values, the class includes a property called `CGColor` that returns a `CGColor` value with the color of the object. This simplifies the creation of the colors for our paths. The following example creates two boxes painted with two different colors generated from the type methods provided by the `UIColor` class.

Listing 24-8: Creating colors from UIColor

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        let color1 = UIColor.grayColor()
        CGContextSetFillColorWithColor(context, color1.CGColor)
        CGContextFillRect(context, CGRect(x: 10, y: 10, width: 100, height: 100))
        let color2 = UIColor.lightGrayColor()
        CGContextSetFillColorWithColor(context, color2.CGColor)
        CGContextFillRect(context, CGRect(x: 120, y: 10, width: 100, height: 100))
    }
}
```

Configuration

Quartz 2D also includes functions to configure the lines before drawing the path in the context. The following are the most frequently used.

CGContextSetLineWidth(context, width)—This function defines the width for the lines. The **width** attribute is a **CGFloat** value with the new width.

CGContextSetLineDash(context, phase, lengths, number)—This function defines the lines as dash lines. The **phase** attribute is a **CGFloat** value that determines where the line begins, the **lengths** attribute is an array of **CGFloat** values that specify the lengths of the segments, and the **number** attribute is an integer with the total amount of values defined in the **lengths** array.

CGContextSetLineCap(context, cap)—This function sets the style for the ending of the lines. The **cap** attribute is an enumeration of type **CGLineCap** with the values **Butt** (squared end), **Round** (round end), and **Square** (squared end extended half the width of the line).

CGContextSetLineJoin(context, joint)—This function sets the style of the joints of two connected lines. The **joint** attribute is an enumeration of type **CGLineJoin** with the values **Miter** (joins the lines with a sharp end), **Round** (joins the lines with a rounded end), and **Bevel** (joins the lines with a square end).

CGContextSetMiterLimit(context, limit)—This function sets the limit of the lines when the joint is set as **Miter**. The **limit** attribute is a **CGFloat** value that determines how long the joint will extend.

CGContextSetBlendMode(context, mode)—This function sets the blend mode used when paths overlapped. The **mode** attribute is an enumeration of type **CGBlendMode** with the values **Normal**, **Multiply**, **Screen**, **Overlay**, **Darken**, **Lighten**, **ColorDodge**, **ColorBurn**, **SoftLight**, **HardLight**, **Difference**, **Exclusion**, **Hue**, **Saturation**, **Color**, **Luminosity**, **Clear**, **Copy**, **SourceIn**, **SourceOut**, **SourceAtop**, **DestinationOver**, **DestinationIn**, **DestinationOut**, **DestinationAtop**, **XOR**, **PlusDarker**, and **PlusLighter**.

The following example creates a path with two lines configured as dashed lines with a width of 10 points and round endings.

Listing 24-9: Configuring the path’s lines

```
import UIKit
```

```
class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        CGContextSetLineWidth(context, 10)
        CGContextSetLineCap(context, .Round)
        CGContextSetLineJoin(context, .Round)
        CGContextSetLineDash(context, 0, [15], 1)
        CGContextBeginPath(context)
        CGContextMoveToPoint(context, 20, 80)
        CGContextAddLineToPoint(context, 70, 10)
        CGContextAddLineToPoint(context, 120, 80)
        CGContextStrokePath(context)
    }
}
```



Figure 24-6: Dashed lines with round endings

Saving the State

Changes in the configuration are performed in the context and affect all the paths drawn after the changes were made. Sometimes we need to configure the context only for one

path but we don't want the rest of the paths to be affected. If the change we have performed is simple, like when we modify the width of the line, we can call the same function again to restore the previous value, but when going back to a previous state is not that simple, we have to save and restore the context's state using the functions provided by Quartz 2D.

CGContextSaveGState(context)—This function saves the current state of the context.

CGContextRestoreGState(context)—This function restores the previous state of the context.

The state must be saved before any change is introduced and restored after the path is drawn in the context. For example, we can change the width of the lines to 10 points for the first path and then restore the state to use the value by default for the remaining paths in the graphic.

Listing 24-10: Saving and restoring the state of the context

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()

        CGContextSaveGState(context)
        CGContextSetLineWidth(context, 10)
        CGContextBeginPath(context)
        CGContextMoveToPoint(context, 20, 20)
        CGContextAddLineToPoint(context, 20, 100)
        CGContextStrokePath(context)

        CGContextRestoreGState(context)
        CGContextStrokeEllipseInRect(context, CGRect(x: 40, y: 60, width: 40, height: 40))
    }
}
```



Figure 24-7: Paths drawn in different states

Clipping

The context creates a mask that delimits the space available for the graphics to the boundaries of the drawing surface (the view in our example). We can create our own mask to determine which parts of the area can be drawn and which not. Quartz 2D offers the following function to set the mask.

CGContextClip(context)—This function creates a mask from the current path.

The mask is created from a path, so we have to define the path first and then call the **CGContextClip()** function. The following example creates a circular mask and then draws a path with multiple straight lines. Only the part of the lines that are inside the mask are drawn.

Listing 24-11: Delimiting the drawing area

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        CGContextAddEllipseInRect(context, CGRect(x: 50, y: 50, width: 100, height: 100))
        CGContextClip(context)
        CGContextBeginPath (context)
        for var pos: CGFloat = 10; pos <= 150; pos += 10 {
            CGContextMoveToPoint(context, 50, pos)
            CGContextAddLineToPoint(context, 150, pos)
        }
        CGContextStrokePath(context)
    }
}
```



Figure 24-8: Mask

Gradients

Quartz 2D also allows us to paint the paths with gradients. There are two types of gradients: linear and radial. The linear gradient is composed by perpendicular lines of color distributed along an axis, while a radial gradient is composed by circles of different color that expand along an axis. Core Graphics defines a data type called **CGGradient** (also known as **CGGradientRef**) to store the basic information for a gradient and the following functions to create and draw a gradient in the context.

CGGradientCreateWithColors(space, colors, locations)—This function returns a **CGGradient** value with a gradient configured according to the values specified by the attributes. The **space** attribute is a **CGColorSpace** value that represents the color space we want to use to create the colors, the **colors** attribute is an array of **CGColor** values that represent the colors we want to use to create the gradient, and the **locations** attribute is an array with values between 0.0 and 1.0 that represent the locations along the axis in which the gradient turns from one color to another.

CGContextDrawLinearGradient(context, gradient, startPoint, endPoint, options)—This function adds a linear gradient to the context. The **gradient** attribute is a reference to the **CGGradient** value we want to use, the **startPoint** and **endPoint** attributes indicate the starting and ending point of the axis over which the gradient will be created, and the **options** attribute is one of two constants, **DrawsBeforeStartLocation** and **DrawsAfterEndLocation**, that indicate whether the gradient will extend beyond the starting location or after the ending location.

CGContextDrawRadialGradient(context, gradient, startCenter, startRadius, endCenter, endRadius, options)—This function adds a radial gradient to the context. Radial gradients are created from one circle to another. The **gradient** attribute is a reference to the **CGGradient** value we want to use, the **startCenter** and **endCenter** attributes are **CGPoint** values that determine the positions of the center of the circles, the **startRadius** and **endRadius** attributes are **CGFloat** values that determine the radius of the circles, and the **options** attribute is one of two constants, **DrawsBeforeStartLocation** and **DrawsAfterEndLocation**, that indicate whether the gradient will extend beyond the starting location or after the ending location.

Gradients are applied to the context. If we want to paint a path with a gradient, we have to create a mask with that path to delimit the area of the gradient. The following example creates a rectangular path to use as the mask and then adds a linear gradient to the context that goes from the color white to black.

Listing 24-12: Creating a linear gradient

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        CGContextBeginPath(context)
        CGContextAddRect(context, CGRect(x: 10, y: 10, width: 100, height: 100))
        CGContextClip(context)

        let space = CGColorSpaceCreateDeviceRGB()
        let locations: [CGFloat] = [0.0, 0.9]
        let color1 = UIColor.whiteColor()
        let color2 = UIColor.blackColor()
        let colors = [color1.CGColor, color2.CGColor]

        let gradient = CGGradientCreateWithColors(space, colors, locations)
        let startPoint = CGPoint(x: 10, y: 10)
        let endPoint = CGPoint(x: 110, y: 110)
        CGContextDrawLinearGradient(context, gradient, startPoint, endPoint, .DrawsBeforeStartLocation)
    }
}
```

There are two steps involved in the creation of a gradient. We have to define the properties of the gradient first with the `CGGradientCreateWithColors()` function and then add the gradient to the context with the function corresponding to the gradient we want to generate (linear or radial). The `CGGradientCreateWithColors()` function requires a few values to configure the gradient. The first value is the color space we want to use to define the colors of the gradient. In Listing 24-12 we used the most common model called RGB. The second value is an array with the list of colors that are going to participate in the gradient. The colors are added to the gradient in the order they are declared in this array. The last value is another array containing values between 0.0 and 1.0 indicating the position in which every color will start. In our example we decided to create a gradient from white to black, starting with white (0.0) and incorporating black close to the end (0.9). Once the gradient is defined, we add it to the context as a linear gradient with the `CGContextDrawLinearGradient()` function. Because linear gradients are conformed by straight lines perpendicular to an axis, we have to provide the points in which the axis starts and ends. In this example we define those points to be the corners of the rectangular area we want to paint, so the gradient is distributed homogeneously through that area. The result is shown in Figure 24-9.

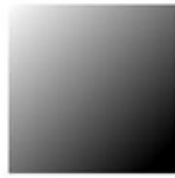


Figure 24-9: *Linear gradient*

Radial gradients are similar, but they are created with circular lines extended from one circle to another. The following example uses a small circle inside a bigger one to create a tridimensional effect. Notice that in this example we don't have to define a mask to delimit the area we want to paint because the gradient is drawn in between the boundaries determined by the circles.

Listing 24-13: *Creating a radial gradient*

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        let space = CGColorSpaceCreateDeviceRGB()
        let locations: [CGFloat] = [0.0, 0.9]
        let color1 = UIColor.lightGrayColor()
        let color2 = UIColor.blackColor()
        let colors = [color1.CGColor, color2.CGColor]

        let gradient = CGGradientCreateWithColors(space, colors, locations)

        CGContextDrawRadialGradient(context, gradient, CGPoint(x: 30, y: 30), 0, CGPoint(x: 50, y: 50), 40,
        .DrawsBeforeStartLocation)
    }
}
```



Figure 24-10: *Radial gradient*

Shadows

Another good effect we can create with Quartz 2D is shadows. The shadow is declared as part of the configuration for the context. Once the shadow is set, every path drawn afterwards will include a shadow. The framework provides the following functions to set a shadow.

CGContextSetShadow(context, offset, blur)—This function configures the shadow for the context with the color by default. The **offset** attribute is a **CGSize** value that determines the horizontal and vertical offset of the shadow in relation to the path, and the **blur** attribute is a **CGFloat** value that specifies the amount of blur.

CGContextSetShadowWithColor(context, offset, blur, color)—This function configures the shadow for the context with a specific color. The **offset** attribute is a **CGSize** value that determines the horizontal and vertical offset of the shadow in relation to the path, the **blur** attribute is a **CGFloat** value that specifies the amount of blur, and the **color** attribute is a **CGColor** value.

Listing 24-14: Generating shadows

```
import UIKit
```

```
class DrawingView: UIView {  
    override func drawRect(rect: CGRect) {  
        let context = UIGraphicsGetCurrentContext()  
        CGContextSetLineWidth(context, 10)  
        CGContextSetShadow(context, CGSize(width: 5, height: 5), 5)  
        CGContextStrokeEllipseInRect(context, CGRect(x: 10, y: 10, width: 100, height: 100))  
    }  
}
```



Figure 24-11: Shadows

Transformations

As we mentioned in Chapter 4, Apple devices work with two coordinate systems, one expressed in pixels that represent the native resolution of the device and another expressed in points that provide a standard resolution that works the same way on any device. Quartz

2D is responsible for transforming the graphics from one coordinate system to another. The process is performed using a transformation matrix that calculates the positions of every pixel. The current transformation matrix (CTM) may be modified to change the aspect of the graphics.

Modifying the CTM we can introduce multiple changes to the graphics, but the most common operations are translation, scaling, and rotation. Quartz 2D offers the following functions to achieve these effects.

CGContextTranslateCTM(context, x, y)—This function changes the origin of the coordinates system. The **x** and **y** attributes are **CGFloat** values that determine the new positions of the origin (0, 0).

CGContextRotateCTM(context, angle)—This function rotates the coordinate system. The **angle** attribute is a **CGFloat** value with the rotation angle in radians.

CGContextScaleCTM(context, x, y)—This function changes the scale of the coordinate system. The **x** and **y** attributes are **CGFloat** values that determine the scaling factor (1.0 keeps the current scale and negative values invert the coordinate system).

The transformations are performed in the context, affecting all the paths drawn afterwards. For example, if we translate the origin of the coordinate system to a new position, from that moment on the position of the paths in the context will be determined according to the new origin. The following example creates a rectangle and an ellipse, both with the same size and positioned at the coordinates 5, 5, but because the origin is changed to the point 100, 0, they are drawn closer to the center of the view instead of the edge.

***Listing 24-15:** Translating the coordinate system's origin*

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        CGContextTranslateCTM(context, 100, 0)
        let frame = CGRect(x: 5, y: 5, width: 50, height: 20)
        CGContextStrokeRect(context, frame)
        CGContextStrokeEllipseInRect(context, frame)
    }
}
```

Figure 24-12 shows how translation affects the paths. The picture on the left shows the

paths drawn without translation, and the picture on the right shows the paths drawn after the origin is displaced 100 points to the right.



Figure 24-12: Translation

Establishing the origin of the coordinate system is useful in many circumstances but especially when we need to rotate the graphics. The rotation is performed around the origin, so if we want to rotate a path around its center we have to move the origin to the center of the path before performing the rotation.

Listing 24-16: Applying two transformations to rotate a shape

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        let degrees = CGFloat(90 * M_PI / 180)

        CGContextTranslateCTM(context, 100, 100)
        CGContextRotateCTM(context, degrees)
        let frame = CGRect(x: -25, y: -10, width: 50, height: 20)
        CGContextStrokeRect(context, frame)
    }
}
```



Figure 24-13: Rotation

The transformations are accumulative. If we rotate before translating, the translation is performed over a context that was already rotated. If we rotate the context 45 degrees, and then apply another rotation of 45 degrees, the final rotation of the context will be 90 degrees. The following example takes advantage of this to draw multiple rectangles in the

context, each with a different rotation.

Listing 24-17: Applying successive transformations

```
import UIKit
```

```
class DrawingView: UIView {  
    override func drawRect(rect: CGRect) {  
        let context = UIGraphicsGetCurrentContext()  
        CGContextTranslateCTM(context, 100, 100)  
        for var f: Double = 0; f < 360; f = f + 45 {  
            let degrees = CGFloat(f * M_PI / 180)  
            CGContextRotateCTM(context, degrees)  
            let frame = CGRect(x: 0, y: 0, width: 50, height: 20)  
            CGContextStrokeRect(context, frame)  
        }  
    }  
}
```

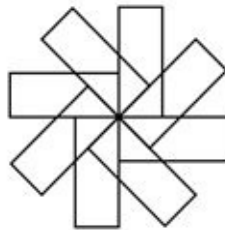


Figure 24-14: Multiple rotations

We can also change the scale of the context with the `CGContextScaleCTM()` function. For example, if we want to duplicate the size of the paths drawn in the context we can call this function with the values 2.0 for the horizontal scale and 2.0 for the vertical scale. But modifying the scale is not only about resizing the graphics; we can also achieve cool effects. For example, we can shrink or stretch paths by declaring different values for the horizontal and vertical scales, or we can create a mirror image by declaring a negative scale value. The following example uses this trick to invert the coordinate system and draw an inverted triangle on the other side of the screen.

Listing 24-18: Inverting the context

```
import UIKit
```

```
class DrawingView: UIView {  
    override func drawRect(rect: CGRect) {  
        let context = UIGraphicsGetCurrentContext()
```

```

drawTriangle(context!)
CGContextTranslateCTM(context, self.bounds.size.width, 0)
CGContextScaleCTM(context, -1.0, 1.0)
drawTriangle(context!)
}
func drawTriangle(context: CGContext) {
    CGContextBeginPath(context)
    CGContextMoveToPoint(context, 10, 10)
    CGContextAddLineToPoint(context, 60, 60)
    CGContextAddLineToPoint(context, 10, 110)
    CGContextAddLineToPoint(context, 10, 10)
    CGContextStrokePath(context)
}
}

```

The axis used to create the mirror is determined by the origin. If we create a horizontal mirror, the negative values are now on the right side of the horizontal origin and the positive values are on the left. If we want the paths with positive coordinates to appear on the screen, we have to translate the context, as we do in Listing 24-18. The code creates a method called `drawTriangle()` to draw a triangle on the left side of the context. The first time the method is called, the context is in its state by default, but the second time the method is called after the context was transformed and therefore the new triangle is inverted and drawn on the right side of the context.



Figure 24-15: Mirror context

Besides modifying the current transformation matrix with these functions we can also create a transformation matrix. Core Graphics defines the `CGAffineTransform` data type to store a transformation matrix and the following functions to create one.

CGAffineTransformMakeTranslation(x, y)—This function returns a matrix that translates a coordinate system. The **x** and **y** attributes are `CGFloat` values that determine the new positions of the origin (0, 0).

CGAffineTransformMakeRotation(angle)—This function returns a matrix that rotates a coordinate system. The **angle** attribute is a `CGFloat` value with the rotation angle

in radians.

CGAffineTransformMakeScale(x, y)—This function returns a matrix that scales a coordinate system. The **x** and **y** attributes are **CGFloat** values that determine the scaling factor (1.0 keeps the current scale and negative values invert the coordinate system).

A **CGAffineTransform** transformation matrix is usually applied by functions that require their own matrix, as we will see later, but we can also apply it directly to the context using a function provided by Core Graphics called **CGContextConcatCTM()**. The function takes two parameters, a reference to the context and a **CGAffineTransform** value, and concatenates this matrix with the CTM.

Listing 24-19: Concatenating matrices

import UIKit

```
class DrawingView: UIView {  
    override func drawRect(rect: CGRect) {  
        let context = UIGraphicsGetCurrentContext()  
        CGContextTranslateCTM(context, 100, 100)  
        let degrees = CGFloat(45 * M_PI / 180)  
        let matrix = CGAffineTransformMakeRotation(degrees)  
        CGContextConcatCTM(context, matrix)  
        CGContextStrokeRect(context, CGRect(x: -25, y: -10, width: 50, height: 20))  
    }  
}
```

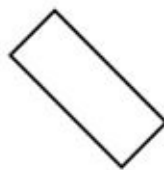


Figure 24-16: *Using an Affine Transformation matrix*

Text

There are different ways to draw text in the context, but Apple recommends using Core Text. Core Text is the low-level framework Apple systems use to draw text on the screen. Several high-level elements like **UILabel**, for example, use this framework to generate their content. Core text can style and configure lines and entire paragraphs of text, but it is usually implemented along with Core Graphics to draw labels in the context. The

framework defines a data type called `CTLine` to store a line of text and the following functions to create the line and draw it in the context.

CTLineDraw(line, context)—This function draws a line in the context. The **line** attribute is a `CTLine` value with the line of text we want to draw.

CTLineCreateWithAttributedString(string)—This function returns a `CTLine` value that represents a line of text. The **string** attribute is the attributed string we want to assign to the line.

CTLineCreateTruncatedLine(line, width, type, ending)—This function returns a `CTLine` value with a truncated line created from an existing line of text. The **line** attribute is a `CTLine` value with the line of text we want to truncate, the **width** attribute is a `Double` value that determines where the line is truncated, the **type** attribute is an enumeration value of type `CTLineTruncationType` that indicates where the truncation should be performed (the possible values are `Start`, `End`, and `Middle`), and the **ending** attribute is an optional `CTLine` value that determines the characters added where the truncation took place (`nil` cuts the line off).

CTLineCreateJustifiedLine(line, factor, width)—This function returns a `CTLine` value with a justified line created from an existing line of text. The **line** attribute is a `CTLine` value with the line of text we want to justify, the **factor** attribute is a value around 1.0 that determines the degree of justification performed (values below 1.0 determine partial justification and values over 1.0 determine full justification), and the **width** attribute is a `Double` value that determines the width to which the line has to be justified.

Core Graphics and Core Text use a coordinate system that is a mirror of the coordinate system implemented by UIKit. When we work with Core Graphics and `UIView` objects, the view takes care of inverting the coordinate system to show the graphics in the right position on the screen, but it doesn't do that for text generated by Core Text. For this reason, before drawing any text in the context we have to apply the necessary transformations to invert and position the text. Core Graphics offers two practical functions to set the text's position and its own transformation matrix, so we don't have to deal with changing the transformation matrix of the entire context.

CGContextSetTextPosition(context, x, y)—This function sets the position of the text in the context. The vertical position is determined from the text baseline. The **x** and **y** attributes are `CGFloat` values that determine the horizontal and vertical position, respectively.

CGContextSetTextMatrix(context, matrix)—This function sets the transformation matrix for the text. The **matrix** attribute is a `CGAffineTransform` value.

The line of text is created from an attributed string, so the first thing we need to do is to create an `NSMutableAttributedString` object. With the attributed text ready, we can call the Core Text functions to draw it in the context.

Listing 24-20: Drawing text in the context

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        let font = UIFont(name: "BradleyHandITCTT-Bold", size: 30)
        let attributes = [NSFontAttributeName: font!]
        let string = NSMutableAttributedString(string: "Hello", attributes: attributes)

        let line = CTLineCreateWithAttributedString(string)
        CGContextSetTextMatrix(context, CGAffineTransformMakeScale(1.0, -1.0))
        CGContextSetTextPosition(context, 10.0, 30.0)
        CTLineDraw(line, context!)
    }
}
```

Images

Core Graphics defines its own data type to work with images called `CGImage` (also known as `CGImageRef`). The `CGImage` data type provides a low-level representation of images that is used by other high-level frameworks like UIKit to process images. We can work with `CGImage` values directly or through the `UIImage` class. This class offers the following property and initializer to get a `CGImage` value from a `UIImage` object and to create a `UIImage` object from a `CGImage` value.

CGImage—This property returns a `CGImage` value with the image in the `UIImage` object.

UIImage(image, scale, orientation)—This initializer creates a `UIImage` object from the image provided by the **image** attribute. The **image** attribute is a `CGImage` value with the image we want to transform, the **scale** attribute is a `CGFloat` value with the scale of the image, and the **orientation** attribute is an enumeration of type `UIImageOrientation`

that indicates the orientation we want to associate to the image. The possible values are **Up**, **Down**, **Left**, **Right**, **UpMirrored**, **DownMirrored**, **LeftMirrored**, and **RightMirrored**.

The **UIImage** class also offers two methods to draw an image onto the context without having to turn it first into a **CGImage** value.

drawAtPoint(point)—This method draws the image at the location specified by the **point** attribute. The attribute is a **CGPoint** value.

drawInRect(rectangle)—This function draws the image inside the rectangle determined by the **rectangle** attribute. The attribute is a **CGRect** value.

The difference between **CGImage** values and **UIImage** objects is that **UIImage** was designed to work along with other UIKit elements and therefore it considers aspects of the image like the scale and orientation, while **CGImage** can just store a rectangular array of pixels. Therefore, depending on what we need, we may work with one or another. Core Graphics offers a set of functions to process **CGImage** values when we have to. The following are the most frequently used.

CGImageGetWidth(image)—This function returns an integer with the width of the image in pixels.

CGImageGetHeight(image)—This function returns an integer with the height of the image in pixels.

CGContextDrawImage(context, rectangle, image)—This function draws an image onto the context. The **rectangle** attribute is a **CGRect** value with the location and size of the area in which the image will be drawn (the image is scaled to fit this area), and the **image** attribute is a **CGImage** value with the image we want to draw.

CGContextDrawTiledImage(context, rectangle, image)—This function draws an image as many times as necessary to cover the context's visible area. The **rectangle** attribute is a **CGRect** value with the location and size of the area in which each image will be drawn (the image is scaled to fit this area), and the **image** attribute is a **CGImage** value with the image to draw.

CGImageCreateWithImageInRect(image, rectangle)—This function returns a **CGImage** value with an image extracted from another image. The **image** attribute is a **CGImage** value with the image we want to process and the **rectangle** attribute is a **CGRect** value that indicates the rectangle of the original image we want to use to create the new one.

The following example creates an `UIImage` object as we always do, read its `CGImage` property, and use the value returned to draw the image in the context.

Listing 24-21: Drawing an image in the context

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        if let image = UIImage(named: "husky") {
            let imageData = image.CGImage
            let width = image.size.width
            let height = image.size.height

            CGContextDrawImage(context, CGRect(x: 0, y: 50, width: width, height: height), imageData)
        }
    }
}
```

Drawing `CGImage` values into the context, as we did in Listing 24-21, presents the same problem we had when working with Core Text. The coordinate system used by `CGImage` values is not the same as the one used by UIKit, and therefore the image is inverted. Unlike Core Text, the `CGImage` data type doesn't provide its own matrix function so we have to transform the image ourselves. One solution is to translate and scale the entire context. For this purpose we have to position the origin below the image and then mirror the context, as in the following example.

Listing 24-22: Inverting the image

```
import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        if let image = UIImage(named: "husky") {
            let imageData = image.CGImage
            let width = image.size.width
            let height = image.size.height
            let posX: CGFloat = 0
            let posY: CGFloat = 50

            CGContextTranslateCTM(context, 0, height + (posY * 2))
        }
    }
}
```

```

    CGContextScaleCTM(context, 1.0, -1.0)
    CGContextDrawImage(context, CGRect(x: posX, y: posY, width: width, height: height), imageData)
}
}
}

```

The coordinates of UIKit establish the origin at the top-left corner of the context, however the origin in Core Graphics is at the bottom-left corner. The positive coordinates are also inverted. In UIKit they are incremented from top to bottom and in Core Graphics they are incremented from bottom to top. This means that the image drawn by the `CGContextDrawImage()` function is completely inverted. To get it to the right position we not only have to mirror the context but also calculate new coordinates. The easiest way to do it is to move the origin to the point below the image that represents the origin in UIKit coordinates and only then mirror the context. For this we need to know the position of the image before it is drawn in the context and that is why in Listing 24-22 we use the `posX` and `posY` constants to store the values.

The example of Listing 24-22 accomplishes what we need, the image is drawn at the right position and with the right orientation, but we have to perform a transformation in the context that affects future drawings and the final code is too long to maintain. The `UIImage` class offers the `drawAtPoint()` and `drawInRect()` methods that highly simplify this task. The methods draw the image contained in the `UIImage` object in the current context considering its orientation and using the UIKit coordinate system, saving us all this work.

Listing 24-23: Drawing a UIImage object directly into the context

```

import UIKit

class DrawingView: UIView {
    override func drawRect(rect: CGRect) {
        if let image = UIImage(named: "husky") {
            let width = image.size.width
            let height = image.size.height
            image.drawInRect(CGRect(x: 0, y: 50, width: width, height: height))
        }
    }
}

```

The `drawInRect()` method used in Listing 24-23 draws the image directly into the current context, so unless we want to draw other graphics besides the image, a reference to the current context is not required.

Do It Yourself: Download the `husky.png` image from our website and add it to the Assets Catalog. Update the `DrawingView` class in your project with the codes of Listings 24-21, 24-22, and 24-23 to see the difference in every approach.

Drawing Cycle

Views are not drawn just once; the system actually calls the `drawRect()` method in each view every time it determines that the interface needs an update. We don't have control over this process but we can indicate to the system that the view's content has changed by calling a method provided by the `UIView` class called `setNeedsDisplay()`. There may be an infinite number of situations in which we need to update the view. For instance, our application may draw a clock and need to update the graphics every second, or the graphics may respond to the user and the view has to be updated every time an action is performed. The following example follows this last approach. We draw a circle in the context when the user taps on the view. For this we need to add a gesture recognizer to the view and create the method to respond to that gesture. The following are the modifications we have to make in the `ViewController` class of our application to prepare the view.

Listing 24-24: Responding to the tap gesture

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var surface: DrawingView!
    override func viewDidLoad() {
        super.viewDidLoad()

        let gesture = UITapGestureRecognizer(target: surface, action: "addCircle:")
        surface.addGestureRecognizer(gesture)
    }
}
```

In Listing 24-24, we have added an Outlet called `surface` to connect the view in the interface of Figure 24-1 to the `ViewController` class. When the main view is loaded, this reference is used to add a Tap Gesture to the view. The gesture recognizer defines the view as the target and the `addCircle()` method as the method that responds to the gesture. This method is called on the view, so it has to be defined in the `DrawingView` class, but the method can't perform any drawing. The only time when we have access to a drawing context inside the view is when the `drawRect()` method is executed. The context is created by the view before this method is called and it is destroyed when the execution of the

method is over. Therefore, the response to the user is indirect. When the `addCircle()` method is called in the view we have to calculate the position of the graphics and then ask the system to update the view with the `setNeedsDisplay()` method, but all the drawing must be performed inside the `drawRect()` method.

The following example draws a circle at the position the user touched the screen. The location is processed by the `addCircle()` method and stored in a property called `position`, so when the `drawRect()` method is executed, we can read the current value of `position` and draw the circle in the context.

Listing 24-25: Drawing circles at the location determined by the tap gesture

```
import UIKit

class DrawingView: UIView {
    var position: CGPoint!

    override func drawRect(rect: CGRect) {
        let context = UIGraphicsGetCurrentContext()
        if position != nil {
            let radius: CGFloat = 20
            let frame = CGRect(x: position.x - radius, y: position.y - radius, width: radius * 2, height: radius * 2)
            CGContextFillEllipseInRect(context, frame)
        }
    }

    func addCircle(sender: UITapGestureRecognizer) {
        let location = sender.locationInView(self)
        position = CGPoint(x: location.x, y: location.y)
        self.setNeedsDisplay()
    }
}
```

After establishing the position of the circle, the `addCircle()` method calls the `setNeedsDisplay()` method to tell the system that the view needs an update. The system then calls the `drawRect()` method and the circle is drawn on the screen.

Every time the `drawRect()` method is called, the view has to perform all the drawings again, not only those that are being added. The system clears the context and expects the view to provide all the graphics again. In our example, we only store the coordinates of the last touch, so only one circle is drawn at a time, but we could store more information in a data model to create more complex graphics.

Do It Yourself: Update the `ViewController` class of your project with the code of Listing 24-24. Update the `DrawingView` class with the code of Listing 24-25. Run the

application and tap on the screen. Every time you tap, the system should erase the previous circle and draw a new one where your finger is.

IMPORTANT: A view creates its own coordinate system, with the origin at its top-left corner (position 0, 0). When the view does not occupy the entire screen, the coordinates of the view and the screen don't match. For example, if the view is positioned at the screen's coordinates 20, 20, anything drawn at the top-left corner of the view will be 20 points from the side of the screen, but it will be positioned at the view's coordinates 0, 0 (not 20, 20). If we need to transform the coordinates from one view to another, we can use the methods `convertPoint(point, toView: view)` and `convertPoint(point, fromView: view)`. These methods are provided by the `UIView` class and can transform the coordinates of a view to the coordinates of another view. For more information, visit our website and follow the links for this chapter.

Image Context

The context used by a `UIView` object has the purpose to show the graphics inside the view as soon as they are created, but we can also draw the graphics into an image using an image context and then store the image, process it, or show it to the user if we need to. The following are the functions provided by `UIKit` to create and manage an image context.

`UIGraphicsBeginImageContextWithOptions(size, opaque, scale)`—This function creates a graphic context for an image. The **size** attribute is a `CGSize` value that determines the area occupied by the context, the **opaque** attribute is a Boolean value that determines if the image is going to be opaque or not, and the **scale** attribute is a `CGFloat` value that determines the scale of the image (a value of 0.0 assigns the scale of the device to the image).

`UIGraphicsGetImageFromCurrentImageContext()`—This function creates a `UIImage` object with the image in the current image context.

`UIGraphicsEndImageContext()`—This function removes the current image context. It is required to call this function after the context is no longer necessary as a way to free memory and clean the context stack for future use.

Because the purpose of an image context is to create an image, if we want to show the result on the screen, instead of a `UIView` object we have to use a `UIImageView` object. The following example assumes that we have added a `UIImageView` to the initial view and connected it to an `Outlet` in the `ViewController` class called `imageDraw`. The code creates an image context of the size of the Image View and then draws a circle on it. After the

graphics are ready, we get the image from the context using the `UIGraphicsGetImageFromCurrentImageContext()` function and assign it to the Image View to show it on the screen.

Listing 24-26: Creating a new image

import UIKit

```
class ViewController: UIViewController {
    @IBOutlet weak var imageDraw: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let width = imageDraw.bounds.size.width
        let height = imageDraw.bounds.size.height
        let imageSize = CGSize(width: width, height: height)
        UIGraphicsBeginImageContextWithOptions(imageSize, false, 0.0)
        let context = UIGraphicsGetCurrentContext()
        let frame = CGRect(x: 10, y: 10, width: width - 20, height: height - 20)
        CGContextStrokeEllipseInRect(context, frame)
        let newImage = UIGraphicsGetImageFromCurrentImageContext()
        imageDraw.image = newImage
        UIGraphicsEndImageContext()
    }
}
```

The `UIGraphicsBeginImageContextWithOptions()` function creates an image context but it doesn't return a reference to it. To get the reference and be able to draw in the context we have to call the `UIGraphicsGetCurrentContext()` function. Because the context can't be stored or preserved in any form, we have to turn its content into an image with the `UIGraphicsGetImageFromCurrentImageContext()` function and then close it with the `UIGraphicsEndImageContext()`.

Do It Yourself: Create a new Single View Application project. Add an Image View to the initial view and pin it to the margins of the main view, as we did for the view in the interface of Figure 24-1. Connect the Image View to the `ViewController` class with an Outlet called `imageDraw`. Complete the class with the code of Listing 24-26 and run the application. You should see an ellipse covering the entire area occupied by the Image View.

Of course, an image context behaves exactly the same way as a view context so we can draw anything we like in it, including images. The following example creates a

composition with two images and a rectangle.

Listing 24-27: Composing an image

import UIKit

```
class ViewController: UIViewController {
    @IBOutlet weak var imageDraw: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()

        view.setNeedsLayout()
        view.layoutIfNeeded()
        let width = imageDraw.bounds.size.width
        let height = imageDraw.bounds.size.height
        let imageSize = CGSize(width: width, height: height)

        UIGraphicsBeginImageContextWithOptions(imageSize, false, 0.0)
        let context = UIGraphicsGetCurrentContext()
        if let image1 = UIImage(named: "clouds"), let image2 = UIImage(named: "husky") {
            image1.drawInRect(CGRect(x: 0, y: 0, width: width, height: height))
            let image2Width: CGFloat = 200
            let image2Height = image2Width * image2.size.height / image2.size.width
            let posX: CGFloat = (width - image2Width) / 2
            let posY: CGFloat = (height - image2Height) / 2
            let frame = CGRect(x: posX, y: posY, width: image2Width, height: image2Height)
            image2.drawInRect(frame)
            CGContextSetLineWidth(context, 10)
            CGContextSetShadow(context, CGSize(width: 5, height: 5), 10)
            CGContextStrokeRect(context, frame)
        }
        let newImage = UIGraphicsGetImageFromCurrentImageContext()
        imageDraw.image = newImage
        UIGraphicsEndImageContext()
    }
}
```

The code of Listing 24-27 loads two images, clouds.jpg and husky.png, and draws them in the context. The clouds.jpg image is drawn first to use as a background, and the husky.png image is drawn last at the center of the context with a rectangle around it to make it look like a portrait. Once the graphics are done, the image is retrieved with the `UIGraphicsGetImageFromCurrentImageContext()` function and shown on the screen as always. The final result is illustrated in Figure 24-17.



Figure 24-17: *Graphics in an image context*

Do It Yourself: Download the `clouds.jpg` and `husky.png` images from our website and add them to the Assets Catalog. Update the `ViewController` class of our last example with the code of Listing 24-27. Run the application. You should see something similar to Figure 24-17.

The Basics: Because of Auto Layout, sometimes the values returned from the interface are not the current values but those defined in the Storyboard. If we want to make sure we get the right values, we have to call the view's `setNeedsLayout()` and `layoutIfNeeded()` methods. These methods tell the system that the layout is invalid and has to be calculated again. See Chapter 6 for more information.

The `UIGraphicsGetImageFromCurrentImageContext()` function returns an image of the size of the context, but we can use the `CGImageCreateWithImageInRect()` function introduced before to create an image from a piece of another image. The function works with `CGImage` values, so we have to get the `UIImage` object from the context and use the value returned by its `CGImage` property to extract the part of the image we want. The following example processes the `husky.png` image, extracts the rectangle with the husky's head, and shows it on the screen.

Listing 24-28: *Extracting an image from another image*

```
import UIKit
```

```
class ViewController: UIViewController {  
    @IBOutlet weak var imageDraw: UIImageView!  
    override func viewDidLoad() {  
        super.viewDidLoad()  
    }  
}
```

```

if let image = UIImage(named: "husky") {
    let width = image.size.width
    let height = image.size.height
    let imageSize = CGSize(width: width, height: height)
    UIGraphicsBeginImageContextWithOptions(imageSize, false, 0.0)
    image.drawAtPoint(CGPoint(x: 0, y: 0))

    let newImage = UIGraphicsGetImageFromCurrentImageContext()
    let screen = UIScreen.mainScreen()
    let scale = screen.scale

    let head = CGImageCreateWithImageInRect(newImage.CGImage, CGRect(x: 75 * scale, y: 15 * scale, width:
210 * scale, height: 200 * scale))
    imageDraw.image = UIImage(CGImage: head!, scale: screen.scale, orientation: .Up)
    UIGraphicsEndImageContext()
}
}
}

```

Images are loaded considering the scale of the screen, but **CGImage** values are processed using the real resolution in pixels. This means that every time we process an image in a **CGImage** format we have to consider the scale of the screen. The code of Listing 24-28 takes this into account before calling the **CGImageCreateWithImageInRect()** function. We get the current scale from the **scale** property of the main screen and then use it to calculate the right size of the rectangle provided to the function to extract the image. The values that determine the size of the rectangle are specified for a scale 1x but multiplied by the current scale to get the right amount of pixels.

Do It Yourself: Update the **ViewController** class with the code of Listing 24-28. Set the Image View mode as Aspect Fit and run the application. You should see on the screen only the part of the original image corresponding to the husky's head.

24.2 Core Animation

Core Animation is the framework in charge of rendering and animating the visual elements of an application. The drawing of the user interface and also basic transitions, like those generated by a Segue, are produced by code from this framework. Several high-level frameworks use the capacity of Core Animation to render the content they produce, but we can also implement this framework to modify the elements ourselves and create custom animations.

CALayer

Core Animation uses a system of layers to manage the graphics. When a view draws its content, the graphics are stored in a layer that can be processed by the graphics hardware and presented on the screen. These layers store a bitmap (a series of values that represent every pixel of the image) and also the information associated with it, like the position of the image, its size, etc. This process is very efficient and allows the system to produce smooth transitions and animations by just modifying the values associated with the layers involved.

Layers are created from a class provided by Core Animation called `CALayer`. The class includes the following properties to modify the layer's configuration.

position—This property sets or returns a `CGPoint` value with the layer's coordinates.

frame—This property sets or returns a `CGRect` value that determines the origin and size of the layer.

bounds—This property sets or returns a `CGRect` value that determines the origin and size of the layer in its own coordinate space.

anchorPoint—This property sets or returns a `CGPoint` value that determines the layer's anchor point. The coordinates are values between 0.0 and 1.0, with 0, 0 being the top-left corner of the layer. Any transformation applied to the layer occurs around the anchor point. By default the value is 0.5, 0.5, which positions the anchor point at the center of the layer.

cornerRadius—This property sets or returns a `CGFloat` value that determines the radius used to draw the corners of the layer.

backgroundColor—This property sets or returns a `CGColor` value that determines the layer's background color.

opacity—This property sets or returns a **Float** value that determines the layer’s opacity. The possible values are from 0.0 (transparent) to 1.0 (opaque).

hidden—This property sets or returns a **Boolean** value that determines whether the layer is visible or not.

borderWidth—This property sets or returns a **CGFloat** value that determines the width of the layer’s border. By default this value is set to 0.0 (no border).

borderColor—This property sets or returns a **CGColor** value that determines the color of the layer’s border.

shadowColor—This property sets or returns a **CGColor** value that determines the color of the layer’s shadow.

shadowOffset—This property sets or returns a **CGSize** value that determines the shadow’s offset.

shadowRadius—This property sets or returns a **CGFloat** value that determines the blur radius of the shadow.

shadowOpacity—This property sets or returns a **Float** value that determines the shadow’s opacity. The possible values are between 0.0 (transparent) and 1.0 (opaque).

In iOS, a layer is automatically assigned to each view. The **UIView** class includes the **layer** property to get a reference to the layer that belongs to the view. From this reference we can modify any property of the layer, as shown in the following example.

Listing 24-29: Modifying the layer

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var animationView: UIView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let layer = animationView.layer
        layer.borderColor = UIColor.blackColor().CGColor
        layer.borderWidth = 5
        layer.cornerRadius = 10
    }
}
```

This example assumes that we have added a view to the initial view of a Single View

Application project and the view is connected to the `ViewController` class with an Outlet called `animationView`. In the `viewDidLoad()` method, we assign new values to the layer's properties to generate a black border of 5 points wide and round corners. The result is shown in Figure 24-18 (for this example we have created a small view pinned to the center of the screen containing a label with the text “The Layer”).

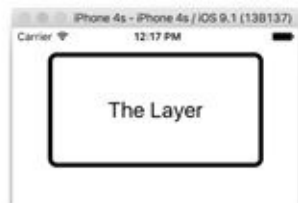


Figure 24-18: Layer with custom configuration

Do It Yourself: Create a new Single View Application project. Add a small view and pin it with constraints to the center and top of the main view. Add a label to the view with the text “The Layer”. Connect the view to the `ViewController` class with an Outlet called `animationView`. Complete the `ViewController` class with the code of Listing 24-29. Run the application. You should see something similar to Figure 24-18.

Layers can also contain sublayers. These are instances of `CALayer` added to the layer with a method provided by the class called `addSublayer()`. The following example adds a small layer with a round shape at the top-left corner of the layer.

Listing 24-30: Adding a sublayer

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var animationView: UIView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let layer = animationView.layer
        layer.borderColor = UIColor.blackColor().CGColor
        layer.borderWidth = 5
        layer.cornerRadius = 10

        let sublayer = CALayer()
        sublayer.frame = CGRect(x: 10, y: 10, width: 30, height: 30)
        sublayer.backgroundColor = UIColor.blackColor().CGColor
        sublayer.cornerRadius = 15
        layer.addSublayer(sublayer)
    }
}
```

```
}  
}
```



Figure 24-19: Sublayer

3D Transformations

Another attribute stored in a `CALayer` object is the transformation matrix for the layer. As we explained before, a transformation matrix specifies what points in one coordinate system correspond to the points in another coordinate system, but unlike the `CGAffineTransform` transformation matrix we used before in this chapter, layers work with the `CATransform3D` matrix, allowing us to transform the graphics in a tridimensional space.

The `CALayer` class includes a property called `transform` to define the `CATransform3D` matrix for the layer. Core Animation offers the following functions to get this value.

`CATransform3DTranslate(matrix, x, y, z)`—This function returns a transformation matrix of type `CATransform3D` that translates the layer in a tridimensional space. The **`matrix`** attribute is the matrix from which the new matrix is calculated and the **`x`**, **`y`**, and **`z`** attributes are values of type `CGFloat` with the coordinates of the translation.

`CATransform3DScale(matrix, x, y, z)`—This function returns a transformation matrix of type `CATransform3D` that scales the layer in a tridimensional space. The **`matrix`** attribute is the matrix from which the new matrix is calculated and the **`x`**, **`y`** and **`z`** attributes are values of type `CGFloat` with the new scale of every axis.

`CATransform3DRotate(matrix, angle, x, y, z)`—This function returns a transformation matrix of type `CATransform3D` that rotates the layer in a tridimensional space. The **`matrix`** attribute is the matrix from which the new matrix is calculated, the **`angle`** attribute is the rotation angle in radians, and the **`x`**, **`y`**, and **`z`** attributes determine the axis in which the rotation is going to take place. The value 0.0 cancels the rotation for that axis and values greater than 0 activate the rotation (the values frequently used are 0.0 and 1.0).

These functions calculate the new matrix from another matrix. When we are applying changes to a layer for the first time, we can use the identity transform matrix. This is a neutral value that represents the matrix of a layer in its initial state. The `CALayer` class includes the `CATransform3DIdentity` constant to return the identity transform matrix defined by Core Animation.

Listing 24-31: Transforming a layer

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var animationView: UIView!
    override func viewDidLoad() {
        super.viewDidLoad()
        let layer = animationView.layer
        layer.borderColor = UIColor.blackColor().CGColor
        layer.borderWidth = 5
        layer.cornerRadius = 10
        let degrees = CGFloat(10 * M_PI / 180)

        var matrix = CATransform3DIdentity
        matrix = CATransform3DRotate(matrix, degrees, 0, 0, 1)
        layer.transform = matrix
    }
}
```

The transformation operates on the three axes. For example, if we rotate the layer, we have to specify around which axis the rotation will be performed. In the example of Listing 24-31 we decided to rotate the layer clockwise 10 degrees on its **z** axis (the axis that determines depth and is defined by the line between the user and the screen). The result is shown in figure 24-20.



Figure 24-20: Layer transformed with a rotation matrix

The layer may be rotated around any of the three axes, **x**, **y**, and **z**. When the layer is rotated around the **z** axis, as we did in the example of Listing 24-31, it just turns clockwise

around the axis the amount of degrees we specified, but when the rotation is around the **x** or **y** axes, the transformation makes one side of the layer move to the back and the other side to the front. The problem is that in real life, when an object rotates on these axes, the side that goes to the back looks smaller and the side that goes to the front looks bigger. This effect is called *perspective*, but it is not included in the matrix produced by the methods provided by Core Animation. To add perspective to the transformation we have to define the values of the matrix ourselves.

A tridimensional matrix is composed by values placed in a grid of 4 columns and 4 rows. These values are represented by indexes according to their position in the matrix. The value at the top-left corner of the grid is identified with the index 1-1, the value on the left is identified with the index 1-2, and so on. To provide access to these values, the `CATransform3D` structure defines a series of properties, the names of which start with the letter **m** and end with the index numbers (e.g., **m11**, **m12**, etc.). To provide perspective to the rotation we have to modify the value at the position 3-4, which is represented by the **m34** property.

Listing 24-32: Adding perspective to the layer

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var animationView: UIView!
    override func viewDidLoad() {
        super.viewDidLoad()
        let layer = animationView.layer
        layer.borderColor = UIColor.blackColor().CGColor
        layer.borderWidth = 5
        layer.cornerRadius = 10
        let degrees = CGFloat(45 * M_PI / 180)

        var matrix = CATransform3DIdentity
        matrix.m34 = 1 / -500.0
        matrix = CATransform3DRotate(matrix, degrees, 0, 1, 0)
        layer.transform = matrix
    }
}
```

The value assigned to the **m34** property affects the skew factor of the transformation, altering the perspective. The further this value gets to 0 the deepest is the illusion of perspective. The formula to obtain the value divides 1 by the negative of the distance in the **z** axis. The distance could be determined by the size of the layer but it always depends on the effect we want to achieve. In Listing 24-32 we use the value -500.0 that is usually considered as the value by default in most transformations. The result is shown in Figure



Figure 24-21: Layer with perspective

CABasicAnimation

The most important feature of Core Animation is, of course, the capacity to animate layers. The animations are created with objects of the **CABasicAnimation** class provided by the framework. The class includes the following initializer.

CABasicAnimation(keyPath: **property)**—This initializer creates a **CABasicAnimation** object with an animation for the property defined by the **property** attribute. The attribute is a string with the name of the property we want to animate.

The **CABasicAnimation** class is a subclass of other classes like **CAAnimation** and **CAPropertyAnimation** that provide basic functionality for animations, and also adopts protocols like **CAMediaTiming** and **CAAction** to include parameters that determine the characteristics of the animation. The following are the most frequently used properties available from a **CABasicAnimation** object to configure the animation.

fromValue—This property defines the initial value of the layer’s property that is going to be animated. If no value is assigned to this property, the animation starts from the current value of the property.

toValue—This property defines the final value of the layer’s property that is going to be animated.

duration—This property defines the duration of the animation in seconds. It is a value of type **CFTimeInterval** (a typealias of **Double**).

A **CABasicAnimation** object defines an animation but it doesn’t execute it. The animation has to be added to the layer for the process to start. The **CALayer** class offers the following methods to add and remove animations.

addAnimation(animation, forKey: property)—This method adds the animation defined by the **animation** attribute to the layer. The **animation** attribute is the `CABasicAnimation` object that defines the animation we want to perform, and the **property** attribute is a string with the name of the property we want to animate.

removeAnimationForKey(property)—This method removes the animation associated with the property defined by the **property** attribute.

An animation can operate on any of the layer’s properties, but it doesn’t actually change the value of the property. If we want to preserve the value, we have to define it after the animation is added to the layer. The following example shows how to animate the layer of the view of our application.

Listing 24-33: Animating the layer

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var animationView: UIView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let layer = animationView.layer
        layer.borderColor = UIColor.blackColor().CGColor
        layer.borderWidth = 5
        layer.cornerRadius = 10
        let gesture = UITapGestureRecognizer(target: self, action: "animateView:")
        view.addGestureRecognizer(gesture)
    }

    func animateView(sender: UITapGestureRecognizer) {
        let layer = animationView.layer
        let animation = CABasicAnimation(keyPath: "opacity")
        animation.fromValue = 1.0
        animation.toValue = 0.0
        animation.duration = 2.0
        layer.addAnimation(animation, forKey: "opacity")
        layer.opacity = 0.0
    }
}
```

The code of Listing 24-33 adds a gesture to the main view that responds to a tap on the screen. The gesture recognizer calls the `animateView()` method where we create the animation and add it to the layer. The animation modifies the `opacity` property, taking the opacity of the layer from a value of 1.0 to a value of 0.0 in 2 seconds. At the end, the value of the layer's `opacity` property is modified to reflect the final value.

Do It Yourself: Update the `ViewController` class of your project with the code of Listing 24-33 and run the application. Tap on any place on the screen to start the animation. The view connected to the `animationView` Outlet should gradually vanish as the value of the `opacity` property gets close to 0.0.

UIView Animations

A `CABasicAnimation` object can affect only one property at a time and requires multiple lines of code to configure a simple animation. Because of the close relationship between layers and the views they belong to, Apple developers decided to simplify our work including a series of type methods in the `UIView` class to create and configure animations. These methods require only a few parameters and can manage multiple animations at a time. The most frequently used are variations of the same method called `animateWithDuration()` that provide different options for configuration.

`animateWithDuration(duration, animations: animations)`—This type method creates and executes single or multiple animations with the values defined in the `animations` block. The `duration` attribute is an `NSTimeInterval` value (a typealias of `Double`) that determines how much time the animations take to complete, and the `animations` attribute is a closure with statements that set the final values of the properties.

`animateWithDuration(duration, animations: animations, completion: completion)`—This type method creates and executes single or multiple animations with the values defined in the `animations` block. This version of the method adds the `completion` attribute to specify the statements we want to execute after the animations are completed.

`animateWithDuration(duration, delay: delay, options: options, animations: animations, completion: completion)`—This type method creates and executes single or multiple animations with the values defined in the `animations` block. This version of the method adds the `delay` and `options` attributes. The `delay` attribute is an `NSTimeInterval` value (a typealias of `Double`) that determines when the animations will begin, and the `options` attribute is a set of values of type

`UIViewAnimationOptions` used to configure different characteristics of the animation. There are several values available to configure animations and also transitions. For example, we can use the values `CurveEaseInOut`, `CurveEaseIn`, `CurveEaseOut`, and `CurveLinear` to determine how smooth the animation will be, or the values `AllowUserInteraction`, `Repeat` and `Autoreverse` to set different behaviours.

`animateWithDuration(duration, delay: delay, usingSpringWithDamping: damping, initialSpringVelocity: velocity, options: options, animations: animations, completion: completion)`—This type method creates and executes single or multiple animations with the values defined in the **`animations`** block. This version of the method adds the **`damping`** and **`velocity`** attributes to incorporate physics to the animation that make the objects look like they are boost by springs. The **`damping`** attribute is a `CGFloat` value that determines the oscillation (values close to 0 increase oscillation), and the **`velocity`** attribute is a `CGFloat` value that determines the velocity of the animation (a value of 1.0 corresponds to the elements travelling the total distance in 1 second).

The animations begin from the current state and move to the state specified by the statements inside the **`animations`** closure. The following example implements the simplest version of the `animateWithDuration()` method to animate a change in the transformation matrix of the view used in our previous examples. The animations start from the current matrix and end with the matrix defined by the `CATransform3DTranslate()` function.

Listing 24-34: Animating a transformation with UIView methods

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var animationView: UIView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let layer = animationView.layer
        layer.borderColor = UIColor.blackColor().CGColor
        layer.borderWidth = 5
        layer.cornerRadius = 10

        let gesture = UITapGestureRecognizer(target: self, action: "animateView:")
        view.addGestureRecognizer(gesture)
    }

    func animateView(sender: UITapGestureRecognizer) {
        let layer = animationView.layer
        var matrix = CATransform3DIdentity
        matrix = CATransform3DTranslate(matrix, 0.0, 200.0, 0.0)
```

```

UIView.animateWithDuration(2.0, animations: {
    layer.transform = matrix
})
}
}

```

The animation produced by the example of Listing 24-34 displaces the view from the top to the center of the screen in a uniform motion, but we can apply a more complex version of the `animateWithDuration()` method to include options and establish a different behaviour. In the following example we set the options `CurveEaseInOut` and `Autoreverse` to smooth and reverse the animation.

Listing 24-35: Configuring the animation's behaviour

```

func animateView(sender: UITapGestureRecognizer) {
    let layer = animationView.layer
    var matrix = CATransform3DIdentity
    matrix = CATransform3DTranslate(matrix, 0.0, 200.0, 0.0)

    UIView.animateWithDuration(2.0, delay: 0.0, options: [.CurveEaseInOut, .Autoreverse], animations: {
        layer.transform = matrix
    }, completion: { (finished: Bool) in
        layer.transform = CATransform3DIdentity
    })
}

```

The animations created from the `UIView` type methods set the properties to the last value processed by the animation. This means that we don't have to assign the last values to the properties at the end of the animation, as we did in the example of Listing 24-33. But this is not true when we use the `Autoreverse` option. This option forces the system to invert the animation but it doesn't reinstate the values of the properties. For this reason, if we want the interface to reflect what the users sees on the screen after the animation is over, we have to declare a completion handler and set the values ourselves. In the example of Listing 24-35 we include a closure that assigns the identity transform matrix to the `transform` property to effectively return the layer to its original state when the process is over.

Do It Yourself: Replace the `animateView()` method introduced in the example of Listing 24-34 by the method of Listing 24-35 and run the application. When you tap on the screen, the view should move down to the center and back to the top.

Because of the **Autoreverse** option, when the animation is over the system reverses the animation and moves the view back to its original position, creating a bouncing effect. The more complex version of the **animateWithDuration()** method allows us to produce similar effects but adds more precision.

Listing 24-36: Adding a spring effect

```
func animateView(sender: UITapGestureRecognizer) {  
    let layer = animationView.layer  
    var matrix = CATransform3DIdentity  
    matrix = CATransform3DTranslate(matrix, 0.0, 200.0, 0.0)  
  
    UIView.animateWithDuration(1.0, delay: 0.0, usingSpringWithDamping: 0.5, initialSpringVelocity: 10.0,  
options: [], animations: {  
        layer.transform = matrix  
    }, completion: nil)  
}
```

IMPORTANT: Changes in the layer are reflected in the view the layer belongs to. If you move a layer using its **position** property or with a translation, the system will ignore the current constraints and the view won't adapt anymore to changes in its superview (it won't adapt to the rotation of the screen, for example). If you need to keep the constraints active, you have to erase the current constraints, perform the animation, and create new ones when the animation is over. Another option is to add the views that are going to be animated from code and change their positions every time the **viewWillTransitionToSize()** method is called (see Chapter 6, Listing 6-9). You can also animate the constraints by connecting them to the view controller with Outlets and modifying their constant values, as we did in Chapter 6, Listing 6-5.

Chapter 25

Media

25.1 Video

Almost every mobile device is capable of playing videos these days, and Apple devices are no exception. Because of the importance of video, Apple provides a framework designed specifically to work with video and audio called *AV Foundation*. The framework defines several classes to create the structure necessary to play media. There is a class in charge of the asset (video or audio), a class in charge of the presentation of the media to the player, a class in charge of playing the media, and a class in charge of showing the media on the screen. Figure 25-1 illustrates this structure.

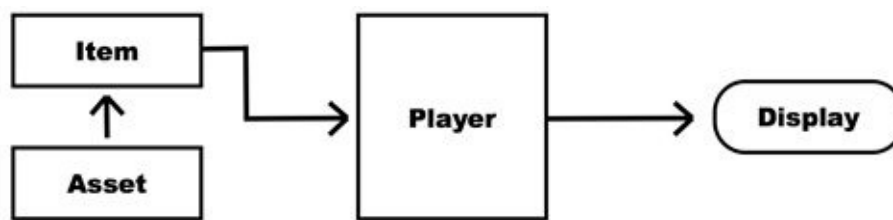


Figure 25-1: System to play media

AVAsset

The information of the media we want to play is called asset. An asset is conformed by one or more tracks of media, including video, audio, subtitles, etc. The AV Foundation framework defines a class called **AVAsset** to load an asset and a subclass called **AVURLAsset** to create an asset object from the contents of a URL. This last subclass includes the following initializer.

AVURLAsset(URL: URL)—This initializer creates an **AVURLAsset** object with the media in the location indicated by the **URL** attribute. The attribute is an **NSURL** object with the location of a local or remote source.

AVPlayerItem

An asset contains static information and can't manage its state when it is being played. To control the asset, the framework defines the **AVPlayerItem** class. With this class we can reference an asset and manage its timeline. The class includes initializers to create the object from an **AVAsset** object or directly from a URL for live stream.

AVPlayerItem(asset: **asset)**—This initializer creates an **AVPlayerItem** object to represent the asset defined by the **asset** attribute. The attribute is an **AVAsset** object.

AVPlayerItem(URL: **URL)**—This initializer creates an **AVPlayerItem** object to represent a live stream. The **URL** attribute is an **NSURL** object with the location of the live stream.

The **AVPlayerItem** class includes multiple properties and methods to control the state of the asset. The following are the most frequently used.

status—This property returns a value that indicates the status of the player item. It is an enumeration of type **AVPlayerItemStatus** with the values **Unknown**, **ReadyToPlay**, and **Failed**.

duration—This property returns a value that indicates the duration of the player item. It is a structure of type **CMTime**.

currentTime()—This method returns a **CMTime** value with the current time of the player item.

seekToTime(time)—This method moves the playback cursor to the time specified by the **time** attribute. The attribute is a structure of type **CMTime**.

seekToTime(time, toleranceBefore: **tolerance1, toleranceAfter: **tolerance2**, completionHandler: **block**)**—This method moves the playback cursor to the time specified by the **time** attribute. The **time** attribute is a **CMTime** structure that indicates the position on the timeline where we want to move the playback cursor, the **tolerance1** and **tolerance2** attributes are **CMTime** structures with the time we allow before and after the position indicated by the **time** attribute, and the **block** attribute is a closure with the statements we want to execute after the cursor is moved.

AVPlayer

The **AVPlayerItem** object manages the information necessary for playback but it doesn't play the media; this is done by an instance of the **AVPlayer** class. The class provides the following initializers to create the player.

AVPlayer(playerItem: **item)**—This initializer creates an **AVPlayer** object to play the media represented by the **item** attribute. The attribute is an object of type **AVPlayerItem**.

AVPlayer(URL: URL)—This initializer creates an **AVPlayer** object to play the **AVPlayerItem** object generated from the URL specified by the **URL** attribute. The attribute is an **NSURL** object with the location of the media.

The **AVPlayer** class also includes properties and methods to control the playback.

volume—This property sets or returns a value that determines the player’s volume. It is a value of type **Float** between 0.0 and 1.0.

muted—This property is a Boolean value that determines whether the player’s audio is muted or not.

rate—This property sets or returns a **Float** value that determines the rate at which the media is played. A value of 0.0 pauses the video and 1.0 sets the normal rate.

play()—This method begins playback.

pause()—This method pauses playback.

addPeriodicTimeObserverForInterval(interval, queue: queue, usingBlock: block)—This method adds an observer that executes a closure repeatedly every certain period of time. The **interval** attribute is a **CMTime** structure that determines the time between executions, the **queue** attribute is the queue in which the closure should be executed (the main queue is recommended), and the **block** attribute is the closure we want to execute.

AVPlayerLayer

The structure still requires another object to display the media on the screen. This is a subclass of **CALayer** called **AVPlayerLayer** that provides the code necessary to draw the media’s frames. The class includes the following initializer and property to create and configure the layer.

AVPlayerLayer(player: player)—This initializer creates an **AVPlayerLayer** object associated with the player specified by the **player** attribute. The attribute is a value of type **AVPlayer**.

videoGravity—This property defines how the video adjusts its size to the preview layer’s size. The framework defines the following constants to assign to this property:

`AVLayerVideoGravityResizeAspect,`
`AVLayerVideoGravityResize.`

`AVLayerVideoGravityResizeAspectFill,`

and

CMTime

Because the precision of floating-point values is not enough for media playback, the framework adopts, among other things, the `CMTime` structure defined in an old framework programmed in the C language called Core Media. The structure contains four values to represent time as a fraction. The most important are `value` and `timescale`, which represent the numerator and denominator, respectively. To determine the time we have to specify at least these two values. For example, if we want to create a `CMTime` structure to represent 0.5 seconds, we may declare 1 as the numerator and 2 as the denominator (1 divided by 2 is equal to 0.5). The framework defines several functions to create and retrieve values from a `CMTime` structure. The following are the most frequently used.

`CMTimeMake(value, timescale)`—This function returns a `CMTime` with the values specified by the `value` and `timescale` attributes. The attributes are integers of type `Int64` and `Int32`, respectively.

`CMTimeGetSeconds(time)`—This function returns a floating-point value with the time represented by a `CMTime` value expressed in seconds. The `time` attribute is the `CMTime` value we want to transform to seconds.

Video Player

The process to create a player is straightforward. We have to load the asset (`AVURLAsset`), create the item to manage the asset (`AVPlayerItem`), add the item to the player (`AVPlayer`), and associate the player to a layer to show the media on the screen (`AVPlayerLayer`). But playing the media requires an additional step. The media doesn't become immediately available after it is loaded and therefore we can't play it right away, we have to wait until its state is ready. The state is reported by the `status` property of the `AVPlayerItem` object, so we have to use KVO and add an observer for this property to start playing the media only after the property returns the value `ReadyToPlay`.

The following code demonstrates how to build a video player. The example assumes that we have added the `trailer.mp4` file to the project and have an interface with a view to display the video.

Listing 25-1: Building a video player

```
import UIKit
import AVFoundation

class ViewController: UIViewController {
    @IBOutlet weak var videoView: UIView!
    var playerItem: AVPlayerItem!
    var player: AVPlayer!
    var playerLayer: AVPlayerLayer!

    override func viewDidLoad() {
        super.viewDidLoad()

        let bundle = NSBundle.mainBundle()
        let videoURL = bundle.URLForResource("trailer", withExtension: "mp4")
        let asset = AVURLAsset(URL: videoURL!)
        playerItem = AVPlayerItem(asset: asset)
        playerItem.addObserver(self, forKeyPath: "status", options: [], context: nil)
        player = AVPlayer(playerItem: playerItem)
        playerLayer = AVPlayerLayer(player: player)
        playerLayer.frame = view.bounds
        let layer = videoView.layer
        layer.addSublayer(playerLayer)
    }

    override func observeValueForKeyPath(keyPath: String?, ofObject object: AnyObject?, change: [String : AnyObject]?, context: UnsafeMutablePointer<Void>) {
        if keyPath == "status" {
            if playerItem.status == .ReadyToPlay {
                playerItem.removeObserver(self, forKeyPath: "status")
                player.play()
            }
        }
    }

    override func viewWillTransitionToSize(size: CGSize, withTransitionCoordinator coordinator: UIViewControllerTransitionCoordinator) {
        super.viewWillTransitionToSize(size, withTransitionCoordinator: coordinator)
        playerLayer.frame.size = size
    }
}
```

In Listing 25-1 we load the video from the bundle and create the player structure as soon as the view is loaded. The player is associated with an **AVPlayerLayer** layer and the layer is added as a sublayer of the view where we want to show the video, but the video

doesn't start until the value of the `status` property changes to `ReadyToPlay` (see Key/Value Observing in Chapter 18).

The size of the layer for the player is defined according to the size of the view (`playerLayer.frame = view.bounds`). Because this value is not determined by constraints, we have to update it ourselves when the device is rotated. This is why at the end we added the `viewWillTransitionToSize()` method. Before rotating the interface, the system calls this method to report the new size the view is going to adopt. Changing the layer's size to this value we effectively adapt the video to any orientation.

Do It Yourself: Create a new Single View Application project. Add an empty view to the initial view and pin it to the sides with Space constraints. Connect the view to the `ViewController` class with an Outlet called `videoView`. Complete the class with the code of Listing 25-1. Download the file `trailer.mp4` from our website and add it to your project. Run the application. The video should start playing as soon as the application is launched.

The previous example plays the video but it doesn't provide any tools for the user to control the process. The `AVPlayer` class includes methods to play, pause, and check the state of the media, but we are responsible for creating the interface. Figure 25-2, next, introduces an interface with the view to show the video and an additional view on top of it with two controls, a button and a Progress View, to let the user play and pause the video and see its progression.

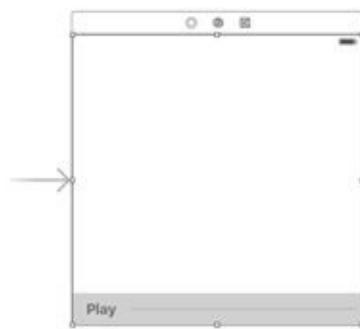


Figure 25-2: *Controllers for a custom video player*

How we control the process and respond to the interface depends on the requirements of our application. For this example we decided to define two properties to keep track of the state of the media. The `ready` property will be `true` when the media is ready to be played and the `playing` property will be `true` while the media is being played.

Listing 25-2: *Preparing the video player*

```
import UIKit
```

```

import AVFoundation

class ViewController: UIViewController {
    @IBOutlet weak var videoView: UIView!
    @IBOutlet weak var controlBar: UIView!
    @IBOutlet weak var playButton: UIButton!
    @IBOutlet weak var progressBar: UIProgressView!

    var playerItem: AVPlayerItem!
    var player: AVPlayer!
    var playerLayer: AVPlayerLayer!

    var ready = false
    var playing = false

    override func viewDidLoad() {
        super.viewDidLoad()

        playButton.enabled = false
        controlBar.backgroundColor = UIColor(red: 0.6, green: 0.6, blue: 0.6, alpha: 0.25)
        let bundle = NSBundle.mainBundle()
        let videoURL = bundle.URLForResource("trailer", withExtension: "mp4")
        let asset = AVAsset(URL: videoURL!)
        playerItem = AVPlayerItem(asset: asset)
        playerItem.addObserver(self, forKeyPath: "status", options: [], context: nil)
        player = AVPlayer(playerItem: playerItem)
        playerLayer = AVPlayerLayer(player: player)
        playerLayer.frame = view.bounds
        let layer = videoView.layer
        layer.addSublayer(playerLayer)
    }
}

```

In Listing 25-2 we add a few more Outlets to the **ViewController** class to reference the elements of the interface and disable the Play button until the video is ready to play, but the code to prepare the player remains the same. The difference is in how the video is played, because this time we have to wait until the user presses the Play button. This change is reflected in the **observeValueForKeyPath()** method. Instead of playing the video when the value of the **status** property changes to **ReadyToPlay**, we assign the value **true** to the **ready** property to let the rest of the code know that the video is ready. But this is not all the method does. We also have to register an observer to keep the progress bar updated while the video is being played. Normal observers are not fast enough, so the AV Foundation framework offers the **addPeriodicTimeObserverForInterval()** method to create an observer that will provide a more accurate response.

Listing 25-3: Registering the observer for the progress bar

```
override fun observeValueForKeyPath(keyPath: String?, ofObject object: AnyObject?, change: [String : AnyObject]?,
context: UnsafeMutablePointer<Void>) {
    if keyPath == "status" {
        if playerItem.status == .ReadyToPlay {
            ready = true
            playButton.enabled = true
            playerItem.removeObserver(self, forKeyPath: "status")

            let mainQueue = dispatch_get_main_queue()
            let interval = CMTimeMake(1, 2)

            player.addPeriodicTimeObserverForInterval(interval, queue: mainQueue, usingBlock: { (time) in
                let current = CMTimeGetSeconds(time)
                let duration = CMTimeGetSeconds(self.playerItem.duration)
                let position = current / duration
                self.progressBar.progress = Float(position)
            })
        }
    }
}
```

As we mentioned before, AV Foundation takes advantage of old frameworks programmed in the C language like Core Media. Another C library used by AV Foundation is Grand Central Dispatch (GCD). This is an old library that provides similar functionality as the `NSOperation` class. Among other functions, the library includes the `dispatch_get_main_queue()` function to return a reference to the main queue. In Listing 25-3 we store this reference in the `mainQueue` constant, create a `CMTime` value to represent a time of 0.5 seconds, and then call the `addPeriodicTimeObserverForInterval()` method with this information to register the observer. After this, the closure provided to the observer will be executed every 0.5 seconds during playback. In this closure we get the current time and the duration of the video in seconds, and calculate the right position of the progress bar by turning seconds into a value between 0.0 and 1.0 (the default minimum and maximum values of a Progress View).

The player and the progress bar are ready, so the only thing left is to add the Action for the Play button. This is a simple method that just has to execute the player's `play()` or `pause()` methods depending on the value of the `playing` property.

Listing 25-4: Playing and pausing the video

```
@IBAction func playVideo(sender: UIButton) {
```



```

if ready {
    if playing {
player.pause()
        playing = false
        playButton.setTitle("Play", forState: .Normal)
    } else {
player.play()
        playing = true
        playButton.setTitle("Pause", forState: .Normal)
    }
}
}
}

```



Figure 25-3: Custom video player in action

© Copyright 2008, Blender Foundation / www.bigbuckbunny.org

Do It Yourself: Add a second view to the initial view to work as the control bar for the video player. Position this view on top of the video’s view and at the bottom, and include a button called Play and a Progress View inside (Figure 25-2). Connect the view, the button, and the Progress View with Outlets called **controlBar**, **playButton**, and **progressBar**. Connect the Play button with an Action called **playVideo()**. Update the **ViewController** class with the codes of Listing 25-2, 25-3, and 25-4. Remember to add the **viewWillTransitionToSize()** method of Listing 25-1 to adapt the layer to different orientations. Run the application and press the Play button. You should see something similar to Figure 25-3.

IMPORTANT: You cannot only play videos from files added to the project (local) but also from the web (remote). All you need to do is to create an **NSURL** object with the web URL pointing to the file you want to play and use it as the source for the asset. There is a restriction though. You can only play remote files that are stored in secure servers (only URLs starting with **https://**). In Chapter 27 we will learn how to create **NSURL** objects to access remote documents and how to configure the application to be able to work with unsecure URLs.

The observer added by the `addPeriodicTimeObserverForInterval()` method is not the only observer we can have for our player. The `AVPlayerItem` class defines several notifications to report events that happened during playback. For example, we can add an observer to the `AVPlayerItemDidPlayToEndTimeNotification` notification to know when the video finishes playing.

Listing 25-5: Adding and observer to detect the end of the video

```
let center = NSNotificationCenter.defaultCenter()
center.addObserver(self, selector: "videoEnded:", name: AVPlayerItemDidPlayToEndTimeNotification, object:
playerItem)
```

The method that responds to the notification has to move the playback cursor back to the beginning and prepare the interface to let the user play the video again.

Listing 25-6: Restarting the video

```
func videoEnded(notification: NSNotification) {
    playerItem.seekToTime(kCMTimeZero)
    playing = false
    playButton.setTitle("Play", forState: .Normal)
}
```

The `seekToTime()` method implemented in Listing 25-6 to rewind the video uses a constant instead of a `CMTime` value. The `kCMTimeZero` is defined in the Core Media framework and its purpose is to return a `CMTime` value with the time initialized to 0.

Do It Yourself: Add the code of Listing 25-5 to the `viewDidLoad()` method and the `videoEnded()` method of Listing 25-6 to the `ViewController` class. Run the application, play the video, and wait until it is over. Press the Play button to play the video again.

AVKit

The video player created in the previous example illustrates how to create controls for the video player, but also shows how much code we have to write to provide all the tools a modern player demands. We still have to add a gesture recognizer to detect a tap in the progress bar and move the playback cursor to the position selected by the user, add labels to show the current time and duration, buttons to jump to the next video, buttons to cancel

the operation, etc. There is a lot we can add to our player, but this work is usually not necessary. Because most applications provide a similar interface, Apple decided to include a small framework called AVKit, which includes all the classes necessary to create a standard video player. The framework defines the **AVPlayerViewController** class to create a controller that presents the player in a new view. The class includes the following initializer.

AVPlayerViewController()—This initializer creates an **AVPlayerViewController** object. The controller includes the **player** property to define the **AVPlayer** object with the media we want to play.

The **AVPlayerViewController** controller takes care of creating the view and the controls to play the video. All it needs is an **AVPlayer** object to know what to play. The following example shows how to create, configure, and open this controller.

Listing 25-7: Using the standard video player

```
import UIKit
import AVFoundation
import AVKit

class ViewController: UIViewController {
    @IBAction func playVideo(sender: UIButton) {
        let bundle = NSBundle.mainBundle()
        let videoURL = bundle.URLForResource("trailer", withExtension: "mp4")
        let player = AVPlayer(URL: videoURL!)
        let controller = AVPlayerViewController()
        controller.player = player
        presentViewController(controller, animated: true, completion: {
            player.play()
        })
    }
}
```

This example assumes that we have a button in the initial view connected to an Action in the **ViewController** class called **playVideo()**. When the button is pressed, the method creates the **AVPlayer** object from a URL, initializes the **AVPlayerViewController** controller, assigns the player object to the controller's **player** property, and presents the controller on the screen. In this example we use the completion handler of the **presentViewController()** method to play the video as soon as the view is loaded, but we can let the user decide.

Do It Yourself: Create a new Single View Application project. Add a button called Play Video to the initial view. Connect the button to an Action called `playVideo()`. Complete the `ViewController` class with the code of Listing 25-7. Download the trailer.mp4 video from our website and add it to your project. Run the application and press the button. A new view is shown on the screen with a standard video player. Press the Done button in the view to remove it.

25.2 Audio

Although the `AVPlayer` class can play video and audio, the AV Foundation framework defines specific classes for audio alone. The class we need to instantiate to create an audio player is called `AVAudioPlayer`. The following are some of its initializers, properties, and methods.

`AVAudioPlayer(contentsOfURL: URL)`—This initializer creates an `AVAudioPlayer` object with an audio player configured to play the media in the location specified by the `URL` attribute.

`volume`—This property sets or returns a `Float` value between 0.0 and 1.0 that determines the volume of the player.

`duration`—This property returns the duration of the audio in seconds. It is a value of type `NSTimeInterval` (a typealias of `Double`).

`currentTime`—This property sets or returns the position of the playback cursor in seconds. It is a value of type `NSTimeInterval` (a typealias of `Double`).

`numberOfLoops`—This property sets or returns an integer that determines the number of times the audio player is going to play the audio.

`playing`—This property is a `Boolean` value that indicates whether the audio is currently being played or not.

`play()`—This method plays the audio assigned to the audio player.

`pause()`—This method pauses the audio player.

`stop()`—This method stops the audio player.

`prepareToPlay()`—This method preloads the audio and connects the audio player to the hardware to minimize the lag when the audio starts playing.

`playAtTime(time)`—This method moves the playback cursor to the time indicated by the `time` attribute. The attribute is a value of type `NSTimeInterval` (a typealias of `Double`).

When playing audio, we don't need to check for the state of the media, we can play it right away, but it is recommended to get the media ready as soon as the view is loaded and execute the `prepareToPlay()` method to avoid any delay. The following example loads an mp3 file from the bundle called `beach.mp3` and let the user play, stop, and pause the player.

Listing 25-8: Playing audio

```
import UIKit
import AVFoundation

class ViewController: UIViewController {
    var audioPlayer: AVAudioPlayer!
    override func viewDidLoad() {
        super.viewDidLoad()

        let bundle = NSBundle.mainBundle()
        let audioURL = bundle.URLForResource("beach", withExtension: "mp3")
        audioPlayer = try? AVAudioPlayer(contentsOfURL: audioURL!)
        if audioPlayer != nil {
            audioPlayer.prepareToPlay()
        }
    }

    @IBAction func playAudio(sender: UIButton) {
        if let player = audioPlayer {
            player.play()
        }
    }

    @IBAction func stopAudio(sender: UIButton) {
        if let player = audioPlayer {
            player.stop()
            player.currentTime = 0
        }
    }

    @IBAction func pauseAudio(sender: UIButton) {
        if let player = audioPlayer {
            player.pause()
        }
    }
}
```

The example of Listing 25-8 assumes that we have created an interface with three buttons to play, stop, and pause the player, and they are connected to the Actions `playAudio()`, `stopAudio()`, and `pauseAudio()`, respectively. The code creates the `AVAudioPlayer` objects and prepares the media to be played. When the user presses any of the buttons, the right methods from this object are called. Notice that the `stop()` method doesn't rewind the media and we have to do it ourselves by modifying the `currentTime` property. The difference between `stop()` and `pause()` is that the first one resets the player and frees the hardware.

Do It Yourself: Create a new Single View Application project. Add three buttons called Play, Stop, and Pause to the initial view. Connect the buttons to Actions called `playAudio()`, `stopAudio()`, and `pauseAudio()`. Complete the `ViewController` class with the code of Listing 25-8. Download the `beach.mp3` file from our website and add it to your project. Run the application and press Play.

AVAudioPlayerDelegate

AV Foundation defines the `AVAudioPlayerDelegate` protocol to report changes in the state of the audio player. The protocol includes the following methods.

`audioPlayerDidFinishPlaying(player, successfully: flag)`—This method is called on the delegate when the audio has finished playing.

`audioPlayerDecodeErrorDidOccur(player, error: error)`—This method is called on the delegate when an error occurs while processing the audio.

The `AVAudioPlayer` class includes the `delegate` property to assign a delegate to the audio player and being able to implement the protocol's methods we need. The `ViewController` class in the following example adopts the protocol and implements the `audioPlayerDidFinishPlaying(player, successfully: flag)` method to modify the interface when the audio finishes playing. The example assumes that we have only one button to stop and play the audio. Every time the button is pressed, we have to check the `playing` property to know the state of the player and respond accordingly.

Listing 25-9: Detecting when the audio has finished playing

```
import UIKit
import AVFoundation

class ViewController: UIViewController, AVAudioPlayerDelegate {
    @IBOutlet weak var playButton: UIButton!
    var audioPlayer: AVAudioPlayer!

    override func viewDidLoad() {
        super.viewDidLoad()

        let bundle = NSBundle.mainBundle()
        let audioURL = bundle.URLForResource("beach", withExtension: "mp3")
        audioPlayer = try? AVAudioPlayer(contentsOfURL: audioURL!)
```

```

    if audioPlayer != nil {
audioPlayer.delegate = self
        audioPlayer.prepareToPlay()
    }
}

func audioPlayerDidFinishPlaying(player: AVAudioPlayer, successfully flag: Bool) {
    playButton.setTitle("Play", forState: .Normal)
}

@IBAction func playAudio(sender: UIButton) {
    if let player = audioPlayer {
        if player.playing {
            playButton.setTitle("Play", forState: .Normal)
            player.stop()
            player.currentTime = 0
        } else {
            playButton.setTitle("Stop", forState: .Normal)
            player.play()
        }
    }
}
}

```

Do It Yourself: Erase the buttons in the previous interface and empty the **ViewController** class. Add a single button to the initial view with the title Play. Connect the button to the **ViewController** class with an Outlet called **playButton** and an Action called **playAudio()**. Complete the **ViewController** class with the code of Listing 25-9 and run the application. The title of the Play button should change every time you press it and when the audio finishes playing.

Recording

AV Foundation also offers the possibility to record audio. The class to create an audio recorder is called **AVAudioRecorder**. The following is the initializer for this class.

AVAudioRecorder(URL: URL, settings: settings)—This initializer creates an audio recorder object. The **URL** attribute is an **NSURL** object with the location of the file where the audio will be stored, and the **settings** attribute is a dictionary with the settings for the recording session. The framework defines multiple keys for configuration. The most frequently used are **AVFormatIDKey** (audio format), **AVEncoderAudioQualityKey** (audio

quality), `AVEncoderBitRateKey` (audio bit rate), `AVSampleRateKey` (sample rate in hertz), and `AVNumberOfChannelsKey` (number of audio channels).

The `AVAudioRecorder` class includes the following properties and methods to configure and manage the recording.

currentTime—This property returns the time in seconds since the beginning of the recording. It is a value of type `NSTimeInterval` (a typealias of `Double`).

recording—This property returns a Boolean value that indicates if the recorder is recording.

prepareToRecord()—This method creates the audio file and prepares the system for recording.

record()—This method begins or resumes recording.

recordAtTime(time)—This method begins recording at the time specified by the **time** attribute. The attribute is a value of type `NSTimeInterval` (a typealias of `Double`).

recordForDuration(duration)—This method begins recording and ends at the time specified by the **duration** attribute (in seconds). The attribute is a value of type `NSTimeInterval` (a typealias of `Double`).

recordAtTime(time, forDuration: duration)—This method begins recording at the time specified by the **time** attribute and ends at the time specified by the **duration** attribute (in seconds). The attributes are values of type `NSTimeInterval` (a typealias of `Double`).

stop()—This method stops the recording.

pause()—This method pauses the recording. To continue recording we have to call the `record()` method again.

As well as the `AVAudioPlayer` class, the `AVAudioRecorder` class can designate a delegate to report the state of the recording. The delegate must adopt the `AVAudioRecorderDelegate` protocol and implement any of its methods.

audioRecorderDidFinishRecording(recorder, successfully: flag)—This method is called on the delegate when the recording is over.

audioRecorderEncodeErrorDidOccur(recorder, error: error)—This

method is called on the delegate when an error occurs.

The system creates an audio session that connects the application to the hardware and is shared by every app installed on the device. By default, this session is not configured to work with the microphone. To record sound we have to get a reference to the audio session and change its configuration. The session is created from an object of the `AVAudioSession` class, which includes the following type method to get a reference.

sharedInstance()—This type method returns a reference to the shared audio session.

The following are the methods provided by the `AVAudioSession` class to configure the audio session and ask for permission to use the microphone.

setCategory(category)—This method sets the session's category. A category determines the behaviour of the audio session. The framework defines the following constants to describe it: `AVAudioSessionCategoryAmbient`, `AVAudioSessionCategorySoloAmbient` (by default), `AVAudioSessionCategoryPlayback`, `AVAudioSessionCategoryRecord`, `AVAudioSessionCategoryPlayAndRecord`, `AVAudioSessionCategoryAudioProcessing` and `AVAudioSessionCategoryMultiRoute`.

setMode(mode)—This method sets the session's mode. The mode determines how the audio is going to be processed. The framework defines the following constants to describe it: `AVAudioSessionModeDefault`, `AVAudioSessionModeVoiceChat`, `AVAudioSessionModeGameChat`, `AVAudioSessionModeVideoRecording`, `AVAudioSessionModeMeasurement`, `AVAudioSessionModeMoviePlayback`, `AVAudioSessionModeVideoChat` and `AVAudioSessionModeSpokenAudio`.

requestRecordPermission(block)—This method asks the user's permission to use the microphone. The **block** attribute is a closure to be executed after the user accepts or declines the request (permission is requested only once). The closure receives a Boolean value to indicate if the user granted permission or not.

recordPermission()—This method returns the current permission status. The value returned is an enumeration of type `AVAudioSessionRecordPermission` with the values `Undetermined`, `Denied`, and `Granted`.

setActive(activate)—This method activates or deactivates the audio session. The **activate** attribute is a Boolean value. The value `true` means activate and `false` deactivate.

As always, we have to set up the system first and then record or play the audio in response to feedback from the user. To prepare the audio recorder, we have to create a

URL pointing to the file in which the recording is going to be stored, configure the audio session to work with the microphone, create the **AVAudioRecorder** object with the settings we want, and call the **prepareToRecord()** method to get it ready to start recording.

Listing 25-10: Setting up the recorder

```
import UIKit
import AVFoundation

class ViewController: UIViewController, AVAudioPlayerDelegate {
    @IBOutlet weak var recordButton: UIButton!
    @IBOutlet weak var playButton: UIButton!
    var audioRecorder: AVAudioRecorder!
    var audioPlayer: AVAudioPlayer!

    override func viewDidLoad() {
        super.viewDidLoad()

        playButton.enabled = false
        let manager = NSFileManager.defaultManager()
        let documents = manager.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomainMask)
        let docURL = documents.first!
        let audioURL = docURL.URLByAppendingPathComponent("myaudio.caf")

        let audioSession = AVAudioSession.sharedInstance()
        do {
            try audioSession.setCategory(AVAudioSessionCategoryPlayAndRecord)
            do {
                let recordSettings: [String: AnyObject] = [AVEncoderAudioQualityKey: AVAudioQuality.Medium.rawValue,
                AVEncoderBitRateKey: 16, AVSampleRateKey: 44100.0, AVNumberOfChannelsKey: 2]
                audioRecorder = try AVAudioRecorder(URL: audioURL, settings: recordSettings)
            } catch {
                print("Error")
            }
            if audioRecorder != nil {
                audioRecorder.prepareToRecord()
            }
        } catch {
            print("Error")
        }
    }
}
```

In the example of Listing 25-10 we call the audio file myaudio.caf. The caf extension is

short for Core Audio Format, an audio container created by Apple and designed to contain audio in different formats, like AAC, MP3, etc. Once the file's URL is created, we set the category for the audio session as `AVAudioSessionCategoryPlayAndRecord` to be able to record and also play the recorded audio later, and then establish the settings for the recorder. The settings include values for different aspects of the audio, including the level of quality, which is defined with an enumeration value of type `AVAudioQuality` that includes the values **Min**, **Low**, **Medium**, **High**, and **Max** (we have to provide the raw value for this parameter).

The `prepareToRecord()` method prepares the recorder, but doesn't begin recording. This is done by the `record()` method that we call when the user or our application decides it is time to do it. For this purpose, our example includes two Actions to let the user decide whether to begin recording or play the audio recorded previously (our example assumes that we have an interface with two buttons called Record and Play connected to these Actions).

Listing 25-11: Recording and playing the audio recorded

```
@IBAction func recordAudio(sender: UIButton) {
    if let recorder = audioRecorder {
        if recorder.recording {
            recorder.stop()
            playButton.enabled = true
            recordButton.setTitle("Record", forState: .Normal)
        } else {
            recorder.record()
            playButton.enabled = false
            recordButton.setTitle("Stop", forState: .Normal)
        }
    }
}

@IBAction func playAudio(sender: UIButton) {
    if let player = audioPlayer {
        if player.playing {
            playButton.setTitle("Play", forState: .Normal)
            recordButton.enabled = true
            player.stop()
            player.currentTime = 0
        } else {
            startPlayer()
        }
    } else {
        startPlayer()
    }
}
```

```

func startPlayer() {
    if let audioURL = audioRecorder?.url {
        audioPlayer = try? AVAudioPlayer(contentsOfURL: audioURL)
        if audioPlayer != nil {
            playButton.setTitle("Stop", forState: .Normal)
            recordButton.enabled = false
            audioPlayer.delegate = self
            audioPlayer.play()
        }
    }
}

func audioPlayerDidFinishPlaying(player: AVAudioPlayer, successfully flag: Bool) {
    playButton.setTitle("Play", forState: .Normal)
    recordButton.enabled = true
}

```

The Action methods of Listing 25-11 complement the **ViewController** class of Listing 25-10. The procedure is similar to the one we used before. Every time a button is pressed, we check the current condition of the recorder or the player and execute the methods to begin or stop the recording or playback. Notice that for the recorder we check the condition with the **recording** property and for the player we use the **playing** property, as we did in previous examples.

Do It Yourself: Create a new Single View Application project. Add two buttons called Record and Play to the initial view. Connect the buttons to the **ViewController** class with Outlets called **recordButton** and **playButton** and Actions called **recordAudio()** and **playAudio()**, respectively. Complete the **ViewController** class with the codes of Listings 25-10 and 25-11. Run the application on your device and press Record. Say something and press Stop. Now press the Play button to hear the recorded audio.

25.3 Media

Besides playing audio files incorporated to our application, we can also access and play the media already on the device. There is a framework we can use for this purpose called Media Player. This framework provides access to the media added to the device from iTunes, including songs, audio podcasts, audio books, etc. The framework offers two alternatives to let the user select what to play: we can use a predefined controller or generate a media query from our own code.

Media Controller

Media Player defines the `MPMediaPickerController` class to create a view controller with a predefined interface to select media items. The class includes the following initializer.

`MPMediaPickerController(mediaTypes: type)`—This initializer creates a media controller configured for the media type specified by the `type` attribute. The attribute is an enumeration of type `MPMediaType` that defines the type of media we want to retrieve. The possible values are `Music`, `Podcast`, `AudioBook`, `AudioITunesU`, `AnyAudio`, `Movie`, `TVShow`, `VideoPodcast`, `MusicVideo`, `VideoITunesU`, `HomeVideo`, `AnyVideo`, and `Any`.

The `MPMediaPickerController` class works with a delegate to inform our view controller the decisions made by the user. The framework defines the `MPMediaPickerControllerDelegate` protocol for this delegate with the following methods.

`mediaPicker(mediaPicker, didPickMediaItems: mediaItemCollection)`—This method is called on the delegate when the user selects a media item. The `mediaItemCollection` attribute is an object of type `MPMediaItemCollection` that contains a list of `MPMediaItem` objects that represent each item selected.

`mediaPickerDidCancel(mediaPicker)`—This method is called on the delegate when the user presses the Cancel button to close the controller.

When the user selects an item, the controller returns an object of type `MPMediaItemCollection` that contains a collection of `MPMediaItem` objects. To retrieve the items, the `MPMediaItemCollection` class offers the following properties.

items—This property returns an array with the **MPMediaItem** objects in the collection.

count—This property returns an integer with the number of items in the collection.

The items returned are objects of type **MPMediaItem**. The class includes the following method to retrieve the item's attributes (the method belongs to the **MPMediaItem** superclass called **MPMediaEntity**).

valueForProperty(property)—This method returns the value of the item's property specified by the **property** attribute. The attribute is a string that represents the item's property. The class defines constants for each property available. The most frequently used are **MPMediaItemPropertyAssetURL**, **MPMediaItemPropertyMediaType**, **MPMediaItemPropertyTitle**, **MPMediaItemPropertyAlbumTitle**, **MPMediaItemPropertyArtist**, **MPMediaItemPropertyAlbumArtist**, **MPMediaItemPropertyPlaybackDuration**, **MPMediaItemPropertyArtwork**, and **MPMediaItemPropertyReleaseDate**.

To include this functionality in our application, we have to initialize the **MPMediaPickerController** object, present it on the screen like we do with other presentation view controllers before, and implement the methods of the **MPMediaPickerControllerDelegate** protocol to get the media item selected by the user.

Listing 25-12: Playing songs stored in the device

```
import UIKit
```

```
import MediaPlayer
```

```
class ViewController: UIViewController, MPMediaPickerControllerDelegate {
```

```
    @IBOutlet weak var playButton: UIButton!
```

```
    @IBOutlet weak var titleLabel: UILabel!
```

```
    var audioPlayer: AVAudioPlayer!
```

```
    @IBAction func selectSong(sender: UIButton) {
```

```
        let controller = MPMediaPickerController(mediaTypes: .Music)
```

```
        controller.delegate = self
```

```
        presentViewController(controller, animated: true, completion: nil)
```

```
    }
```

```
    func mediaPicker(mediaPicker: MPMediaPickerController, didPickMediaItems mediaItemCollection: MPMediaItemCollection) {
```

```
        let item = mediaItemCollection.items[0]
```

```
        let mediaURL = item.valueForProperty(MPMediaItemPropertyAssetURL) as! NSURL
```

```
        audioPlayer = try? AVAudioPlayer(contentsOfURL: mediaURL)
```

```
        if audioPlayer != nil {
```

```

        titleLabel.text = item.valueForProperty(MPMediaItemPropertyTitle) as? String
        audioPlayer.prepareToPlay()
        playButton.setTitle("Play", forState: .Normal)
    }
    dismissViewControllerAnimated(true, completion: nil)
}

func mediaPlayerDidCancel(mediaPicker: MPMediaPickerController) {
    dismissViewControllerAnimated(true, completion: nil)
}

@IBAction func playSong(sender: UIButton) {
    if let player = audioPlayer {
        if player.playing {
            playButton.setTitle("Play", forState: .Normal)
            player.stop()
            player.currentTime = 0
        } else {
            playButton.setTitle("Stop", forState: .Normal)
            player.play()
        }
    }
}
}

```

The example of Listing 25-12 assumes that we have an interface with two buttons called Select Song and Play to select a song and play it, and a label to show the name of the song to the user. When the Select Song button is pressed, the `selectSong()` method is executed. In this method we create the `MPMediaPickerController` controller to retrieve Music items, declare the `ViewController` object as its delegate, and present it on the screen with the `presentViewController()` method. The view includes all the tools necessary for the user to perform a selection. Once the user selects an item, the `mediaPicker()` method is executed and we can read the item's properties from the `mediaItemCollection` attribute. In our example we get the first item in the collection (we only work with one item this time) and read its `MPMediaItemPropertyAssetURL` property to get the URL pointing to the item. With this URL we can create the `AVAudioPlayer` object we need to play the audio. In this method we also read the `MPMediaItemPropertyTitle` property and assign its value to the label on the interface to show the song's title on the screen. At the end, the view controller is dismissed, taking the user back to the initial view where the Play button is ready to play the song (the Action for this button is the same we used in the example of Listing 25-9 to play an audio file).

Do It Yourself: Create a new Single View Application project. Add two buttons to the initial view called Select Song and Play. Add a label with the message "No Song

Selected”. Connect the Play button with an Outlet called `playButton` and an Action called `playSong()`. Connect the label with an Outlet called `titleLabel` and the Select Song button with an Action called `selectSong()`. Complete the `ViewController` class with the code of Listing 25-12. Run the application on your device. When you press the Select Song button, a view with an interface to select a song should open. Select a song and then press Play to play it.

Query Media

The media controller is an easy way to provide access to the user’s media on the device, but we can also create our custom picker or search for specific media items using a query. The class that provides this functionality is `MPMediaQuery`. We can create a generic query to retrieve all the media items available, use a predicate to filter the items, or execute type methods to get a query configured to return a specific type of media. The following are the initializer and some of the type methods available.

`MPMediaQuery(filterPredicates: predicate)`—This initializer creates an `MPMediaQuery` object with the predicate determined by the **`predicate`** attribute. The attribute is an `MPMediaPropertyPredicate` object. We can also declare the initializer with no attributes and assign the predicate later with the object’s `addFilterPredicate(predicate)` method.

`albumsQuery()`—This type method creates a `MPMediaQuery` object that returns media items sort by album.

`artistsQuery()`—This type method creates a `MPMediaQuery` object that returns media items sort by the name of the artists.

`songsQuery()`—This type method creates a `MPMediaQuery` object that returns media items sort by the name of the songs.

`playlistsQuery()`—This type method creates a `MPMediaQuery` object that returns media items sort by the name of the playlist they belong to.

`podcastsQuery()`—This type method creates a `MPMediaQuery` object that returns podcast items sort by name.

`audiobooksQuery()`—This type method creates a `MPMediaQuery` object that returns audio books sort by name.

The `MPMediaQuery` class offers properties to configure the query and return the items.

items—This property returns an array with the media items found by the query. The items are objects of type **MPMediaItem**.

groupingType—This property determines how the media items are going to be grouped to create collections. It is an enumeration of type **MPMediaGrouping** with the values **Title** (value by default), **Album**, **Artist**, **AlbumArtist**, **Composer**, **Genre**, **Playlist**, and **PodcastTitle**.

collections—This property returns an array of **MPMediaItemCollection** objects that contain a list of **MPMediaItem** objects that represent the items in the collection.

Predicates are defined with an object of the **MPMediaPropertyPredicate** class. The class includes the following initializer.

MPMediaPropertyPredicate(value: value, forProperty: property)—This initializer creates an **MPMediaPropertyPredicate** object with a predicate that matches the property defined by the **property** attribute, and with the value specified by the **value** attribute. The **value** attribute is the value we want the property of the media to have, and the **property** attribute is one of the constants that represent the properties of an **MPMediaItem** object. The most frequently used are **MPMediaItemPropertyAssetURL**, **MPMediaItemPropertyMediaType**, **MPMediaItemPropertyTitle**, **MPMediaItemPropertyAlbumTitle**, **MPMediaItemPropertyArtist**, **MPMediaItemPropertyAlbumArtist**, **MPMediaItemPropertyPlaybackDuration**, **MPMediaItemPropertyArtwork**, and **MPMediaItemPropertyReleaseDate**.

With media queries we can get a list of media items of certain type and present them on the screen for the user to choose from. The following example searches for a song with the title “again” and lets the user play it or prints an error message on the console if it can’t find it.

Listing 25-13: Searching for a song

```
import UIKit
import MediaPlayer

class ViewController: UIViewController {
    @IBOutlet weak var playButton: UIButton!
    var audioPlayer: AVAudioPlayer!

    override func viewDidLoad() {
        super.viewDidLoad()
```

```

let predicate = MPMediaPropertyPredicate(value: “again”, forProperty: MPMediaItemPropertyTitle)
let mediaItems = MPMediaQuery(filterPredicates: [predicate])
if let items = mediaItems.items {
    if items.count > 0 {
        let song = items[0]
        let songURL = song.valueForProperty(MPMediaItemPropertyAssetURL) as! NSURL
        audioPlayer = try? AVAudioPlayer(contentsOfURL: songURL)
    if audioPlayer != nil {
        audioPlayer.prepareToPlay()
        }
        } else {
        print(“Song not found”)
        }
    }
}

@IBAction func playSong(sender: UIButton) {
    if let player = audioPlayer {
        if player.playing {
            playButton.setTitle(“Play”, forState: .Normal)
            player.stop()
            player.currentTime = 0
        } else {
            playButton.setTitle(“Stop”, forState: .Normal)
            audioPlayer.play()
        }
    }
}

```

After creating the **MPMediaQuery** object all we need to do is to read its **items** property to get a list of the media items retrieved by the query. In Listing 25-13 we check if there are items available with the **count** property, get the first item, and create an **AVAudioPlayer** object with its **URL**. The example assumes that we have a Play button connected to an Action called **playSong()**. When the button is pressed, the song returned by the query is played.

Do It Yourself: Create a new Single View Application project. Add a button called Play to the initial view and connect it to the **ViewController** class with an Outlet called **playButton** and an Action called **playSong()**. Complete the **ViewController** class with the code of Listing 25-13. Replace the name of the song in the predicate with the name of a song you have in your device and run the application.

Chapter 26

Pictures and Videos

26.1 Image Picker Controller

Taking and storing pictures are the most common uses of mobile devices, and this is why no device is currently sold without a camera anymore. Because of how normal it is for an application to access the camera and manage pictures, UIKit offers a controller with built-in functionality to provide all the tools necessary for the user to take pictures and videos or select them from storage. The class to create this controller is called `UIImagePickerController`. The following are the properties included in this class for configuration.

sourceType—This property sets or returns a value that determines the type of source we want to use to get the pictures. It is an enumeration of type `UIImagePickerControllerSourceType` with the values `Camera` (for the camera), `PhotoLibrary` (for all the photos available on the device), and `SavedPhotosAlbum` (for the photos taken by the camera).

mediaTypes—This property sets or returns a value that determines the type of media we want to work with. It takes an array of strings with the values that represent every media we want to use. The possible values are `public.image` for pictures, `com.apple.live-photo` for live photos, and `public.movie` for movies.

cameraCaptureMode—This property sets or returns a value that determines the capture mode used by the camera. It is an enumeration of type `UIImagePickerControllerCameraCaptureMode` with the values `Photo` and `Video`.

cameraFlashMode—This property sets or returns a value that determines the flash mode used by the camera. It is an enumeration of type `UIImagePickerControllerCameraFlashMode` with the values `On`, `Off`, and `Auto`.

allowsEditing—This property sets or returns a Boolean value that determines if the user is allowed to edit the image.

videoQuality—This property sets or returns a value that determines the quality of the recorded video. It is an enumeration of type `UIImagePickerControllerQualityType` with the values `TypeHigh`, `TypeMedium`, `TypeLow`, `Type640x480`, `TypeIFrame960x540`, and `TypeIFrame1280x720`.

The `UIImagePickerController` class also offers the following type methods to detect the source of images available (Camera and Photo Library) and the type of media the sources can manage.

isSourceTypeAvailable(source)—This type method returns a Boolean value to indicate if the source specified by the **source** attribute is supported by the device. The

attribute is an enumeration of type `UIImagePickerControllerSourceType` with the values `Camera` (for the camera), `PhotoLibrary` (for all the photos available on the device), and `SavedPhotosAlbum` (for the photos taken by the camera).

availableMediaTypesForSourceType(source)—This type method returns an array with strings that represent the media types available for the source specified by the **source** attribute. The attribute is an enumeration of type `UIImagePickerControllerSourceType` with the values `Camera` (for the camera), `PhotoLibrary` (for all the photos available on the device), and `SavedPhotosAlbum` (for the photos taken by the camera).

isCameraDeviceAvailable(camera)—This type method returns a Boolean value to indicate if the camera specified by the **camera** attribute is available on the device. The attribute is an enumeration of type `UIImagePickerControllerCameraDevice` with the values `Rear` and `Front` to identify each camera.

The `UIImagePickerController` class creates a new view where the user can take pictures or select old ones from storage (depending on how we have configured the controller). After the image is created or selected, the view controller has to be dismissed and the selected image processed. The way our code gets access to this image and is able to know when to dismiss the view controller is through a delegate. We have to declare the view controller that opens the `UIImagePickerController` controller as its delegate and implement the following methods.

imagePickerController(controller, didFinishPickingMediaWithInfo: info)—This method is called on the delegate when the user finishes taking or selecting the image or video. The **info** attribute contains a dictionary with the information about the media. The values in the dictionary are identified with predefined keys. The most frequently used are `UIImagePickerControllerMediaType`, `UIImagePickerControllerOriginalImage`, `UIImagePickerControllerEditedImage`, and `UIImagePickerControllerLivePhoto`.

imagePickerControllerDidCancel(controller)—This method is called on the delegate when the user cancels the process.

Camera

The interface for these types of applications has to include an action that the user can perform to open the controller. For this example we have decided to create a simple interface with an Image View to show the image taken or selected, and a button called Get Picture to open the controller.

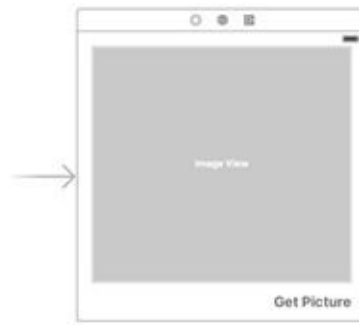


Figure 26-1: Interface to work with the camera

The view controller that is going to be the delegate of the `UIImagePickerController` controller is required to adopt two protocols, `UINavigationControllerDelegate` and `UIImagePickerControllerDelegate`. The following example includes an Action for the Get Picture button that creates, configures, and presents a `UIImagePickerController` controller to take pictures from the camera. The class also implements the methods defined in the `UIImagePickerControllerDelegate` protocol to process the media received from the controller.

Listing 26-1: Taking a picture

```
import UIKit
```

```
class ViewController: UIViewController, UINavigationControllerDelegate, UIImagePickerControllerDelegate {
    @IBOutlet weak var pictureView: UIImageView!

    @IBAction func takePicture(sender: UIButton) {
        let mediaPicker = UIImagePickerController()
        mediaPicker.delegate = self

        let sourceAvailable = UIImagePickerController.isSourceTypeAvailable(.Camera)

        if sourceAvailable {
            mediaPicker.sourceType = .Camera
            mediaPicker.mediaTypes = ["public.image"]
            mediaPicker.allowsEditing = false
            mediaPicker.cameraCaptureMode = .Photo
            presentViewController(mediaPicker, animated: true, completion: nil)
        } else {
            print("The media is not available")
        }
    }

    func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String :
AnyObject]) {
        let picture = info[UIImagePickerControllerOriginalImage] as! UIImage
        pictureView.image = picture
        dismissViewControllerAnimated(true, completion: nil)
    }
}
```



```

}
func imagePickerControllerDidCancel(picker: UIImagePickerController) {
    dismissViewControllerAnimated(true, completion: nil)
}
}

```

When the button is pressed, the `takePicture()` method creates an instance of the `UIImagePickerController` class and assigns the `ViewController` object as its delegate. Next, it checks if the camera is available and presents the controller in case of success or shows a message on the console otherwise. Before calling the `presentViewController()` method to show the view on the screen, the code configures the controller to work with the camera and only take pictures. The `sourceType` property is assigned the value `Camera` to tell the controller that we are going to get the picture from the camera, the `mediaTypes` property is assigned an array with the value `public.image` to set images as the media we want to retrieve, the `allowEditing` property is set as `false` to not let the user edit the image, and the `cameraCaptureMode` is assigned the value `Photo` to only let the user take pictures. The interface created by the controller is shown in Figure 26-2 (right).

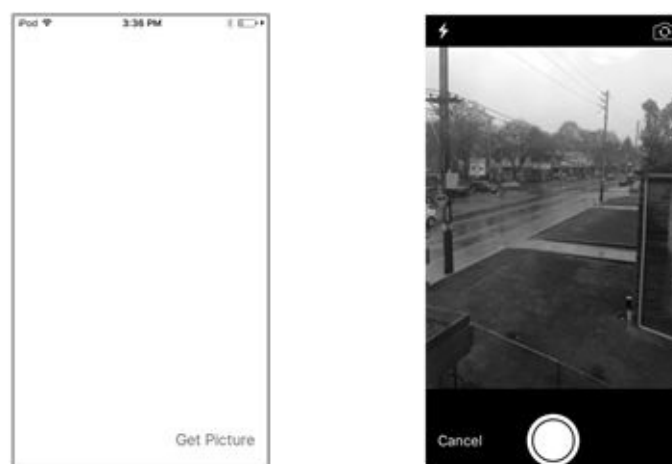


Figure 26-2: Camera's interface

The camera's interface includes buttons to control the camera and take the picture. After the user takes the picture, a new set of buttons is shown to let the user select the picture or take another one. If the user decides to use the current picture, the controller calls the `imagePickerController(controller, didFinishPickingMediaWithInfo: info)` method on its delegate to report the action. This method receives a parameter called `info` that we can read to get the media returned by the controller and process it (store it in a file, Core Data, or show it on the screen). In our example, we read the value of the `UIImagePickerControllerOriginalImage` key to get a `UIImage` object that represents the image taken or selected by the user, and assign this object to the Image View to show the picture on the screen.

Do It Yourself: Create a new Single View Application project. Add an Image View and a button to the initial view, as shown in Figure 26-1. Connect the Image View to the **ViewController** class with an Outlet called **pictureView** and the button with an Action called **takePicture()**. Complete the **ViewController** class with the code of Listing 26-1. Run the application on a device and press the button. Take the picture and press the button to use it. You should see on the screen the photo you just took with the camera.

The Basics: Most devices include a camera these days, but if you try this example in the simulator, you will see a message printed on the console that says, “The media is not available”. Instead of this message, you should display an Alert View to inform the user that the camera was not accessible in case there is a problem with the device or the application is being executed in a new device that doesn’t include a camera.

Photos Library

Except for the value of the **sourceType** property, the rest of the configuration values used in the previous example are the ones defined by default, so if all we want to do is to let the user take a picture, we don’t need to declare anything else. The same goes for the Photo Library. If we want to let the user select a picture from a list of images available on the device, we just have to indicate a different source changing the value of the **sourceType** property. There are two options available: **PhotoLibrary** and **SavedPhotosAlbum**. The **PhotoLibrary** option shows all the photos organized in a list of albums, and the **SavedPhotosAlbum** option gives priority to the photos in the camera roll.

Listing 26-2: Selecting a picture

```
import UIKit

class ViewController: UIViewController, UINavigationControllerDelegate, UIImagePickerControllerDelegate {
    @IBOutlet weak var pictureView: UIImageView!

    @IBAction func takePicture(sender: UIButton) {
        let mediaPicker = UIImagePickerController()
        mediaPicker.delegate = self
        mediaPicker.sourceType = .SavedPhotosAlbum
        presentViewController(mediaPicker, animated: true, completion: nil)
    }

    func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String :
AnyObject]) {
        let picture = info[UIImagePickerControllerOriginalImage] as! UIImage
        pictureView.image = picture
    }
}
```

```

dismissViewControllerAnimated(true, completion: nil)
}
func imagePickerControllerDidCancel(picker: UIImagePickerController) {
    dismissViewControllerAnimated(true, completion: nil)
}
}

```

In Listing 26-2 we assigned the value `SavedPhotosAlbum` to the `sourceType` property to let the user select a photo from the camera roll. With this simple change, instead of showing a view with an interface to operate the camera, the system now opens a view with a list of the pictures available. When the user selects a picture, the `imagePickerController()` method is called on the controller's delegate and the picture is shown on the screen.



Figure 26-3: Photo's interface

The photos library may contain images and also videos. We can define the type of media we want the controller to display assigning a new value to the `mediaTypes` property and then read the value for the `UIImagePickerControllerMediaType` key to select the media we want to process.

Listing 26-3: *Selecting images and videos*

```

import UIKit

class ViewController: UIViewController, UINavigationControllerDelegate, UIImagePickerControllerDelegate {
    @IBOutlet weak var pictureView: UIImageView!

    @IBAction func takePicture(sender: UIButton) {
        let mediaPicker = UIImagePickerController()
        mediaPicker.delegate = self
        mediaPicker.sourceType = .SavedPhotosAlbum
    }
}

```

```

    mediaPicker.mediaTypes = ["public.image", "public.movie"]
    presentViewController(mediaPicker, animated: true, completion: nil)
}

func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String :
AnyObject]) {
    let media = info[UIImagePickerControllerMediaType] as! String
    if media == "public.image" {
        let picture = info[UIImagePickerControllerOriginalImage] as! UIImage
        pictureView.image = picture
    }
    dismissViewControllerAnimated(true, completion: nil)
}

func imagePickerControllerDidCancel(picker: UIImagePickerController) {
    dismissViewControllerAnimated(true, completion: nil)
}
}

```

IMPORTANT: Instead of the strings “public.image”, “com.apple.live-photo”, and “public.movie”, you can use the constants `kUTTypeImage`, `kUTTypeLivePhoto`, and `kUTTypeMovie` defined in the Mobile Core Services framework. You have to import the framework (`import MobileCoreServices`) and cast the constants as `String` values (e.g., `kUTTypeImage as String`).

Adapting the Controller

By default, the view managed by the `UIImagePickerController` controller is presented full screen, but we can take advantage of the bigger screens offered by iPads and show the view in a popover. All we have to do is to configure the presentation style of the view controller as `Popover` and set the parameters to let the system position the popover on the screen.

Listing 26-4: Showing the view in a popover

```

import UIKit

class ViewController: UIViewController, UINavigationControllerDelegate, UIImagePickerControllerDelegate {
    @IBOutlet weak var pictureView: UIImageView!
    @IBOutlet weak var pictureButton: UIButton!

    @IBAction func takePicture(sender: UIButton) {
        let mediaPicker = UIImagePickerController()

```

```

mediaPicker.delegate = self
mediaPicker.sourceType = .SavedPhotosAlbum

mediaPicker.modalPresentationStyle = .Popover
if let popover = mediaPicker.popoverPresentationController {
    popover.sourceView = pictureButton
    popover.sourceRect = pictureButton.bounds
    popover.permittedArrowDirections = [.Down]
}
presentViewController(mediaPicker, animated: true, completion: nil)
}

func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String :
AnyObject]) {
    let picture = info[UIImagePickerControllerOriginalImage] as! UIImage
    pictureView.image = picture
    dismissViewControllerAnimated(true, completion: nil)
}

func imagePickerControllerDidCancel(picker: UIImagePickerController) {
    dismissViewControllerAnimated(true, completion: nil)
}
}

```

In this example we anchor the popover to the Get Picture button, so the view is going to show up over the button, close to where the user's finger is at the moment. If we open the view in a device with a small screen, the system shows it full screen, but if we do it on an iPad, the system creates a popover presentation controller and assigns it to the **popoverPresentationController** property. Reading this property we know how the controller is going to be shown and are able to set the values for the presentation, as we did for Action Sheets in Chapter 17, Listing 17-6.

Do It Yourself: Connect the Get Picture button to the **ViewController** class with an Outlet called **pictureButton**. Update the **ViewController** class with the code of Listing 26-4. Run the application on an iPad or on the iPad simulator and press the Get Picture button. The view should be opened in a popover.

Storing Pictures

In the previous examples we show the picture on the screen, but we can store it in a file or using Core Data. An alternative, sometimes useful, is to store the picture in the device's photos library to make it accessible from other applications. The UIKit framework offers

two functions to store images and videos.

UIImageWriteToSavedPhotosAlbum(image, target, action, info)—This function adds the image specified by the **image** attribute to the camera roll. The **image** attribute is a **UIImage** object with the picture we want to add, the **target** attribute is a reference to the object that contains the method for the completion action, the **action** attribute is a string that represents the method that is going to be executed when the process is over, and the **info** attribute is an object with additional data to pass to the method.

UISaveVideoAtPathToSavedPhotosAlbum(path, target, action, info)—This function adds the video at the path indicated by the **path** attribute to the camera roll. The **path** attribute is the path of the video we want to add, the **target** attribute is a reference to the object that contains the method for the completion action, the **action** attribute is a string that represents the method that is going to be executed when the process is over, and the **info** attribute is an object with additional data to pass to the method.

These functions store the picture or the video taken by the camera in the Photos Library and then call the method assigned to the **action** attribute to report the result of the operation. We can declare the **target** and **action** attributes as **nil** if we don't want to know what happened, or define a method in our view controller to handle the response. This method is required to have a specific definition that includes three attributes. For images the definition is **image:didFinishSavingWithError:contextInfo:** and for videos is **video:didFinishSavingWithError:contextInfo:..** In the following example we show on the screen the picture taken with the camera, store it in the Photos Library, and open an Alert View in case of success to inform the user that the picture is available with the rest of the photos.

Listing 26-5: Storing pictures in the Photo Library

```
import UIKit

class ViewController: UIViewController, UINavigationControllerDelegate, UIImagePickerControllerDelegate {
    @IBOutlet weak var pictureView: UIImageView!

    @IBAction func takePicture(sender: UIButton) {
        let mediaPicker = UIImagePickerController()
        mediaPicker.delegate = self
        let sourceAvailable = UIImagePickerController.isSourceTypeAvailable(.Camera)
        if sourceAvailable {
            mediaPicker.sourceType = .Camera
            presentViewController(mediaPicker, animated: true, completion: nil)
        } else {
```

```

        print("The media is not available")
    }
}

func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String :
AnyObject]) {
    let picture = info[UIImagePickerControllerOriginalImage] as! UIImage
    pictureView.image = picture
    UIImageWriteToSavedPhotosAlbum(picture, self, "image:didFinishSavingWithError:contextInfo:", nil)
    dismissViewControllerAnimated(true, completion: nil)
}

func imagePickerControllerDidCancel(picker: UIImagePickerController) {
    dismissViewControllerAnimated(true, completion: nil)
}

func image(image: UIImage, didFinishSavingWithError error: NSError?, contextInfo:UnsafePointer<Void>) {
    if error == nil {
        let alert = UIAlertController(title: "Picture Saved", message: "The picture was added to your photos",
preferredStyle: .Alert)
        let action = UIAlertAction(title: "OK", style: .Default, handler: nil)
        alert.addAction(action)
        presentViewController(alert, animated: true, completion: nil)
    } else {
        print("Error")
    }
}
}
}

```

Do It Yourself: Update the `ViewController` class with the code of Listing 26-5. If you erase the Outlet for the button created in Listing 26-4, remember to delete it also from the button in the Storyboard (see Figure 5-23). Run the application and take a picture. You should see an Alert View with the message "Picture Saved" and the picture should be available in your device's Photos Library.

Modifying Pictures

Due to the limited capacity of mobile devices, we can't work with multiple images in their original size at the same time. Pictures have to be immediately stored and in some cases reduced in size for future use. A common practice is to store the original picture along with a thumbnail to use for preview and selection. The following example creates a thumbnail of the picture taken by the camera.

Listing 26-6: Creating a thumbnail

```
import UIKit
```

```
class ViewController: UIViewController, UINavigationControllerDelegate, UIImagePickerControllerDelegate {
    @IBOutlet weak var pictureView: UIImageView!

    @IBAction func takePicture(sender: UIButton) {
        let mediaPicker = UIImagePickerController()
        mediaPicker.delegate = self
        let sourceAvailable = UIImagePickerController.isSourceTypeAvailable(.Camera)
        if sourceAvailable {
            mediaPicker.sourceType = .Camera
            presentViewController(mediaPicker, animated: true, completion: nil)
        } else {
            print("The media is not available")
        }
    }

    func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String :
    AnyObject]) {
        let picture = info[UIImagePickerControllerOriginalImage] as! UIImage
        let scale = UIScreen.mainScreen().scale
        let maximum: CGFloat = 80
        var width = picture.size.width / scale
        var height = picture.size.height / scale
        if width > height {
            height = height * maximum / width
            width = maximum
        } else {
            width = width * maximum / height
            height = maximum
        }
        UIGraphicsBeginImageContextWithOptions(CGSize(width: width, height: height), true, scale)
        picture.drawInRect(CGRect(x: 0, y: 0, width: width, height: height))
        let thumbnail = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()

        pictureView.image = thumbnail
        dismissViewControllerAnimated(true, completion: nil)
    }

    func imagePickerControllerDidCancel(picker: UIImagePickerController) {
        dismissViewControllerAnimated(true, completion: nil)
    }
}
```


The code of Listing 26-6 takes the picture returned by the camera, calculates the width and height considering a maximum size of 80 points, and creates a new image with this information. Notice that the `UIImage` returned by the `UIImagePickerControllerOriginalImage` key is set in a 1x scale, so we have to calculate its size in points dividing the width and height by the scale of the device. The rest of the process doesn't present anything new. In this example we show the thumbnail on the screen, but these values are usually stored using files or Core Data, as we did with the covers and thumbnails of the books in the example of Listing 22-5.

Do It Yourself: Update the `ViewController` class with the code of Listing 26-6. Set the Mode for the Image View as Top Left, so you can see the thumbnail on the screen in its original size.

26.2 Custom Controllers

The `UIImagePickerController` controller is built from classes defined in two frameworks: AV Foundation and Photos. The AV Foundation framework provides the codes necessary to process media and control input devices, like the camera and the microphone, and the Photos framework provides access to the user's photos in the device. We can use the classes in these frameworks directly to build our own controller and customize the process and the interface.

Camera

Creating our own controller to access the camera and retrieve information from that device demands the manipulation and coordination of several systems. We need to configure the input from at least two devices, the camera and the microphone, process the data received from this input, show a preview to the user, and also generate the output in the form of a still image, video, or audio. Figure 26-4 illustrates all the elements involved.

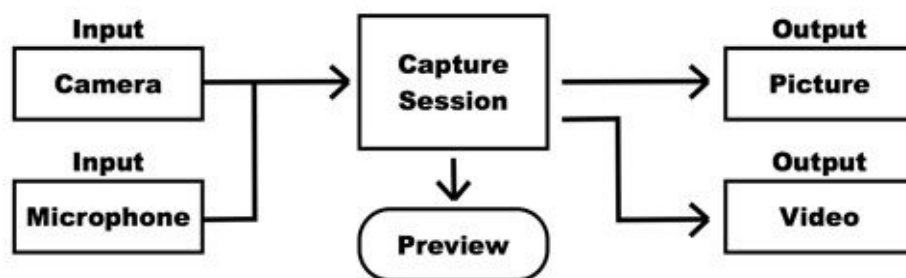


Figure 26-4: System to capture media

The first thing we need to do to build this structure is to determine the input devices. The AV Foundation framework defines the `AVCaptureDevice` class for this purpose. An instance of this class can represent any type of input device, including cameras and microphones. The following are the type methods included in the class to access and manage a device.

defaultDeviceWithMediaType(type)—This type method returns an `AVCaptureDevice` object that represents the default capture device for the media specified by the **type** attribute. The **type** attribute is a constant defined by the framework. To work with the cameras and microphones, we can specify two different constants: `AVMediaTypeVideo` or `AVMediaTypeAudio`.

devicesWithMediaType(type)—This type method returns an array of

AVCaptureDevice objects that represent the devices of the type specified by the **type** attribute. The **type** attribute is a constant defined by the framework. To work with the cameras and microphones, we can specify two different constants: **AVMediaTypeVideo** or **AVMediaTypeAudio**.

requestAccessForMediaType(type, completionHandler: block)—This type method requests to the user access to the device (we can only use the device after the user allows us to do it). The **type** attribute is a constant defined by the framework that represents the media we want to access. To work with the cameras and microphones, we can specify two different constants: **AVMediaTypeVideo** or **AVMediaTypeAudio**. The **block** attribute is a closure with the statements we want to execute after the request is completed.

authorizationStatusForMediaType(type)—This type method returns a value that determines the status of the authorization to use the device. The **type** attribute is a constant defined by the framework that represents the media we want to access. To work with the cameras and microphones, we can specify two different constants: **AVMediaTypeVideo** or **AVMediaTypeAudio**. The method returns an enumeration value of type **AVAuthorizationStatus**. The possible values are **NotDetermined**, **Restricted**, **Denied**, and **Authorized**.

An instance of the **AVCaptureDevice** class represents a capture device. To define this device as an input device, we have to create an object that controls the ports and connections of the input. The framework defines the **AVCaptureDeviceInput** class for this purpose. The class includes the following initializer to create the input object for the device.

AVCaptureDeviceInput(device: device)—This initializer creates an input for the device specified by the **device** attribute. The attribute is an object of type **AVCaptureDevice**.

As well as inputs, we also have to define outputs to process the data captured by the device. The framework defines subclasses of a generic class called **AVCaptureOutput** to describe the outputs. There are several subclasses available. The most frequently used are **AVCaptureVideoDataOutput** to process the frames of a video being captured, **AVCaptureMovieFileOutput** to create a QuickTime movie file, **AVCaptureAudioDataOutput** to get audio, and **AVCaptureStillImageOutput** to capture a single video frame (take a picture).

To control the flow of data from input to output, the framework defines the **AVCaptureSession** class. From an instance of this class we can control the inputs and outputs and also when the process begins and ends by calling the following methods.

addInput(input)—This method adds an input to the capture session. The **input**

attribute is an object of type `AVCaptureDeviceInput` that represents the input device we want to add.

addOutput(output)—This method adds an output to the capture session. The **output** attribute is an object of type `AVCaptureOutput` that represents the output we want to generate from the capture session.

startRunning()—This method starts the capture session.

stopRunning()—This method stops the capture session.

The framework also defines the `AVCaptureVideoPreviewLayer` class with the specific purpose of showing a preview to the user. This is a subclass of `CALayer` that creates a sublayer with predefined code to display the video captured by the input device. The class includes the following initializer and properties to create and manage the preview layer.

AVCaptureVideoPreviewLayer(session: session)—This initializer creates an `AVCaptureVideoPreviewLayer` object with a preview layer connected to the capture session defined by the **session** attribute. The attribute is an object of type `AVCaptureSession`.

videoGravity—This property defines how the video adjusts its size to the preview layer's size. The frameworks define the following constants to assign to this property: `AVLayerVideoGravityResizeAspect`, `AVLayerVideoGravityResizeAspectFill`, and `AVLayerVideoGravityResize`.

connection—This property returns an object of type `AVCaptureConnection` that defines the connection between the capture session and the preview layer.

The input, outputs, and preview layer are connected to the capture session by objects of the `AVCaptureConnection` class. The class manages the information of the connection, including ports and data. The following are its most frequently used properties.

videoOrientation—This property sets or returns the orientation of the video. It is an enumeration of type `AVCaptureVideoOrientation` with the values `Portrait`, `PortraitUpsideDown`, `LandscapeRight`, and `LandscapeLeft`.

supportsVideoOrientation—This property returns a Boolean value that determines whether or not it is allowed to set the video's orientation.

Because we are not presenting the view generated by the `UIImagePickerController` class anymore, we have to create our own. Figure 26-5 shows the interface we have created for this example. The initial view was embedded in a Navigation Controller and an additional

view was included to manage the camera. The initial view includes an Image View to show the picture taken, and a bar button called Get Picture to open the second view. The second view, on the other hand, has an empty view to which we can add the preview layer, and a narrow view at the bottom added over the preview view with a button to let the user take a picture.

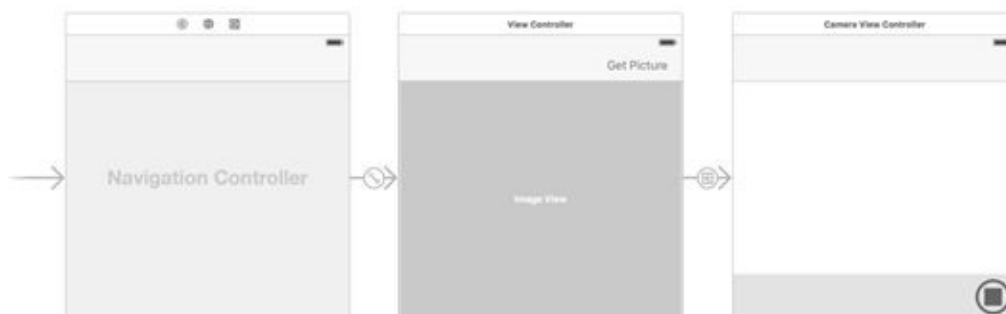


Figure 26-5: Custom controller for the camera

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add an Image View and a bar button with the title Get Picture to the initial view (Figure 26-5, center). Add a new view to the Storyboard. Connect the Get Picture button to the second view with a Show Segue. Add an empty view to the second view with a gray background, an alpha value of 0.5, and a Height constraint of 64 points. Pin this view to the bottom of the second view and add a button at its right side. Download the images camerabutton.png and camerabuttondown.png from our website and assign them to the button for the Default and Highlighted state. Add another empty view for the preview layer and pin it to the edges of the main view to adopt the size of the screen (below the Navigation Bar and over the view with the button). Open the Document Outline and drag this view over the first view on the list to move it to the back. The interface should look like Figure 26-5.

The view controller for the initial view is the simplest. Here we just have to get the picture from the second view and show it on the screen. For this example, we have decided to create an Action that we are going to connect to an Unwind Segue to get the image from the second view and assign it to the Image View.

Listing 26-7: Showing on the screen the picture taken by the camera

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var pictureView: UIImageView!

    @IBAction func goBack(segue: UIStoryboardSegue) {
```

```

let controller = segue.sourceViewController as! CameraViewController
pictureView.image = controller.picture
}
}

```

For the second view we have created a subclass of `UIViewController` called `CameraViewController`. In this view controller, we have to follow a series of steps to activate the camera, let the user take a picture, and process that image. But before to even access the camera we have to ask the user's permission. This is done automatically when we use a `UIImagePickerController` controller, but we have to do it ourselves in a custom controller using the type methods provided by the `AVCaptureDevice` class.

We start the `CameraViewController` class by defining the properties to manage the capture session, the output and the preview layer, and then check if we are authorized to use the camera.

Listing 26-8: Asking permission to use the camera

```

import UIKit
import AVFoundation

class CameraViewController: UIViewController {
    @IBOutlet weak var cameraView: UIView!
    var captureSession: AVCaptureSession!
    var stillImage: AVCaptureStillImageOutput!
    var previewLayer: AVCaptureVideoPreviewLayer!
    var picture: UIImage!

    override func viewDidLoad() {
        super.viewDidLoad()

        let status = AVCaptureDevice.authorizationStatusForMediaType(AVMediaTypeVideo)
        if status == AVAuthorizationStatus.Authorized {
            prepareCamera()
        } else if status == AVAuthorizationStatus.NotDetermined {
            AVCaptureDevice.requestAccessForMediaType(AVMediaTypeVideo, completionHandler: { (granted: Bool) in
                NSOperationQueue.mainQueue().addOperationWithBlock({
                    if granted {
                        self.prepareCamera()
                    } else {
                        self.notAuthorized()
                    }
                })
            })
        }
    }
}

```

```

    } else {
        notAuthorized()
    }
}
}

```

The `authorizationStatusForMediaType()` method returns a `AVAuthorizationStatus` value to inform the current status of the authorization. If the value is `Authorized`, it means that we have been previously authorized and we can use the camera, but if the value is `NotDetermined`, we have to call the `requestAccessForMediaType()` method to ask for permission. This method includes a closure as the second parameter that is executed after the user responds to the request. The closure includes an attribute of type `Bool` to report the user's decision.

We use two methods to respond to each situation. The `prepareCamera()` method is executed every time the code detects that we are authorized to use the camera, and the `notAuthorized()` method is executed otherwise. In the following implementation of the latter, we show an Alert View to inform the situation to the user.

Listing 26-9: Informing the user that the camera is not accessible

```

func notAuthorized() {
    let alert = UIAlertController(title: "No Camera", message: "This app is not authorized to use the camera",
    preferredStyle: .Alert)

    let action = UIAlertAction(title: "OK", style: .Default, handler: nil)
    alert.addAction(action)

    presentViewController(alert, animated: true, completion: nil)
}

```

The `prepareCamera()` method is where we begin to build the network of objects introduced in Figure 26-4. This method gets a reference to the current capture device for video and creates the inputs and outputs we need to capture a still image (to take a picture).

Listing 26-10: Initializing the camera

```

func prepareCamera() {
    captureSession = AVCaptureSession()
    let device = AVCaptureDevice.defaultDeviceWithMediaType(AVMediaTypeVideo)
    if let input = (try? AVCaptureDeviceInput(device: device)) {
        captureSession.addInput(input)
        stillImage = AVCaptureStillImageOutput()
        captureSession.addOutput(stillImage)
        showCamera()
    }
}

```

```

} else {
    notAuthorized()
}
}

```

We can create and add all the inputs and outputs we need, in any possible order, but because the `AVCaptureDeviceInput()` initializer throws an error, we use it first. This initializer creates an object that manages the input for the capture device. If the initializer is successful, we add it to the capture session with the `addInput()` method and then create the output. For this example we have decided to use the session to capture a still image (to take a picture), so we use the `AVCaptureStillImageOutput` class to create the output and add it to the session with the `addOutput()` method.

IMPORTANT: The `AVCaptureStillImageOutput` class, along with the rest of the `AVCaptureOutput` subclasses, includes several properties and methods for configuration. The configuration by default for this class generates a JPG image, but there are other data formats available. For more information, visit our website and follow the links for this chapter.

After adding the inputs and outputs to the capture session, the `prepareCamera()` method executes an additional method called `showCamera()` to generate the preview layer and show the video generated by the camera on the screen. In this method we have to create the layer and set its size and orientation.

Listing 26-11: Showing the video from the camera on the screen

```

func showCamera() {
    view.setNeedsLayout()
    view.layoutIfNeeded()
    let width = cameraView.bounds.size.width
    let height = cameraView.bounds.size.height

    previewLayer = AVCaptureVideoPreviewLayer(session: captureSession)
    previewLayer.videoGravity = AVLayerVideoGravityResizeAspectFill
    previewLayer.frame = CGRect(x: 0, y: 0, width: width, height: height)

    let connection = previewLayer.connection
    let device = UIDevice.currentDevice()
    switch device.orientation {
        case .Portrait:
            connection.videoOrientation = .Portrait
    }
}

```



```

case .LandscapeRight:
connection.videoOrientation = .LandscapeLeft
case .LandscapeLeft:
connection.videoOrientation = .LandscapeRight
default:
connection.videoOrientation = .Portrait
}
let layer = cameraView.layer
layer.addSublayer(previewLayer)
captureSession.startRunning()
}

```

The `AVCaptureVideoPreviewLayer()` initializer creates a layer that we have to adjust to the size of its view and add as a sublayer of the current view's layer. But setting the position and size of the layer doesn't determine how its content is going to be shown. The video coming from the camera could have a different size and orientation. How the video is going to adjust to the size of the layer is determined by assigning a new value to the layer's `videoGravity` property, but the orientation is set from the connection established between the capture session and the preview layer. This is why after setting the value of the `videoGravity` property and the layer's frame, we get a reference to the connection from the layer's `connection` property. Modifying the `videoOrientation` property of the connection we can adjust the orientation according to the device's orientation and finally add the sublayer to the view's layer with the `addSublayer()` method. When the sublayer is ready, the capture session can be initiated with the `startRunning()` method.

At this point the video is playing on the screen and the system is ready to perform a capture. To let the user take a picture, we just have to connect the button added at the bottom of the view (Figure 26-5, right) to an Action in the `CameraViewController` class, but the way we process the output in this Action depends on what type of subclass we used to create it. For example, the `AVCaptureStillImageOutput` class used to capture still images includes the following property and methods to get and process the image.

connections—This property returns an array of `AVCaptureConnection` objects that represent the connections between the output and the inputs.

captureStillImageAsynchronouslyFromConnection(connection, completionHandler: block)—This method captures a still image from the capture device connected to the connection defined by the `connection` attribute. The `block` attribute is a closure with the statements we want to execute after the image is captured. This closure receives two values, a value of type `CMSampleBuffer` with the media's raw data and a value of type `NSError` to report errors.

jpegStillImageNSDataRepresentation(buffer)—This type method creates an

`NSData` object that represents an image from the raw data specified by the **buffer** attribute. The attribute is a value of type `CMSampleBuffer`.

The following code adds an Action called `takePicture()` to capture an image from the output created by the `prepareCamera()` method. The Action gets a reference to the connection created for the output and then captures a still image with the `captureStillImageAsynchronouslyFromConnection()` method. If the method is successful, we get the data from the value returned by the **buffer** attribute and create a `UIImage` object to contain the picture.

Listing 26-12: Taking a picture

```
@IBAction func takePicture(sender: UIButton) {
    let connection = stillImage.connections[0] as! AVCaptureConnection
    stillImage.captureStillImageAsynchronouslyFromConnection(connection, completionHandler: { (buffer:
CMSampleBufferRef!, error: NSError!) in
        if buffer != nil {
            let imageData = AVCaptureStillImageOutput.jpegStillImageNSDataRepresentation(buffer)
            let scale = UIScreen.mainScreen().scale
            self.picture = UIImage(data: imageData, scale: scale)
            self.performSegueWithIdentifier("goBackSegue", sender: self)
        }
    })
}
```

In the code of Listing 26-12, the picture captured from the camera is assigned to the **picture** property and then the Segue identified with the string “goBackSegue” is triggered to take the user back to the initial view. This is the Unwind Segue we have created for the **goBack()** Action included in the **ViewController** class of Listing 26-7 (see Figure 13-19 to learn how to create the Unwind Segue and Figure 13-16 to learn how to change its name).

The controller for the camera is ready, but there is one more method we need to add to the **CameraViewController** class to adapt the preview layer to the orientation of the device. As we have done before for other examples in this book, we can implement the **viewWillTransitionToSize()** method to get the new main view’s size and adjust the preview layer according to the orientation.

Listing 26-13: Rotating the preview layer

```
override func viewWillTransitionToSize(size: CGSize, withTransitionCoordinator coordinator:
UIViewViewControllerTransitionCoordinator) {
    super.viewWillTransitionToSize(size, withTransitionCoordinator: coordinator)
    previewLayer.frame.size = size
}
```

```

if let connection = previewLayer.connection {
    let device = UIDevice.currentDevice()
    switch device.orientation {
    case .Portrait:
        connection.videoOrientation = .Portrait
    case .LandscapeRight:
        connection.videoOrientation = .LandscapeLeft
    case .LandscapeLeft:
        connection.videoOrientation = .LandscapeRight
    default:
        connection.videoOrientation = .Portrait
    }
}
}
}

```

Do It Yourself: Update the **ViewController** class with the code of Listing 26-7. Create a subclass of **UIViewController** called **CameraViewController** and assign it to the second view. Connect the view for the preview layer with an Outlet called **cameraView**. Complete the class with the code of Listing 26-8. Add the methods of Listings 26-9, 26-10, and 26-11 to this class. Connect the button in the second view to an Action called **takePicture()** and complete the method with the code of Listing 26-12. Complete the **CameraViewController** class with the method of Listing 26-13 and run the application.

In the **prepareCamera()** method of Listing 26-10 we get a reference to the device set by default to capture video. This is usually the back camera, but we can select a specific one by listing all the devices available with the **devicesWithMediaType()** method.

Listing 26-14: Selecting the front camera

```

func prepareCamera() {
    captureSession = AVCaptureSession()
    var device: AVCaptureDevice!
    let devices = AVCaptureDevice.devicesWithMediaType(AVMediaTypeVideo)
    for item in devices {
        if item.position == .Front {
            device = item as! AVCaptureDevice
        }
    }
    if device != nil {
        if let input = (try? AVCaptureDeviceInput(device: device)) {
            captureSession.addInput(input)
        }
    }
}

```

```

stillImage = AVCaptureStillImageOutput()
captureSession.addOutput(stillImage)
showCamera()
} else {
notAuthorized()
}
} else {
notAuthorized()
}
}

```

The **AVCaptureDevice** class offers the **position** property to know the physical position of the device. This property returns a value of type **AVCaptureDevicePosition** that determines the position. The possible values are **Unspecified**, **Back**, and **Front**. In Listing 26-14 we get a reference to the device at the front and use that device to capture the image.

Do It Yourself: Replace the **prepareCamera()** method in your project by the method of Listing 26-14 and run the application. This time the preview layer should display the video from the front camera.

Photos

The same way we can create a custom controller for the cameras we can do it for the Photos Library. Apple provides a specific framework called Photos to access the photos and videos stored in the device. The most important class of this framework is called **PHPhotoLibrary**. This class can be used to create, delete, and modify items, and it also includes the following type methods to ask the user for permission to access the Photos library.

authorizationStatus()—This type method returns a value that determines the status of the authorization to access the Photos library. It returns an enumeration of type **PHAuthorizationStatus** with the values **NotDetermined**, **Restricted**, **Denied**, and **Authorized**.

requestAuthorization(block)—This type method requests the user access to the Photos library. The **block** attribute is a closure with the statements we want to execute after the request is completed.

The photos and videos in the Photos Library are represented by objects created from the

PHAsset class. The class includes several methods to retrieve assets. The following are the most frequently used.

fetchAssetsWithOptions(options)—This type method retrieves all the assets available in the Photos library. The **options** attribute is an object of type **PHFetchOptions** that includes the **predicate** and **sortDescriptors** properties to define a predicate and a sort descriptor to filter and order the results.

fetchAssetsWithMediaType(type, options: options)—This type method retrieves assets from the Photos library of the type specified by the **type** attribute. The **type** attribute is an enumeration of type **PHAssetMediaType** with the values **Unknown**, **Image**, **Video**, and **Audio**. The **options** attribute is an object of type **PHFetchOptions** that includes the **predicate** and **sortDescriptors** properties to define a predicate and a sort descriptor to filter and order the results.

The **PHAsset** class also includes properties that return the asset's attributes. The following are the most frequently used.

mediaType—This property returns a value that determines the type of the media. It is an enumeration of type **PHAssetMediaType** with the values **Unknown**, **Image**, **Video**, and **Audio**.

pixelWidth—This property returns an integer with the width of the photo or video.

pixelHeight—This property returns an integer with the height of the photo or video.

creationDate—This property returns an **NSDate** value with the media's date of creation.

modificationDate—This property returns an **NSDate** value with the date the last time the media was modified.

duration—This property returns a value of type **NSTimeInterval** (a typealias of **Double**) that determines the duration of the video.

favorite—This property returns a Boolean value that determines whether or not the asset is marked as favourite.

The **PHAsset** objects only include information related to the media but not the media itself. To get the photo or the video represented by the asset, we have to load it using a manager. The framework offers the **PHImageManager** to create and use a manager. The system automatically creates a manager and assigns it to the Photo library. The following are the methods provided by the class to get a reference to this default manager and load the assets.

defaultManager()—This method returns a reference to the `PHImageManager` object that represents the manager assigned by default.

requestImageForAsset(asset, targetSize: size, contentMode: mode, options: options, resultHandler: block)—This method requests the image of the `PHAsset` object specified by the `asset` attribute. The `size` attribute is a `CGSize` value with the size we want to give to the image (usually used to create thumbnails), the `mode` attribute is an enumeration of type `PHImageContentMode` with the values `AspectFit` and `AspectFill` that determines how the photo is going to adapt to the size of the image returned, the `options` attribute is an object of type `PHImageRequestOptions` used to specify image attributes (usually declared as `nil`), and the `block` attribute is a closure that is executed when the image is loaded. The closure receives two attributes with the image and a dictionary containing the status of the request.

requestImageDataForAsset(asset, options: options, resultHandler: block)—This method requests the image of the `PHAsset` object specified by the `asset` attribute in its full size. The `options` attribute is an optional object of type `PHImageRequestOptions` used to specify image attributes (usually declared as `nil`), and the `block` attribute is a closure that is executed when the image is loaded. The closure receives four attributes: an `NSData` object with the image data, a string with the image UTI (the image identifier), a `UIImageOrientation` value with the image's orientation, and a dictionary containing the status of the request.

requestPlayerItemForVideo(asset, options: options, resultHandler: block)—This method requests a representation of the video for playback. The `asset` attribute is a `PHAsset` object with the video we want to play, the `options` attribute is an object of type `PHVideoRequestOptions` used to specify video attributes (usually declared as `nil`), and the `block` attribute is a closure that is executed when the video is ready to be played. The closure receives two attributes: an `AVPlayerItem` object we can use to play the video and a dictionary containing the status of the request.

As an example, we are going to adapt the previous project to include a Collection View Controller to present photos from a Photos library instead of the video from the camera.

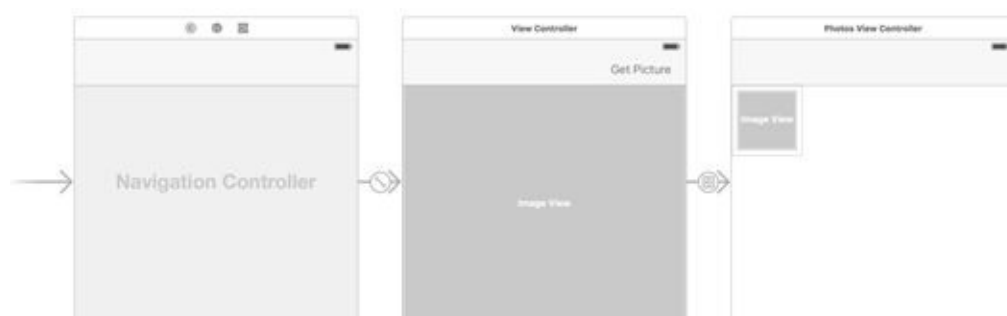


Figure 26-6: Custom controller for photos

The initial view and its view controller are the same, but the second view is now a Collection View Controller with a cell containing an Image View to present each photo.

Do It Yourself: Remove the second view in the previous project and delete the `CameraViewController.swift` file. Add a Collection View Controller to the Storyboard. Connect the Get Picture button to this view with a Show Segue. Create the Unwind Segue for the second view, connect it to the `goBack()` Action introduced in Listing 26-7, and give it the identifier “goBackSegue”. Create a file with a subclass of `UICollectionViewController` called `PhotosViewController` and assign it to the Collection View Controller. Add an Image View to the cell and give the cell a size of 100 x 100 points (see Figure 15-4). Create a subclass of `UICollectionViewCell` called `PhotosCell` and assign it to the cell. Connect the Image View in the cell to the `PhotosCell` class with an Outlet called `imagePhoto`. Give the cell the identifier “photosCell”. Replace the name of the class `CameraViewController` by `PhotosViewController` inside the `ViewController` class.

As always, we start by asking permission to the user to access the photos.

Listing 26-15: Asking permission to access the photos

```
import UIKit
import Photos

class PhotosViewController: UICollectionViewController {
    var photosList: PHFetchResult!
    var picture: UIImage!
    override func viewDidLoad() {
        super.viewDidLoad()
        let status = PHPhotoLibrary.authorizationStatus()
        if status == PHAuthorizationStatus.Authorized {
            loadPhotos()
        } else if status == PHAuthorizationStatus.NotDetermined {
            PHPhotoLibrary.requestAuthorization({ (status) in
                NSOperationQueue.mainQueue().addOperationWithBlock({
                    if status == PHAuthorizationStatus.Authorized {
                        self.loadPhotos()
                        self.collectionView?.reloadData()
                    } else {
                        self.notAuthorized()
                    }
                })
            })
        }
    }
}
```

```

    })
    })
    } else {
        notAuthorized()
    }
}

func notAuthorized() {
    let alert = UIAlertController(title: "No Photos", message: "This app is not authorized to access the photos",
preferredStyle: .Alert)

    let action = UIAlertAction(title: "OK", style: .Default, handler: nil)
    alert.addAction(action)

    presentViewController(alert, animated: true, completion: nil)
}
}

```

The procedure is very similar to the one we used before to get authorization for the camera, although this time we have to implement the methods provided by the Photos framework. If the user gives permission, we call the `loadPhotos()` method to get a list of the photos available. In this method we only need to get the list of asset objects available and assign it to a property we called `photosList`, so the Collection View methods have access to them.

Listing 26-16: Getting the list of photos available

```

func loadPhotos() {
    photosList = PHAsset.fetchAssetsWithMediaType(.Image, options: nil)
}

```

Now that we have the list of photos, we can use it to create the Collection View's cells. The following are the protocol methods we need to implement to show the pictures on the screen and let the user select one.

Listing 26-17: Adding the photos to the Collection View

```

override func collectionView(collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
    if photosList != nil {
        return photosList.count
    }

    return 0
}

override func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) ->
UICollectionViewCell {

```



```

    let cell = collectionView.dequeueReusableCellWithReuseIdentifier("photosCell", forIndexPath: indexPath) as!
    PhotosCell

    let asset = photosList[indexPath.item] as! PHAsset
    let manager = PHImageManager.defaultManager()
    manager.requestImageForAsset(asset, targetSize: CGSize(width: 100, height: 100), contentMode: .AspectFit,
options: nil, resultHandler: { (image, info) in
        cell.imagePhoto.image = image
    })
    return cell
}

override func collectionView(collectionView: UICollectionView, didSelectItemAtIndexPath indexPath: NSIndexPath) {
    let asset = photosList[indexPath.item] as! PHAsset
    let manager = PHImageManager.defaultManager()
    manager.requestImageDataForAsset(asset, options: nil, resultHandler: { (data, dataUTI, orientation, info) in
        if data != nil {
            let scale = UIScreen.mainScreen().scale
            self.picture = UIImage(data: data!, scale: scale)
            self.performSegueWithIdentifier("goBackSegue", sender: self)
        }
    })
}

```

As we mentioned before, to get the image associated with an asset object we have to load it using a manager. In Listing 26-17 we use this manager first to get a thumbnail of the image for each cell and then again to get the image in its original size when a cell is selected. The method we use to create the thumbnail is called `requestImageForAsset()`. This method produces a `UIImage` object with an image of the size specified by its **size** attribute that can be assigned to the cell's Image View. The second method is called `requestImageDataForAsset()`. This method produces an `NSData` object with the data for the entire image that can be used to create the `UIImage` object necessary for the initial view to present the photo to the user.

In this example we use the same process implemented before to provide the image to the initial view. The original image is assigned to the `picture` property and then the Unwind Segue is performed to move back to the initial view and show the photo on the screen.

Do It Yourself: Update the `PhotosViewController` class with the code of Listing 26-15. Add the methods of Listings 26-16 and 26-17 to the class. If you run the application on your device, you should see a list of your photos. Tap on a photo to select it.

The `fetchAssetsWithMediaType()` method used in Listing 26-16 to retrieve the list of photos available in the Photos library receives an attribute that we can use to filter and order the

results. This attribute is an object of a class called **PHFetchOptions** that includes the **predicate** and **sortDescriptors** properties to provide an **NSPredicate** object and an array of **NSSortDescriptor** objects to filter and order the results, as we did in the previous chapter for Core Data requests. The following example orders the photos by date (the names of the keys for the sort descriptor corresponds to the name of the properties defined in the **PHAsset** class).

Listing 26-18: Ordering the photos

```
func loadPhotos() {  
    let options = PHFetchOptions()  
    let sort = NSSortDescriptor(key: "creationDate", ascending: false)  
    options.sortDescriptors = [sort]  
    photosList = PHAsset.fetchAssetsWithMediaType(.Image, options: options)  
}
```

Do It Yourself: Replace the **loadPhotos()** method of the **PhotosViewController** class by the method of Listing 26-18. Run the application. The photos should be ordered according to the date they were created.

Videos

The same way we get the photos from the Photos Library we can get the videos. There is only one change we have to perform in the previous example to get a list of videos instead of photos. The **fetchAssetsWithMediaType()** type method used in the **loadPhotos()** method to fetch the assets in the Photos Library contains an attribute to specify the media. Changing this attribute to **Video** we get a list of the videos available on the device, as shown next.

Listing 26-19: Fetching videos

```
func loadPhotos() {  
    let options = PHFetchOptions()  
    let sort = NSSortDescriptor(key: "creationDate", ascending: false)  
    options.sortDescriptors = [sort]  
    photosList = PHAsset.fetchAssetsWithMediaType(.Video, options: options)  
}
```

This time, the asset selected by the user has to be processed with the manager's **requestPlayerItemForVideo()** method. This method gets the video from the asset and creates an **AVPlayerItem** object that we can use to play the video. In the following example we create

an **AVPlayer** object with the player item received by the closure and present it on the screen with an **AVPlayerViewController** controller (see Chapter 25).

Listing 26-20: Playing videos

```
override func collectionView(collectionView: UICollectionView, didSelectItemAtIndexPath indexPath: NSIndexPath) {
    let asset = photosList[indexPath.item] as! PHAsset
    let manager = PHImageManager.defaultManager()
    manager.requestPlayerItemForVideo(asset, options: nil, resultHandler: { (playerItem, info) in
        if let item = playerItem {
            NSOperationQueue.mainQueue().addOperationWithBlock({
                let player = AVPlayer(playerItem: item)
                let controller = AVPlayerViewController()
                controller.player = player
                self.presentViewController(controller, animated: true, completion: {
                    player.play()
                })
            })
        }
    })
}
```

Do It Yourself: Update the **loadPhotos()** method in the **PhotosViewController** class with the code of Listing 26-19 and the **collectionView(collectionView, didSelectItemAtIndexPath: indexPath)** method with the code of Listing 26-20. Import the AVKit framework to be able to work with the **AVPlayerViewController** class (**import AVKit**). Run the application and select a video to play it.

26.3 Core Image

Core Image is a small but very powerful framework to process images. The framework is used along with other frameworks to apply filters to images and edit pictures. Although its main function is to apply filters, it can also be used to detect faces, remove red eyes, change balance, tone curve, etc. The framework defines its own class to represent the images called `CImage`. A `CImage` object can be created from different sources, like `UIImage` objects, image files, and image data. The following are its most frequently used initializers.

`CImage(CGImage: image)`—This initializer creates a `CImage` object with the image provided by the **`image`** attribute. The attribute is an object of type `CGImage` with the image we want to process.

`CImage(image: image)`—This initializer creates a `CImage` object with the image provided by the **`image`** attribute. The attribute is an object of type `UIImage` with the image we want to process.

`CImage(contentsOfURL: URL)`—This initializer creates a `CImage` object with the image in the URL provided by the **`URL`** attribute. The attribute is an `NSURL` object with the location of the image file we want to process.

`CImage(data: data)`—This initializer creates a `CImage` object with the image data provided by the **`data`** attribute. The attribute is an `NSData` object with the data that represents the image we want to process.

To render the image, Core Image uses a context. The class is called `CImageContext` and provides the following initializer and method to create the context and get the image from it.

`CImageContext(options: options)`—This initializer creates a `CImageContext` object with the configuration specified by the **`options`** attribute. The attribute is a dictionary containing the options and values for configuration. The framework defines four constants to use as keys: `kCImageContextOutputColorSpace`, `kCImageContextWorkingColorSpace`, `kCImageContextUseSoftwareRenderer`, and `kCImageContextWorkingFormat`. The value `nil` sets the configuration by default.

`createCGImage(image, fromRect: frame)`—This method returns a `CGImage` object created from the image and size specified by the attributes. The **`image`** attribute is the `CImage` object with the image we want to transform and the **`frame`** attribute is a `CGRect` value with the size of the image.

As we mentioned, the main function of the framework is to apply filters to an image. This process is performed by an instance of the `CIFilter` class. The class includes the following initializer to create the filter.

`CIFilter(name: name)`—This initializer creates a `CIFilter` object containing the type of filter defined by the **`name`** attribute. The attribute is a string with the name of the filter we want to apply (e.g., `CIGaussianBlur`, `CISepiaTone`, `CIGlassDistortion`, `CIComicEffect`).

The characteristics of the filter are configured through parameters set and retrieved by the `setValue()` and `valueForKey()` methods. The framework defines keys for the parameters of every filter, including two parameters common to every filter that are used to set the input image and to retrieve the output image. To apply a filter to an image, we have to get a `CUIImage` object representing the image, create the filter, set its parameters, and get the image back from the context.

Listing 26-21: Applying a filter

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var pictureView: UIImageView!
    var picture: UIImage!

    override func viewDidLoad() {
        super.viewDidLoad()
        picture = UIImage(named: "husky")
        pictureView.image = picture
    }

    @IBAction func applyFilter(sender: UIButton) {
        let context = CIContext(options: nil)
        let image = CUIImage(image: picture)
        if let filter = CIFilter(name: "CIComicEffect") {
            filter.setValue(image, forKey: "inputImage")
            if let result = filter.valueForKey("outputImage") as? CUIImage {
                let frame = result.extent
                let resultImage = context.createCGImage(result, fromRect: frame)
                let newImage = UIImage(CGImage: resultImage)
                pictureView.image = newImage
            }
        }
    }
}
```

This example assumes that we have an interface with a button and an Image View and have included the husky.png image into the project. When the button is pressed, the `applyFilter()` method creates a `CIFilter` object with the configuration by default, a `CUIImage` object from the picture we want to modify, and finally a filter of type `CIComicEffect`. Every filter includes two basic parameters: the `inputImage` parameter to set the input image and the `outputImage` parameter to get the output image. In Listing 26-21 we assigned the `CUIImage` object we have just created as the filter's input and then obtained the filtered image with the `outputImage` key.

The `valueForKey()` method returns a `CUIImage` object, but we can retrieve the image as a `CGImage` object from the context by calling its `createCGImage()` method. The method requires a reference to the `CUIImage` object and also a `CGRect` value that determines its size. To get the size, we read a property provided by the `CUIImage` class called `extent`. This property returns a `CGRect` value with the image's size, which is exactly what we need. Finally, the `CGImage` object is used to create a `UIImage` object that we assign to the Image View to show the image on the screen.

In Listing 26-21 we first assign the original image to the Image View and then apply the filter when the user presses the button. The result is shown in Figure 26-7.



Figure 26-7: Image before and after a filter is applied

IMPORTANT: There is an extensive list of predefined filters available and each filter in turn defines its own unique parameters. For a complete list, visit our website and follow the links for this chapter.

The `CIComicEffect` filter doesn't have any parameters, but other filters require the definition of one or more parameters to work properly. For example, a filter called `CISepiaTone` that turns the colors in the image into shades of brown have a parameter called `inputIntensity` to define its intensity.

Listing 26-22: Defining filter’s parameters

```
@IBAction func applyFilter(sender: UIButton) {  
    let context = CIColorContext(options: nil)  
    let image = UIImage(image: picture)  
    if let filter = CIFilter(name: "CISepiaTone") {  
        filter.setValue(image, forKey: "inputImage")  
        filter.setValue(0.8, forKey: "inputIntensity")  
  
        if let result = filter.valueForKey("outputImage") as? UIImage {  
            let frame = result.extent  
            let resultImage = context.createCGImage(result, fromRect: frame)  
            let newImage = UIImage(CGImage: resultImage)  
            pictureView.image = newImage  
        }  
    }  
}
```

Do It Yourself: Create a new Single View Application project. Download the husky.png image from our website and add it to the project. Add a button called Apply Filter and an Image View to the initial view. Connect the Image View to the **ViewController** class with an Outlet called **pictureView** and the button with an Action called **applyFilter()**. Complete the **ViewController** class with the code of Listing 26-21. Run the application and press the Apply Filter button. You should see the image changing as illustrated by Figure 26-7. Replace the **applyFilter()** method by the method of Listing 26-22 and run the application again.

IMPORTANT: Core Image is a simple framework but provides multiple alternatives to process images and videos. Besides the **CIFilter** class, the framework includes very powerful classes like **CIDetector** to identify features such as faces or barcodes, or the **CIFilterGenerator** class to create our own filters. The topic is beyond the scope of this book. For more information visit our website and follow the links for this chapter.

Chapter 27

Web

27.1 Links

The most important aspect of the web is the capacity to access documents through what is commonly referred to as *links*. A link is a URL pointing to the location of a document. When the user selects the link, the document is opened. Links were designed for the web, but we can add them to our applications and let the system decide where to open the document. If the URL associated with the link is a web address, the system opens the default browser and asks it to load the document.

Web addresses are created from `NSURL` objects. We have used these types of objects before to determine the location of files, but we can also use them to access remote documents. The class includes the following initializer to create a URL.

NSURL(string: address)—This initializer creates an `NSURL` object with the URL specified by the **address** attribute. The attribute is a string with the address we want to access.

NSURL(string: address, relativeToURL: base)—This initializer creates an `NSURL` object with the URL specified by the attributes. The URL is created by adding the value of the **address** attribute to the value of the **base** attribute. For example, if the **base** attribute is the string “http://www.formasterminds.com” and the **address** attribute is the string “index.php”, the `NSURL` object will contain the URL `http://www.formasterminds.com/index.php`.

NSURL(dataRepresentation: data, relativeToURL: base)—This initializer creates an `NSURL` object with the URL specified by the attributes. The URL is created by adding the value of the **data** attribute to the value of the **base** attribute. The data attribute is an `NSData` object that represents a string of characters.

The `NSURL` object is just a container for the URL of the document we want to open, but the document is opened from methods of the `UIApplication` class. We have to access the `UIApplication` object created for our app and use the following methods to process the URL.

openURL(URL)—This method opens the URL specified by the **URL** attribute. The method returns a Boolean value that determines whether the document was opened or not.

canOpenURL(URL)—This method analyzes the URL specified by the **URL** attribute and returns a Boolean value that determines if the URL can be opened by an application installed on the device.

The following example opens the website www.formasterminds.com when a button in the initial view is pressed. The code creates a `String` constant to contain the string with the URL, creates the `NSURL` object from this value, and finally calls the `openURL()` method to open the URL. The system reads the URL, detects that it is a web address and opens the browser to load the website.

Listing 27-1: Opening a website

```
import UIKit

class ViewController: UIViewController {
    @IBAction func openWeb(sender: UIButton) {
        let web = "http://www.formasterminds.com"
        if let URL = NSURL(string: web) {
            let app = UIApplication.sharedApplication()
            app.openURL(URL)
        }
    }
}
```

Do It Yourself: Create a Single View Application project. Add a button to the initial view and connect it to the `ViewController` class with an Action called `openWeb()`. Complete the `ViewController` class with the code of Listing 27-1. Run the application and press the button. The system should open the browser and load the website.

In this example we have defined the URL in code, but sometimes the URL is provided by the user or taken from another document. In cases like this, the URL may contain characters that are not allowed and can cause the location to be impossible to identify. To make sure that the URL is valid, we have to turn unsafe characters into percent-encoding characters. These are characters represented by the % sign plus a hexadecimal number. Fortunately, there are methods that can easily correct unsafe characters in a string. We have studied a method like this in Chapter 21. The `String` structure offers the `dataUsingEncoding()` method to turn a string into an `NSData` object using a type of encoding that corrects the characters for us. The following example shows how to implement this method.

Listing 27-2: Encoding URLs

```
import UIKit

class ViewController: UIViewController {
    @IBAction func openWeb(sender: UIButton) {
```

```
let web = "http://www.formasterminds.com"
let dataURL = web.dataUsingEncoding(NSUTF8StringEncoding)
let URL = NSURL(dataRepresentation: dataURL!, relativeToURL: nil)

let app = UIApplication.sharedApplication()
app.openURL(URL)
}
}
```

The **NSUTF8StringEncoding** format used in Listing 27-2 works with ASCII characters and therefore it is suitable for the creation of URLs. The method returns an **NSData** object containing the URL, so we have to use the appropriate **NSURL** initializer for this type of value. Once the **NSURL** object is created, the process to open the URL is the same.

27.2 Web Views

Links offer a window to the web from our app, but they open the document in an external application. UIKit includes a subclass of `UIView` called `UIWebView` that implements a web browser. Incorporating this view into our interface we can load a website and let the user navigate without leaving our app. The option to add a Web View to the Storyboard is available in the Object Library.

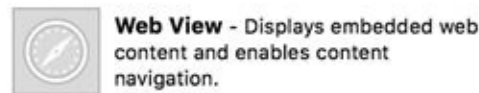


Figure 27-1: Web View option in the Object Library

A Web View can load web content and an entire website. The `UIWebView` class provides the following properties and methods to load and manage the content.

scalesPageToFit—This property sets or returns a Boolean value that determines whether the content is scaled to fit the size of the Web View or not.

loading—This property returns a Boolean value that determines if the content is loading.

canGoBack—This property returns a Boolean value that determines if the Web View can navigate to the previous page.

canGoForward—This property returns a Boolean value that determines if the Web View can navigate to the next page.

dataDetectorTypes—This property sets or returns a value that determines the data in the content that is going to be turned automatically into clickable links. It is an enumeration of type `UIDataDetectorTypes` with the values `PhoneNumber`, `Link`, `Address`, `CalendarEvent`, `None`, and `All`.

loadRequest(request)—This method loads the content of a URL. The `request` attribute is an `NSURLRequest` object with the request for the URL we want to open.

stopLoading()—This method stops the loading of the content.

reload()—This method reloads the content (it refreshes the web page).

goBack()—This method navigates to the previous web page.

goForward()—This method navigates to the next web page.

URLs are addresses that indicate the location of a file, but we also need to establish the kind of protocol we are going to use to make the connection with that location. The UIKit framework offers the `NSURLRequest` class to manage this information. The class includes the following initializer.

NSURLRequest(URL: URL)—This initializer creates an `NSURLRequest` object for the URL and protocol specified by the `URL` attribute. The attribute is an `NSURL` object (the information for the protocol is taken from the URL itself).

A Web View can report the state of the content through a delegate. For this purpose, UIKit defines the `UIWebViewDelegate` protocol with the following methods.

webView(webView, shouldStartLoadWithRequest: request, navigationType: navigationType)—This method is called on the delegate to determine if the Web View should process a request. The method must return a Boolean value to indicate our decision. The `request` attribute is a reference to the request, and the `navigationType` attribute is an enumeration of type `UIWebViewNavigationType` that indicates the action the user performed to generate the request. The possible values are `LinkClicked`, `FormSubmitted`, `BackForward`, `Reload`, `FormResubmitted`, and `Other`.

webViewDidStartLoad(webView)—This method is called on the delegate when the Web View begins loading new content.

webViewDidFinishLoad(webView)—This method is called on the delegate when the Web View finishes loading the content.

webView(webView, didFailLoadWithError: error)—This method is called on the delegate when an error occurs loading the content.

The process to load a website in a Web View is simple. We have to get the URL, create a request, and order the view to load it.

Listing 27-3: Loading a website in a Web View

```
import UIKit
```

```
class ViewController: UIViewController {  
    @IBOutlet weak var web: UIWebView!  
    override func viewDidLoad() {
```

```
super.viewDidLoad()
```

```
web.scalesPageToFit = true
```

```
if let url = NSURL(string: "https://www.google.com") {
```

```
let request = NSURLRequest(URL: url)
```

```
web.loadRequest(request)
```

```
}
```

```
}
```

```
}
```

The example of Listing 27-3 assumes that we have added a Web View to the interface and it is connected to the **ViewController** class with an Outlet called **web**. The code begins by setting the **scalesPageToFit** property of the Web View to **true**. This is important because most websites are bigger than the size of the screen or the size we decided to give to the Web View. To prepare the request, we get the **NSURL** object with the address we want to access and initialize the **NSURLRequest** object. Finally, the request is loaded with the **loadRequest()** method and the website is shown on the screen.

Do It Yourself: Create a new Single View Application project. Add a Web View to the initial view and pin it to the edges of the view. Connect the Web View to the **ViewController** class with an Outlet called **web**. Complete the **ViewController** class with the code of Listing 27-3. Run the application. You should see Google’s website on the screen.

With this process we can load any website we want, including those specified or selected by the user (we just have to remember to prepare the URL the way we did before for links in Listing 27-2). But users need more control over the content. For instance, if we use the previous example to perform a search in Google, we will notice right away that there is no way to go back to the previous page or start again from the beginning. Web Views offer several tools to manipulate their content, including methods to load a different web page and a delegate we can define to respond to changes. The interface introduced in Figure 27-2 below includes a Navigation Bar with three buttons to go back, forward, and refresh the page.



Figure 27-2: Custom web browser for our application

Before modifying the content of a Web View we need to know what we can do and what we can't. For example, we can only go backwards if the user has already navigated forward. This is when the delegate becomes useful. Implementing the `webViewDidFinishLoad()` method we can check the status of the content every time a new document is loaded. The following is the `ViewController` class for the initial view of the interface of Figure 27-2. When the view is loaded, the buttons are disabled because there is only one document loaded in the Web View at the moment, but the state is changed every time a new document is loaded.

Listing 27-4: Implementing a custom web browser

```
import UIKit

class ViewController: UIViewController, UIWebViewDelegate {
    @IBOutlet weak var backButton: UIBarButtonItem!
    @IBOutlet weak var forwardButton: UIBarButtonItem!
    @IBOutlet weak var refreshButton: UIBarButtonItem!
    @IBOutlet weak var web: UIWebView!

    override func viewDidLoad() {
        super.viewDidLoad()
        backButton.enabled = false
        forwardButton.enabled = false

        web.scalesPageToFit = true
        web.delegate = self
        if let url = NSURL(string: "https://www.google.com") {
            let request = NSURLRequest(URL: url)
            web.loadRequest(request)
        }
    }

    @IBAction func moveBack(sender: AnyObject) {
        web.goBack()
    }

    @IBAction func moveForward(sender: AnyObject) {
        web.goForward()
    }

    @IBAction func refresh(sender: AnyObject) {
        web.reload()
    }

    func webViewDidFinishLoad(webView: UIWebView) {
```



```
        backButton.enabled = web.canGoBack
        forwardButton.enabled = web.canGoForward
    }
}
```

The three buttons are connected to their respective Actions to execute the Web View methods `goBack()`, `goForward()`, and `reload()`. After a new document is loaded, the `webViewDidFinishLoad()` method is executed and the state of the buttons is modified from the values of the Web View's properties `canGoBack` and `canGoForward`, activating the buttons only when there are documents to go to.

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add a Web View to the initial view. Add three Bar Buttons to the Navigation Bar (you can change the appearance of the buttons from the System Item option in the Attributes Inspector panel). Connect the buttons to Outlets called `backButton`, `forwardButton`, and `refreshButton`, and to Actions called `moveBack()`, `moveForward()`, and `refresh()`. Complete the `ViewController` class with the code of Listing 27-4. Run the application and search for a word. Click on a link in the results. The buttons in the Navigation Bar should activate or deactivate according to your location in the navigation history.

App Transport Security

In the previous examples we open secure URLs (URLs that begin with the prefix `https://`), because these are the URLs allowed by default. iOS implements a system called App Transport Security (ATS) to block insecure URLs, like those starting with the prefix `http://` (without the `s`). If we need to load insecure URLs, such as `http://www.formasterminds.com`, we can configure our app to circumvent this security measure for every website or specific domains.

Every project offers a file called `info.plist` to configure some aspects of the application. This is a Property List file and therefore it orders the information in a hierarchy. The values are identified with keys and they may contain other values inside, like dictionaries inside dictionaries. When we select this file from the Navigation Area, Xcode shows an editor in the Editor Area with the current values.

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(3 items)
► Supported interface orientations (L...	Array	(4 items)

Figure 27-3: Content of the *info.plist* file

If we move the mouse over an item, the editor shows a + button to add another item to the item’s hierarchy. For example, if we click on the + button of the root item called “Information Property List” a + button is shown to add a new item to the list (circled in Figure 27-4).

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)

Figure 27-4: Option to add new items to the list

When we click the + button, a new item is created. The option to configure the App Transport Security system is called “App Transport Security Settings”. After we select this option, we will see something similar to Figure 27-5.

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
► App Transport Security Settings	Dictionary	(0 items)
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)

Figure 27-5: Option to configure App Transport Security

This option also has a + button to add values to the item. In this case, the button has two functions: if the arrow on the left is pointing to the item (closed), a new item is added to the list, but if the arrow is pointing down (expanded), the new item is added to the item as a new value (we can click the arrow to close or expand the item). Values added to an item are shown below the item with a little indentation to reflect the hierarchy. The value we need to add to the App Transport Security Settings item to allow insecure URLs to be opened from our app is called “Allow Arbitrary Loads”. Figure 27-6 shows what the list

looks like after we add this value.

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
▼ App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES
Localization native development re...	String	en

Figure 27-6: App Transport Security configured to allow insecure URLs

The Allow Arbitrary Loads key represents a Boolean value specified with the strings YES and NO. Setting this key to YES allows any URL to be opened. If what we want is to allow only specific domains, we have to use the Exception Domains key and add to the key additional items with the domains we want to allow. These items in turn require at least three more items with the keys `NSIncludesSubdomains` (Boolean), `NSTemporaryExceptionAllowsInsecureHTTPLoads` (Boolean), and `NSTemporaryExceptionMinimumTLSVersion` (String). For example, the following configuration allows documents from the formasterminds.com domain to be opened.

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
▼ App Transport Security Settings	Dictionary	(1 item)
▼ Exception Domains	Dictionary	(1 item)
▼ formasterminds.com	Dictionary	(3 items)
NSIncludesSubdomains	Boolean	YES
NSTemporaryExceptionAllowsInsecureHTTPLoads	Boolean	YES
NSTemporaryExceptionMinimumTLSVersion	String	TLSv1.1
Localization native development region	String	en

Figure 27-7: App Transport Security configured to allow URLs from formasterminds.com

27.3 Web Content

Links and Web Views allow us to load a remote document and show it to the user, but they don't let us process the content. Sometimes all we need is to extract a piece of information from the document or process its data instead of showing the entire content as it is. In cases like this we have to load the document in the background and parse it to extract only what we need. Foundation includes a group of classes that conform a loading system that allows our app to get content referenced by a URL. The main class is called `NSURLSession`. This class creates a session that manages an HTTP connection to obtain data, download, or upload files. The following are the initializers and type method provided by the class to create the session.

`NSURLSession(configuration: configuration)`—This initializer creates a new session with the configuration set by its attribute. The **`configuration`** attribute is an object of the `NSURLSessionConfiguration` class that specifies the behaviour for the session.

`NSURLSession(configuration: configuration, delegate: delegate, delegateQueue: queue)`—This initializer creates a new session with the configuration set by its attributes. The **`configuration`** attribute is an object of the `NSURLSessionConfiguration` class that specifies the behaviour for the session, the **`delegate`** attribute is a reference to the delegate object we want to assign to the session, and the **`queue`** attribute is the queue in which the delegate methods and completion handlers are going to be called.

`sharedSession()`—This type method returns a standard session with a configuration by default that is suitable to perform basic requests.

The session sets up the connection but it doesn't perform any task. To download or upload data we have to create an `NSURLSessionTask` object with a specific task and add it to the session. The `NSURLSession` class includes the following methods to create tasks to download data, and download and upload files.

`dataTaskWithURL(URL, completionHandler: block)`—This method creates a task to download data and adds it to the session. The **`URL`** attribute is an `NSURL` object with the location of the data and the **`block`** attribute is a closure that is executed when the task is completed. The closure receives three optional attributes: an `NSData` object with the data returned by the request, an `NSHTTPURLResponse` object with the status of the request, and an `NSError` value to indicate if an error occurred.

`dataTaskWithRequest(request, completionHandler: block)`—This method creates a task to download data and adds it to the session. The **`request`** attribute is an

`NSURLRequest` object with the URL we want to access and the **block** attribute is a closure that is executed when the task is completed. The closure receives three optional attributes: an `NSData` object with the data returned by the request, an `NSHTTPURLResponse` object with the status of the request, and an `NSError` value to indicate if an error occurred.

downloadTaskWithURL(URL, completionHandler: block)—This method creates a task to download a file and adds it to the session. The **URL** attribute is an `NSURL` object with the location of the file and the **block** attribute is a closure that is executed when the task is completed. The closure receives three optional attributes: an `NSURL` object with the location of a temporary file that represents the file downloaded, an `NSHTTPURLResponse` object with the status of the request, and an `NSError` value to indicate if an error occurred.

downloadTaskWithRequest(request, completionHandler: block)—This method creates a task to download a file and adds it to the session. The **request** attribute is an `NSURLRequest` object with the URL of the file we want to access and the **block** attribute is a closure that is executed when the task is completed. The closure receives three optional attributes: an `NSURL` object with the location of a temporary file that represents the file downloaded, an `NSHTTPURLResponse` object with the status of the request, and an `NSError` value to indicate if an error occurred.

uploadTaskWithRequest(request, fromData: data, completionHandler: block)—This method creates a task to upload a file and adds it to the session. The **request** attribute is an `NSURLRequest` object with the URL where we want to upload the file, the **data** attribute is an `NSData` object with the data to create the file, and the **block** attribute is a closure that is executed when the task is completed. The closure receives three optional attributes: an `NSData` object with the data returned by the server, an `NSHTTPURLResponse` object with the status of the request, and an `NSError` value to indicate if an error occurred.

uploadTaskWithRequest(request, fromFile: URL, completionHandler: block)—This method creates a task to upload a file and adds it to the session. The **request** attribute is an `NSURLRequest` object with the URL where we want to upload the file, the **URL** attribute is an `NSURL` object with the location of the file we want to upload, and the **block** attribute is a closure that is executed when the task is completed. The closure receives three optional attributes: an `NSData` object with the data returned by the server, an `NSHTTPURLResponse` object with the status of the request, and an `NSError` value to indicate if an error occurred.

The objects returned by these methods are subclasses of the `NSURLSessionTask` class (`NSURLSessionDataTask`, `NSURLSessionDownloadTask`, and `NSURLSessionUploadTask`). All the subclasses include the following methods to control the task.

cancel()—This method cancels the task.

resume()—This method starts or resumes the task.

suspend()—This method suspends the task until it is resumed.

When the task is finished, it sends the results to a closure for processing. For example, if we use the `dataTaskWithRequest()` method to get data from a website, the closure for this task receives a value with the data and an object of type `NSHTTPURLResponse` to indicate the status of the request. The status is determined by an integer assigned according to web standards. There are several values available to indicate things like the success of the request (200) or more drastic situations like when the website has been moved to a different address (301). The `NSHTTPURLResponse` includes a property called `statusCode` to return this value. To be sure that the data was downloaded correctly, we have to check if the value of this property is equal to 200 before processing anything. The following example shows how to perform a basic request.

Listing 27-5: Loading a remote document

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        let url = NSURL(string: "https://www.yahoo.com")
        let request = NSURLRequest(URL: url!)
        let session = NSURLSession.sharedSession()

        let task = session.dataTaskWithRequest(request, completionHandler: { (data, response, error) in
            if error == nil && data != nil {
                if let resp = response as? NSHTTPURLResponse {
                    let status = resp.statusCode
                    if status == 200 {
                        let content = String(data: data!, encoding: NSASCIIStringEncoding)
                        print(content)
                    } else {
                        print("Error")
                    }
                }
            } else {
                print("Error")
            }
        })
    }
}
```

```

    task.resume()
}
}

```

The example of Listing 27-5 loads the content of the website at www.yahoo.com and prints it in the console. Because we are just loading a single web page, the standard session returned by the `sharedSession()` method is more than enough. We add to this session a task to download data and then call the `resume()` method at the end to perform the task. When the task is finished, we turn the data into a string with the `String()` initializer and then print the string on the console.

A standard session like the one we used in this example comes with a configuration by default that is suitable for most situations, but a custom session requires its own configuration. To configure a session, the loading system provides a class called `NSURLSessionConfiguration` with type methods and properties to set a specific configuration. The following is a type method we can use to get an object with values by default that we can adapt later to the requirements of our application.

defaultSessionConfiguration()—This type method returns an `NSURLSessionConfiguration` object with default settings.

Working with custom sessions only requires us to change how the session is initialized, but the rest of the code remains the same.

Listing 27-6: Instantiating a custom session

```

import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        let url = NSURL(string: "https://www.yahoo.com")
        let request = NSURLRequest(URL: url!)
        let config = NSURLSessionConfiguration.defaultSessionConfiguration()
        let session = NSURLSession(configuration: config)
        let task = session.dataTaskWithRequest(request, completionHandler: { (data, response, error) in
            if error == nil && data != nil {
                if let resp = response as? NSHTTPURLResponse {
                    let status = resp.statusCode
                    if status == 200 {
                        let content = String(data: data!, encoding: NSASCIIStringEncoding)
                        print(content)
                    }
                }
            }
        })
    }
}

```

```

    } else {
    print("Error")
    }
}
} else {
print("Error")
}
})
task.resume()
}
}

```

Do It Yourself: Create a new Single View Application project. Update the **ViewController** class with the codes of Listings 27-5 or 27-6. Run the application. You should see the HTML code of the Yahoo website printed in the console.

Custom sessions are useful when we need to respond to particular circumstances like the website being redirected or a request for authentication. These situations are managed by delegates of the session. The loading system defines a total of four protocols with methods specific for every type of task. The **NSURLSessionDelegate** protocol defines methods to handle session events, the **NSURLSessionTaskDelegate** protocol defines methods to handle events that are common to every task, the **NSURLSessionDataDelegate** protocol defines methods to handle data and upload tasks events, and the **NSURLSessionDownloadDelegate** protocol defines methods to handle download tasks events. Usually, a session requires only a few methods to respond to common situations. For example, the **NSURLSessionDelegate** protocol includes the following method to provide authentication for a secure connection.

URLSession(session, didReceiveChallenge: challenge, completionHandler: completionHandler)—This method is called on the delegate when authentication is requested by the server. Our implementation must call the completion handler received by the method with two attributes that define the settings and credentials.

There is also a useful method defined in the **NSURLSessionTaskDelegate** protocol to respond to a redirection.

URLSession(session, task: task, willPerformHTTPRedirection: response, newRequest: request, completionHandler: completionHandler)—This method is called on the delegate when the server redirected the connection to another URL. Our implementation must call the completion handler received by the method

with an attribute that defines the new request (the value of the **request** attribute) or the value **nil** if we don't want to follow the redirection.

Some websites, like www.google.com, send the user to a different address that contains a version of the website customized according to the user's location or preferences. This means that the URL we provide is not the final destination; the server does not return any data but instead redirects the user to another document. We can define a custom session with a delegate and then implement the method of the `NSURLSessionTaskDelegate` protocol to indicate what we want to do when the server is redirecting.

Listing 27-7: Following a redirection

```
import UIKit

class ViewController: UIViewController, NSURLSessionTaskDelegate {
    override func viewDidLoad() {
        super.viewDidLoad()

        let url = NSURL(string: "https://www.google.com")
        let request = NSURLRequest(URL: url!)
        let configuration = NSURLSessionConfiguration.defaultSessionConfiguration()
        let session = NSURLSession(configuration: configuration, delegate: self, delegateQueue: nil)

        let task = session.dataTaskWithRequest(request, completionHandler: { (data, response, error) in
            if error == nil && data != nil {
                if let resp = response as? NSHTTPURLResponse {
                    let status = resp.statusCode
                    if status == 200 {
                        let content = String(data: data!, encoding: NSASCIIStringEncoding)
                        print(content)
                    } else {
                        print("Error")
                    }
                }
            } else {
                print("Error")
            }
        })
        task.resume()
    }

    func URLSession(session: NSURLSession, task: NSURLSessionTask, willPerformHTTPRedirection response: NSHTTPURLResponse, newRequest request: NSURLRequest, completionHandler: (NSURLRequest?) -> Void) {
        completionHandler(request)
    }
}
```

```
}  
}
```

In the example of Listing 27-7 we define the **ViewController** class as the delegate of our session and call the closure received by the protocol method with the value of the new request. Every time the server asks for a redirection, the protocol method is called and the request with the new URL is executed. The process repeats until the final destination is reached and then the content of that document is again printed in the console.

JSON

Normal websites return documents written in HTML code. This is the basic language for every website. It is composed by predefined tags that are in charge of delineating the document's data. For example, the tags `<title>` and `</title>` enclose the title of the document, so we can use string methods to find the location of the last character of the `<title>` tag and the first character of the `</title>` tag and then cut the text in the middle, extracting the title. The problem with this approach is that most tags in HTML have the same name, making it very difficult to identify the information we want to retrieve. Also, websites don't follow a predefined structure and can change their code without a warning. If we create code that extracts information from the HTML code of a web page, it may stop working before we even ship the application to the App Store. This is not only a problem for mobile applications but also for web applications in general. The solution was found a long time ago with the creation of new formatting languages that have the exclusive purpose of transmitting data through the web. Some are programming languages and others are just format specifications, like JSON.

JSON (Javascript Object Notation) proposes a dictionary-like organization for the data. Every piece of data is stored with a key/value pair and related values are enclosed in braces. The advantage of this format is that the information is easy to find; we just have to follow the keys as in a dictionary, and, except for a few braces and commas, the documents only contain relevant data.

Because of its format, JSON files can easily be converted to dictionaries that we can process from our code. Foundation includes the **NSJSONSerialization** class to help us create a dictionary from a JSON file or a JSON structure from an object. The class includes two type methods for this purpose.

JSONObjectWithData(data, options: options)—This type method returns an **AnyObject** object that represents a collection of values generated from JSON data (array or dictionary). The **data** attribute is a reference to the object containing the JSON data, and the **options** attribute is a constant from the **NSJSONReadingOptions** structure that

determines how the data is going to be processed (usually declare as an empty set to specify default behaviour). The constants available are `MutableContainers` (mutable dictionaries and arrays), `MutableLeaves` (mutable strings), and `AllowFragments` (allows structures that are not dictionaries and arrays to be parsed).

`dataWithJSONObject(object, options: options)`—This type method returns JSON data generated from a Swift object. The **`object`** attribute is the object we want to convert, and the **`options`** attribute is a constant of the `NSJSONWritingOptions` structure. The only constant available at this moment is `PrettyPrinted`, which specifies that whitespaces should be added to the JSON data to make it more readable.

Websites are not automatically translated into JSON files. Developers decide what information to share and how to share it. For example, Yahoo doesn't offer a JSON document with the data in its home page, but it does offer a JSON file with information about the weather. These services are specific of each website and are usually included as an additional service with its own website and domain. For instance, the API provided by Yahoo to generate JSON documents with information about the weather is available at `developer.yahoo.com/weather/`. From these special websites we can get the URLs to access the JSON documents and download the data we need for our app. The following example gets the JSON document from the Yahoo service containing weather information about the city of San Diego.

Listing 27-8: Loading a JSON document

```
import UIKit
```

```
class ViewController: UIViewController {
```

```
    override func viewDidLoad() {  
        super.viewDidLoad()
```

```
        let url = NSURL(string: "https://query.yahooapis.com/v1/public/yql?  
q=select%20item.condition%20from%20weather.forecast%20where%20woeid%20%3D%202487889&format=js
```

```
        let request = NSURLRequest(URL: url!)  
        let session = NSURLSession.sharedSession()
```

```
        let task = session.dataTaskWithRequest(request, completionHandler: { (data, response, error) in
```

```
            if error == nil && data != nil {  
                if let resp = response as? NSHTTPURLResponse {  
                    let status = resp.statusCode  
                    if status == 200 {
```

```
                        self.getValues(data!)  
                    } else {  
                        print("Error")  
                    }  
                }  
            }  
        })
```

```

    }
    } else {
    print("Error")
    }
    })
    task.resume()
  }
}

```

The code of Listing 27-8 loads the JSON document and calls the `getValues()` method to process the data. The code to load the document is the same we use for any other type of document, the difference is in how we process the data received. The following is a possible implementation of our `getValues()` method.

Listing 27-9: Parsing JSON data

```

func getValues(data: NSData) {
  let info = try? NSJSONSerialization.JSONObjectWithData(data, options: [])
  if let listValues = info {
    if let query = listValues["query"] {
      if let results = query?["results"] {
        if let channel = results?["channel"] {
          if let item = channel?["item"] {
            if let condition = item?["condition"] {
              if let temperature = condition?["temp"] {
                print(temperature)
              }
            }
          }
        }
      }
    }
  }
}

```

As we mentioned, the data in a JSON document is organized as key/value pairs. To group related data, a key/value pair is assigned as the value of another key/value pair, creating a nested structure with values inside values. For instance, the weather API provided by Yahoo includes different levels of hierarchy to contain the information, with a level identified with the `condition` key that includes all the values related to the weather. Because each dictionary item is returned as a conditional, we have to unwrap the values

one by one in a chain of **if** statements until we reach the key we are looking for (in this case we look for the **condition** key and read the **temp** key to get the temperature).

IMPORTANT: JSON documents are usually structured as an array of dictionaries. The first level element is an array that contains dictionaries that in turn contain values or other arrays and dictionaries. To represent this collection of values, the `JSONObjectWithData()` method returns an `AnyObject` object. If you want to access the elements individually, as in the example of Listing 27-9, you can read the values directly from the `AnyObject` object, but if you need the system to treat the object as a collection, you have to cast it (e.g., `[AnyObject]`). We will see an example in Listing 27-18.

If we have multiple values to read in different locations, we have to concatenate **if** statements over and over again to find them. The next example shows a better way to do it. We create an array with all the keys and read their values one by one in a **for in** loop. The final value in this example corresponds to the **condition** key, and therefore we are able to read the **temp** key at the end to get the temperature.

Listing 27-10: Parsing JSON data with a loop

```
func getValues(data: NSData) {
    let info = try? NSJSONSerialization.JSONObjectWithData(data, options: [])
    if let listValues = info {
        let listKeys = ["query", "results", "channel", "item", "condition"]
        var current: AnyObject?
        current = listValues
        for key in listKeys {
            current = current?[key]
        }
        if let values = current {
            print(values["temp"])
        }
    }
}
```

Do It Yourself: Create a new Single View Application project. Update the `ViewController` class with the code of Listing 27-8. Add the method of Listing 27-10 to the class and run the application. You should see the value of the temperature in San Diego printed in the console. You can get the link to the JSON document from Yahoo’s website. Go to **developer.yahoo.com/weather/**, select the example of San Diego from the list, and select the option JSON from the responses. The web page

shows the JSON code generated for the document and the link to the document below. You can select other examples and adapt the code to extract the information according to the keys available in each document.

The Basics: The keys used in a JSON document are selected by the developer. Every developer decides what keys to use and how to structure the information. To know the keys used to construct the hierarchy, you have to load the JSON document in your browser or find examples provided on the website and study its structure.

XML

XML is similar to JSON, but instead of key/value pairs it uses tags, like HTML. The difference between XML and HTML is that the tags use custom names, so every tag may represent a specific value. Foundation provides the `NSXMLParser` class to create an XML parser. The class includes the following initializers.

`NSXMLParser(contentsOfURL: URL)`—This initializer creates an XML parser to parse the XML document at the location specified by the **`URL`** attribute.

`NSXMLParser(data: data)`—This initializer creates an XML parser to parse the XML document provided by the **`data`** attribute. The attribute is an `NSData` object with the data returned by a URL Request.

The initializers load the XML document and create the `NSXMLParser` object, but the parsing is controlled by the object's methods.

`parse()`—This method starts the parsing of the document.

`abortParsing()`—This method interrupts the parsing of the document.

The way an XML parser works is different than a JSON parser. The XML document is read line by line and every time a new element or content is found the parser calls delegate methods to process the values. Foundation includes the `NSXMLParserDelegate` protocol to define these methods.

`parserDidStartDocument(parser)`—This method is called on the delegate when the parser begins parsing the document.

parserDidEndDocument(parser)—This method is called on the delegate when the parser ends parsing the document.

parser(parser, didStartElement: elementName, namespaceURI: namespaceURI, qualifiedName: qName, attributes: attributeDict)—This method is called on the delegate when the parser finds an opening tag (e.g., <title>).

parser(parser, didEndElement: elementName, namespaceURI: namespaceURI, qualifiedName: qName)—This method is called on the delegate when the parser finds a closing tag (e.g., </title>).

parser(parser, foundCharacters: string)—This method is called on the delegate when the parser starts parsing the characters enclosed by the tags. The parser may take several attempts to parse all the characters. To get the entire value, we have to concatenate the strings received from the **string** attribute until the closing tag is found.

Because a parser may be created from a URL, the code to load the XML document is simpler than the one we used to parse JSON documents. We have to create the **NSXMLParser** object with a URL pointing to an XML document and then call the **parse()** method to start parsing the document.

Listing 27-11: Parsing an XML document

```
import UIKit

class ViewController: UIViewController, NSXMLParserDelegate {
    override func viewDidLoad() {
        super.viewDidLoad()

        let url = NSURL(string: "https://query.yahooapis.com/v1/public/yql?
q=select%20item.condition%20from%20weather.forecast%20where%20woeid%20%3D%202487889&format=xml&env

        if let parser = NSXMLParser(contentsOfURL: url!) {
            parser.delegate = self
            parser.parse()
        }

        func parser(parser: NSXMLParser, didStartElement elementName: String, namespaceURI: String?,
qualifiedName qName: String?, attributes attributeDict: [String : String]) {
            if elementName == "yweather:condition" {
                print(attributeDict["temp"])
            }
        }
    }
}
```

The Yahoo weather API does not store the values between tags but in attributes inside the opening tags (e.g., `<yweather temp="58">`). For this reason, we only need to implement the `parser(parser, didStartElement: elementName, namespaceURI: namespaceURI, qualifiedName: qName, attributes: attributeDict)` method that is called when an opening tag is found. This method receives the name of the current element and a dictionary containing the names and values of its attributes. To get the value we want, all we have to do is to check if the current tag is the one containing those values and then read it from the `attributesDict` dictionary. In Listing 27-11 we check if the value of the `elementName` attribute is equal to the element that is storing the weather data (`yweather:condition`) and then read its `temp` attribute to get the temperature.

Do It Yourself: Replace the `ViewController` class in the project created for JSON documents by the class in Listing 27-11 and run the application. You should see the value of the temperature printed in the console. You can get the link to the XML document from Yahoo's website. Go to developer.yahoo.com/weather/, select the example of San Diego from the list and select the option XML from the responses. The web page shows the XML code generated for the document and the link to the document below. You can select other examples and adapt the code to extract the information according to the keys available in each document.

27.4 Social Networks

Social networks like Facebook and Twitter also provide their own services to share data. Using these services we can get information from the user's account and also post messages and pictures on behalf of the user. These are very complex services that usually require us to open a developer account, register our application and even ask permission to implement some features, but Apple has simplified the process integrating the most popular networks into their mobile devices.

UIActivityViewController

The simplest way to publish data to a social network is the Activity Controller. This is a view controller created from the `UIActivityViewController` class that presents a view with icons to let the users select the application they want to share the information with (the applications are called *Activities*). The controller cannot only send the information to social networks but also to applications like Mail, Notes, Camera Roll, and the printer, among others. If a social network is selected, the view controller takes the information of the user's account from the device and performs the connection to upload the data.

The `UIActivityViewController` class is a very powerful but simple class. To create the view controller we just have to initialize the object and determine the options we want to offer. The following are its initializer and property.

`UIActivityViewController(activityItems: items, applicationActivities: activities)`—This initializer creates an Activity Controller to send the information specified by the **`items`** attribute. The **`items`** attribute is an array of `AnyObject` objects with the data we want to share, and the **`activities`** attribute is an array of `UIActivity` objects that represent custom services we want to include in the list (declared as `nil` when we only need to present the activities provided by the system).

`excludedActivityTypes`—This property sets or returns an array of strings that represent the activities we want to exclude from the controller. The `UIActivity` class defines a series of constants to represent the activities defined by the system. The constants available are `UIActivityTypePostToFacebook`, `UIActivityTypePostToTwitter`, `UIActivityTypePostToWeibo`, `UIActivityTypeMessage`, `UIActivityTypeMail`, `UIActivityTypePrint`, `UIActivityTypeCopyToPasteboard`, `UIActivityTypeAssignToContact`, `UIActivityTypeSaveToCameraRoll`, `UIActivityTypeAddToReadingList`, `UIActivityTypePostToFlickr`, `UIActivityTypePostToVimeo`, `UIActivityTypePostToTencentWeibo`, and `UIActivityTypeAirDrop`.

The information that is going to be sent to the view controller depends on the destination we consider appropriate for our application. For example, if we want to let the user store a message in the Notes application, we just have to provide the text. Social networks can receive three values: a message, a URL, and an image. In the following example we decided to use the capacity of Facebook and Twitter to post images and created an interface where the user can post pictures taken with the camera.

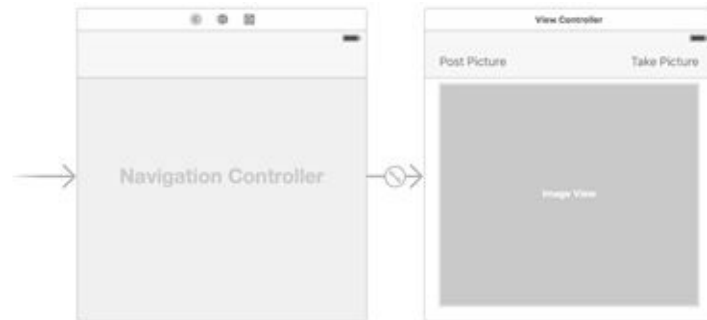


Figure 27-8: Interface to post pictures on social networks

The interface of Figure 27-8 includes an Image View to show the picture and two bar buttons, one called Take Picture to open the `UIImagePickerController` controller used to take the picture and another called Post Picture to open the `UIActivityViewController` controller to let the user post the picture in the social network of preference.

The code to configure and open the view controller for the camera is exactly the same we used in previous examples. When the user takes a picture we store it in a property called `picture` for later use.

Listing 27-12: Taking a picture to send to social networks

```
import UIKit

class ViewController: UIViewController, UINavigationControllerDelegate, UIImagePickerControllerDelegate {
    @IBOutlet weak var pictureView: UIImageView!
    var picture: UIImage?

    @IBAction func takePicture(sender: UIBarButtonItem) {
        let mediaPicker = UIImagePickerController()
        mediaPicker.delegate = self
        let sourceAvailable = UIImagePickerController.isSourceTypeAvailable(.Camera)

        if sourceAvailable {
            mediaPicker.sourceType = .Camera
            mediaPicker.mediaTypes = ["public.image"]
            mediaPicker.allowsEditing = false
            mediaPicker.cameraCaptureMode = .Photo
```

```

    presentViewController(mediaPicker, animated: true, completion: nil)
  } else {
    print("The media is not available")
  }
}

func imagePickerController(picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String :
AnyObject]) {
    picture = info[UIImagePickerControllerOriginalImage] as? UIImage
    pictureView.image = picture
    dismissViewControllerAnimated(true, completion: nil)
}

func imagePickerControllerDidCancel(picker: UIImagePickerController) {
    dismissViewControllerAnimated(true, completion: nil)
}
}

```

There is nothing new in the code of Listing 27-12. The code responsible for posting the picture is in the Action for the Post Picture button. In this method we have to take the picture stored in the **picture** property, initialize a new **UIActivityViewController** object with this value, and present the view controller on the screen.

Listing 27-13: Posting the picture

```

@IBAction func postPicture(sender: UIBarButtonItem) {
    if let image = picture {
        let width = image.size.width / 3
        let height = image.size.height / 3
        let imageSize = CGSize(width: width, height: height)
        UIGraphicsBeginImageContextWithOptions(imageSize, false, 0.0)
        image.drawInRect(CGRect(x: 0, y: 0, width: width, height: height))
        let newImage = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()

        let items = [newImage]
        let controller = UIActivityViewController(activityItems: items, applicationActivities: nil)
        presentViewController(controller, animated: true, completion: nil)
    }
}

```

The pictures may be posted in some services with the original orientation. This means that if we take the picture with the device in landscape mode, it could have the wrong orientation in the final destination. To avoid the issue, in the code of Listing 27-13 we

create an Image Context and generate a new image that will always have the same orientation. Because of the high resolution of the cameras, we also reduce the size of the picture three times, to create an image that is more suitable for social networks.

The `items` array in our example includes only the picture we want to post, but it could also include a text and a URL (an `NSURL` object). The controller takes this information and presents it to the user in an additional view after the social network is selected. This view lets the user incorporate to the post additional information such as the location or who is going to be able to see it. Figure 27-9 shows the views generated by the Activity Controller.



Figure 27-9: Posting to Facebook with an Activity Controller

The Basics: Apple devices have integrated the social networks into the system. Users can set their Facebook and Twitter accounts from the Settings app and then every application can use this information to access the accounts. Because of this, users don't have to insert the user name and password every time they want to connect to the network. Classes like `UIActivityViewController` take advantage of this feature to simplify our work.

By default, the Activity Controller shows every service available, but we can exclude some of those services with the controller's `excludedActivityTypes` property. For example, if we want to exclude everything except Facebook, we can replace the `postPicture()` Action by the following.

Listing 27-14: *Excluding activities*

```
@IBAction func postPicture(sender: UIBarButtonItem) {  
    if let image = picture {  
        let width = image.size.width / 3  
        let height = image.size.height / 3  
        let imageSize = CGSize(width: width, height: height)
```

```

    UIGraphicsBeginImageContextWithOptions(imageSize, false, 0.0)
    image.drawInRect(CGRect(x: 0, y: 0, width: width, height: height))
    let newImage = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()

    let items = [newImage]
    let controller = UIActivityViewController(activityItems: items, applicationActivities: nil)

    controller.excludedActivityTypes = [UIActivityTypePostToTwitter, UIActivityTypePostToWeibo,
    UIActivityTypeMessage, UIActivityTypeMail, UIActivityTypePrint, UIActivityTypeCopyToPasteboard,
    UIActivityTypeAssignToContact, UIActivityTypeSaveToCameraRoll, UIActivityTypeAddToReadingList,
    UIActivityTypePostToFlickr, UIActivityTypePostToVimeo, UIActivityTypePostToTencentWeibo,
    UIActivityTypeAirDrop]
    presentViewController(controller, animated: true, completion: nil)
}
}

```

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add an Image View and two bar buttons to the initial view called Post Picture and Take Picture (Figure 27-8). Connect the Image View to the `ViewController` class with an Outlet called `pictureView`, the Take Picture button with an Action called `takePicture()`, and the Post Picture button with an Action called `postPicture()`. Complete the class with the codes of Listings 27-12 and 27-13. Run the application. Take a picture, press the Post Picture button, and select the Facebook icon. You should see something similar to Figure 27-9. Press the post button. You should see a new post in your Facebook account (the account registered on your device or the simulator).

SLComposeViewController

The Activity Controller is an easy way to let our users post messages to social networks but it presents additional options that may sometimes result in confusion. If we want to customize the process we have to work directly with the classes provided by the Social framework. The framework includes all the classes necessary to perform the connection with the user's account, create the requests, and process the response.

Despite the fact that we can manage every aspect of the connection with the Social framework, we still have to provide an interface for the user to check and prepare the information to be published. For this reason, one of the classes included in the framework is the `SLComposeViewController` class, designed to create a view controller that posts messages to social networks only (at this moment the networks available are Facebook, Twitter, and Weibo). The class provides the following initializer, property, and methods.

SLComposeViewController(forServiceType: type)—This initializer creates a view controller to post a message to the social network specified by the **type** attribute. The class defines the constants **SLServiceTypeFacebook**, **SLServiceTypeTwitter**, **SLServiceTypeSinaWeibo**, and **SLServiceTypeTencentWeibo** to specify the social network.

completionHandler—This property sets a closure that will be executed when the view controller is dismissed. The closure receives an enumeration value of type **SLComposeViewControllerResult** to report the state of the operation. The possible values are **Cancelled** and **Done**.

setInitialText(text)—This method sets the post’s text.

addImage(image)—This method sets the post’s image.

addURL(URL)—This method sets the post’s URL.

Using the **SLComposeViewController** class we are able to select the social network from code, so the only view the user sees is the view to edit the values and post the message. The following example creates a view controller to post a picture on Facebook.

Listing 27-15: Posting only on Facebook

```
@IBAction func postPicture(sender: UIBarButtonItem) {
    if let image = picture {
        let width = image.size.width / 3
        let height = image.size.height / 3
        let imageSize = CGSize(width: width, height: height)
        UIGraphicsBeginImageContextWithOptions(imageSize, false, 0.0)
        image.drawInRect(CGRect(x: 0, y: 0, width: width, height: height))
        let newImage = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()

        let controller = SLComposeViewController(forServiceType: SLServiceTypeFacebook)
        controller.addImage(newImage)
        presentViewController(controller, animated: true, completion: nil)
    }
}
```

The **postPicture()** method of Listing 27-15 replaces the same method from the previous

project. The code creates an `SLComposeViewController` object configured to post the message on Facebook, adds the picture to the controller with the `addImage()` method, and presents the controller on the screen. The view produced by the controller allows the user to edit the message and post it (Figure 27-9, right).

As well as the Activity Controller, this controller can post a text and a URL. We just have to add the values with the corresponding methods (`setInitialText()` and `addURL()`). Additionally, we can set a completion handler to perform tasks according to the action performed. The following example shows an Alert View if the value of the `result` attribute indicates that the posting was successful.

Listing 27-16: Reporting the state of the operation

```
@IBAction func postPicture(sender: UIBarButtonItem) {
    if let image = picture {
        let width = image.size.width / 3
        let height = image.size.height / 3
        let imageSize = CGSize(width: width, height: height)
        UIGraphicsBeginImageContextWithOptions(imageSize, false, 0.0)
        image.drawInRect(CGRect(x: 0, y: 0, width: width, height: height))
        let newImage = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()

        let controller = SLComposeViewController(forServiceType: SLServiceTypeFacebook)
        controller.addImage(newImage)
        controller.completionHandler = { (result) in
        if result == .Done {
        NSOperationQueue.mainQueue().addOperationWithBlock({
        let alert = UIAlertController(title: "Message Posted", message: nil, preferredStyle: .Alert)
        let action = UIAlertAction(title: "OK", style: .Default, handler: nil)
        alert.addAction(action)
        self.presentViewController(alert, animated: true, completion: nil)
        })
        }
        }
        presentViewController(controller, animated: true, completion: nil)
    }
}
```

Do It Yourself: Replace the `postPicture()` method in your project by the methods of Listing 27-15 or 27-16. Run the application, take a picture, and press the Post Picture button. You should see a view similar to Figure 27-9, right.

IMPORTANT: The `SLComposeViewController` class belongs to the Social framework. To work with the classes provided by this framework you must import it with the `import` statement (`import Social`).

SLRequest

Social networks like Facebook and Twitter provide extensive APIs to let developers access users' accounts from mobile and web applications. The view controllers implemented in previous examples access these APIs to post new messages, but we can customize the request to also read old posts, update and delete information, search for posts and users, and more. The Social framework offers the `SLRequest` class to access these capabilities and create a customized request. The class takes care of creating the JSON documents necessary to request and post information, provide the required certificates to access the API, and get the response. The following are the initializer, property, and method necessary to create and execute the request.

SLRequest(forServiceType: `type`, requestMethod: `method`, URL: `URL`, parameters: `parameters`)—This initializer creates a request for the social network indicated by the `type` attribute. The `type` attribute is defined by one of the four constants provided by the framework to represent the social networks: `SLServiceTypeFacebook`, `SLServiceTypeTwitter`, `SLServiceTypeSinaWeibo`, and `SLServiceTypeTencentWeibo`. The `method` attribute is an enumeration of type `SLRequestMethod` that determines the purpose of the connection. The possible values are `GET`, `POST`, `DELETE`, and `PUT`. The `URL` parameter is an `NSURL` with the URL where the JSON document is located or has to be sent. Finally, the `parameters` attribute is a dictionary with the configuration values for the request.

account—This property sets or returns the user account associated with the request. The Social framework provides the `ACAccount` class to define this value.

performRequestWithHandler(handler)—This method performs the request. The `handler` attribute is the closure to be executed to process the response. The closure receives three attributes: an `NSData` object with the data returned by the request, an `NSHTTPURLResponse` object with the status of the request, and an `NSError` value to indicate if an error occurred.

The request requires a reference to the user's account we want to connect to. As we

mentioned before, iOS allows the users to register their social network's accounts on the device. To get access to these accounts, Apple provides the Accounts framework. Before creating a request, we have to use this framework to get the information of the account we are going to use to access the social network's API. The framework defines the `ACAccountStore` class with the following methods to get the list of accounts registered on the device.

accountTypeWithAccountTypeIdentifier(identifier)—This method returns an object of type `ACAccountType` that represents the type of account specified by the **identifier** attribute. The `ACAccountType` class defines the following properties to provide as identifiers: `ACAccountTypeIdentifierFacebook`, `ACAccountTypeIdentifierSinaWeibo`, `ACAccountTypeIdentifierTwitter`, and `ACAccountTypeIdentifierTencentWeibo`.

accountsWithAccountType(type)—This method returns a list of `ACAccount` objects that represents the accounts registered on the device that match the type specified by the **type** attribute. The attribute is an `ACAccountType` object.

requestAccessToAccountsWithType(type, options: options, completion: completion)—This method asks permission to the user to access the accounts of the type specified by the **type** attribute. The **type** attribute is an `ACAccountType` object that represents the type of accounts we want to access, the **options** attribute is a dictionary with the options required by the social network, and the **completion** attribute is the closure to be executed after the user grants or denies access. The closure receives two attributes: a Boolean value to indicate the user's decision and an `NSError` value to report errors.

The `accountsWithAccountType()` method returns a list of objects that represent the accounts registered on the device. These are objects of type `ACAccount`. The class includes the following properties to get the account's values.

username—This property returns a string with the account's user name.

identifier—This property returns a string with the account's identifier.

accountDescription—This property returns a string with the account's description.

accountType—This property returns an `ACAccountType` object that determines the account's type.

Access to the user's accounts is restricted, so we have to ask for permission. The following example asks for permission to access the Twitter accounts.

Listing 27-17: Accessing Twitter's accounts

```
import UIKit
import Accounts
import Social

class ViewController: UIViewController {
    var account: ACAccount!

    override func viewDidLoad() {
        super.viewDidLoad()

        let accountList = ACAccountStore()
        let type = accountList.accountTypeWithAccountTypeIdentifier(ACAccountTypeIdentifierTwitter)
        accountList.requestAccessToAccountsWithType(type, options: nil, completion: { (success, error) in
            if success {
                let accounts = accountList.accountsWithAccountType(type)
                self.account = accounts.first as! ACAccount
            }
        })
    }
}
```

There can only be one Facebook account per device, but multiple Twitter accounts are allowed. In Listing 27-17 we request access to all the Twitter accounts on the device and assign the first one on the list to our `account` property. Once we get this reference, we can create requests to access Twitter's API.

The Basics: For this example we use the first Twitter account found, but you can select one from code by looking at the value of the `username` property or present the list to the users and let them decide.

Every API can generate custom JSON documents with only the information requested, but they usually provide a common URL to retrieve this document. For example, the Twitter API includes a URL where we can get a document with the list of tweets posted on a specific account. The number of tweets and the accounts depends on the configuration of the request, but the URL to obtain these tweets is always the same. For this example we have created an Action that responds to a button in the interface to get the last 10 tweets posted in the user's account. The parameters requested by the URL used in this example are called "screen_name" and "count". The "screen_name" parameter is the account's user name and the "count" parameter is the number of tweets we want to retrieve.

Listing 27-18: Getting the list of tweets

```
@IBAction func readTweets(sender: UIButton) {
    let url = NSURL(string: "https://api.twitter.com/1.1/statuses/user_timeline.json")
    let parameters = ["screen_name": account.username, "count": "10"]
    let request = SLRequest(forServiceType: SLServiceTypeTwitter, requestMethod: SLRequestMethod.GET, URL:
url!, parameters: parameters)
    request.account = account

    request.performRequestWithHandler({ (data, response, error) in
        let resp = response as NSHTTPURLResponse
        let status = resp.statusCode
        if status == 200 {
            self.showValues(data!)
        }
    })
}

func showValues(data: NSData) {
    let values = try? NSJSONSerialization.JSONObjectWithData(data, options: []) as! [AnyObject]
    if let items = values {
        for item in items {
            if let text = item["text"] as? String {
                print(text)
            }
        }
    }
}
```

The documents provided by the Twitter API are JSON files, so we process the response the same way we did for other JSON documents before. In this case, we want to get and print the texts of the tweets, so we read the item with the “text” key of every tweet and print it in the console.

Do It Yourself: Create a new Single View Application project. Add a button to the initial view and connect it to the **ViewController** class with an Action called **readTweets()**. Complete the **ViewController** class with the code of Listings 27-17 and 27-18. Run the application. You should see the tweets from the Twitter account you have registered on your device printed in the console.

IMPORTANT: As always, the parameters and the keys inside the document are customized by the API’s developers. Fortunately, they provide documentation for other developers to know how the API works. If you want to customize the process of

posting and retrieving information from social networks, you should read these documents. The following are the links to the most popular APIs at the moment: **developers.facebook.com/docs/** (Facebook), **open.weibo.com/wiki/** (Sina Weibo), **dev.twitter.com/docs** (Twitter), and **developer.linkedin.com/rest** (LinkedIn).

Chapter 28

Contacts

28.1 Contacts

All mobile devices, even those that can't make phone calls, allow users to store contact information through the Contacts app. Apple provides the Contacts framework to let developers access this data. The framework includes classes to get the list of contacts, update, or delete them, and create new ones. The access to the contacts database is provided by the `CNContactStore` class. Because of the sensibility of this information, one of the jobs of this class is to provide the tools to ask for permission. The following are the methods defined for this purpose.

requestAccessForEntityType(type, completionHandler: block)—This method requests access to the user's contacts. The **type** attribute is an enumeration of type `CNEntityType` that determines the type of values we want to access. The only value available at this moment is `Contacts`. The **block** attribute is a closure that will be executed after the user makes a decision. The closure receives two values: a Boolean value that determines if permission was granted or not and an `NSError` value to report errors.

authorizationStatusForEntityType(type)—This method returns the current authorization status. The value returned is an enumeration of type `CNAuthorizationStatus` with the values `NotDetermined` (by default), `Restricted`, `Denied`, and `Authorized`.

The class offers three methods to read the contacts. We can get multiple contacts with a request, filter contacts with a predefined predicate, or retrieve individual contacts through their identifiers.

enumerateContactsWithFetchRequest(request, usingBlock: block)—This method gets the list of contacts that match the request defined by the **request** attribute. The **request** attribute is a `CNContactFetchRequest` object with the configuration of the request, and the **block** attribute is a closure that processes the values returned. The closure is called for every contact and receives two values: a `CNContact` object that represents the contact and a pointer to a Boolean value that we can set to `true` to stop the enumeration.

unifiedContactsMatchingPredicate(predicate, keysToFetch: keys)—This method returns an array with the contacts that match the predicate defined by the **predicate** attribute. The **predicate** attribute is a predefined `NSPredicate` object returned by methods of the `CNContact` class, and the **keys** attribute is a set of constants that represent the properties of the contact we want to retrieve.

unifiedContactWithIdentifier(identifier, keysToFetch: keys)—This method returns a single contact the identifier of which matches the value of the **identifier**

attribute. The **identifier** attribute is a string that identifies the contact, and the **keys** attribute is a set of constants that represent the properties of the contact we want to retrieve.

The contacts are returned as objects of the `CNContact` class. The class includes the following properties to retrieve the values.

identifier—This property returns a string that identifies the contact.

givenName—This property returns a string with the contact's given name.

middleName—This property returns a string with the contact's middle name.

familyName—This property returns a string with the contact's family name.

birthday—This property returns an `NSDateComponents` object with the components that represent the contact's birthday.

imageData—This property returns an `NSData` object with data that represents the contact's profile picture.

thumbnailImageData—This property returns an `NSData` object with data that represents a thumbnail of the contact's profile picture.

phoneNumbers—This property returns an array of `CNLabelledValue` objects that represent phone numbers.

emailAddresses—This property returns an array of `CNLabelledValue` objects that represent email addresses.

postalAddresses—This property returns an array of `CNLabelledValue` objects that represent postal addresses.

note—This property returns a string with the contact's note.

The `CNContact` class provides properties to retrieve every possible value in the contact, but not all the properties are available for every contact retrieved. A `CNContact` object only includes the values we order the methods to retrieve. To determine the values we want, the `CNContact` class defines a set of constants we can provide as keys for the fetching methods. The most frequently used are `CNContactGivenNameKey`, `CNContactMiddleNameKey`, `CNContactFamilyNameKey`, `CNContactBirthdayKey`, `CNContactImageDataKey`, `CNContactThumbnailImageDataKey`, `CNContactPhoneNumbersKey`, `CNContactEmailAddressesKey`, `CNContactPostalAddressesKey`, and `CNContactNoteKey`.

A typical application accesses the contacts database to let the user select one of the contacts and send a message, perform a call, or use the information in any way necessary.

The interface introduced next presents a simple example that illustrates these types of applications. The initial view includes a button to let the user open a table with a list of the contacts available on the device.



Figure 28-1: Interface to list contacts

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add a button called Select Contact to the initial view. Add a Table View Controller to the Storyboard. Connect the Select Contact button to the Table View with a Show Segue. Select the cell and change the style to Basic and the identifier to “contactCell”. Add a Navigation Item to the Table View’s Navigation Bar. Add a bar button to the Navigation Bar, as shown in Figure 28-1.

In our example, the only function of the initial view is to provide access to the Table View, so the code necessary to retrieve and show the list of contacts goes on the `UITableViewController` subclass that controls this view. We call it `ContactsViewController`.

Listing 28-1: Listing contacts

```
import UIKit
import Contacts

class ContactsViewController: UITableViewController {
    var contactList: [CNContact]!

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)

        let store = CNContactStore()
        store.requestAccessForEntityType(.Contacts, completionHandler: { (success, error) in
            if success {
                let request = CNContactFetchRequest(keysToFetch: [CNContactGivenNameKey,
                                                                CNContactFamilyNameKey])
                do {
                    self.contactList = []
```

```

    try store.enumerateContactsWithFetchRequest(request, usingBlock: { (contact, status) in
        self.contactList.append(contact)
    })
} catch {
    print("Error")
}
NSOperationQueue.mainQueue().addOperationWithBlock({
    self.tableView.reloadData()
})
}
})
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    if contactList != nil {
        return contactList.count
    }
    return 0
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("contactCell", forIndexPath: indexPath)

    let contact = contactList[indexPath.row]
    cell.textLabel?.text = "\(contact.givenName) \(contact.familyName)"
    return cell
}
}

```

Since we are using a Table View to list the contacts, we have to access the contact database every time the view is shown to update the values. The `viewWillAppear()` method creates an instance of the `CNContactStore` class and then calls the `requestAccessForEntityType()` method to ask for permission. If the request is successful, the method executes a closure that we can use to read the contacts. In this example we use the `enumerateContactsWithFetchRequest()` method to get the full list. This method requires a `CNContactFetchRequest` object to specify the request. The `CNContactFetchRequest` class defines the following initializer and properties to create and configure a request.

CNContactFetchRequest(keysToFetch: **keys)**—This initializer creates a request for the properties specified by the **keys** attribute.

mutableObjects—This property sets or returns a Boolean value that determines if

the object that represent the contacts will be mutable or immutable.

unifyResults—This property sets or returns a Boolean value that determines if the same contacts from different sources, like iCloud and Facebook, will be unified and returned as one.

sortOrder—This property sets or returns a value that determines how the contacts will be sorted. It is an enumeration of type `CNContactSortOrder` with the values `None`, `UserDefault` (by default), `GivenName`, and `FamilyName`.

predicate—This property sets or returns the predicate used in the request. It can only be set as a predefined `NSPredicate` object returned by methods of the `CNContact` class.

In Listing 28-1 we use a simple request with no predicate or order, but with the keys `CNContactGivenNameKey`, `CNContactFamilyNameKey` to retrieve the given and family name. The table accesses these values from the `givenName` and `familyName` properties of the `CNContact` object to create each cell. The result is shown in Figure 28-2.

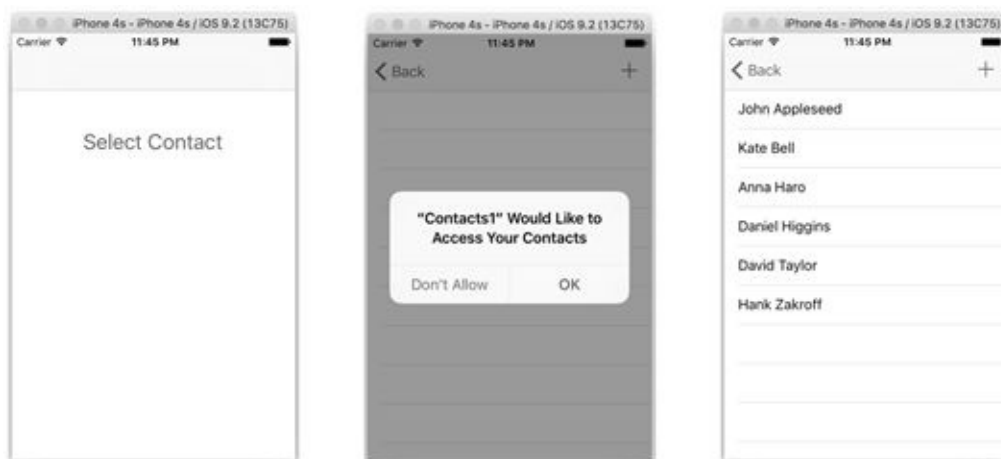


Figure 28-2: Application showing the list of contacts

Do It Yourself: Create a subclass of the `UITableViewController` class, call it `ContactsViewController`, and assign it to the Table View (Figure 28-1, right). Complete the class with the code of Listing 28-1 and run the application in the simulator. When the system asks permission to access the contacts, press the OK button (Figure 28-2, center). You should see the list of sample contacts included in the simulator.

IMPORTANT: The properties available in the `CNContact` object depend on the keys specified in the request. For example, in the code of Listing 28-1 we can only access the `givenName` and `familyName` properties because only the `CNContactGivenNameKey`, `CNContactFamilyNameKey` keys were declared. If we try to access a property that was not requested, the application will crash. You can check if a property is available in a

CNContact objects calling the `isKeyAvailable(key)` or `areKeysAvailable(keys)` methods provided by the CNContact class.

The previous example lists all the contacts available, but the results may be filtered with a predicate. The Contacts framework doesn't support generic predicates, we can only use the predefined predicates created from the following type methods provided by the CNContact class.

predicateForContactsMatchingName(name)—This type method returns a predicate that filters the contacts by name. The **name** attribute is a string with the name we are looking for.

predicateForContactsWithIdentifiers(identifiers)—This type method returns a predicate that searches for the contacts with the identifiers specified by the **identifiers** attribute. The attribute is an array of strings that represent the identifiers of the contacts we are looking for.

predicateForContactsInGroupWithIdentifier(group)—This type method returns a predicate that searches for the contacts inside a group. The **group** attribute is a string identifying the group.

predicateForContactsInContainerWithIdentifier(container)—This type method returns a predicate that searches for the contacts inside a container. The **container** attribute is a string identifying the container.

When using a predicate we can take advantage of the `unifiedContactsMatchingPredicate()` method that returns an array with all the contacts found (we don't have to add the contacts one by one to the array as we did before). The following example searches for the contacts with the name "John".

Listing 28-2: Filtering the contacts with a predicate

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    let store = CNContactStore()
    store.requestAccessForEntityType(.Contacts, completionHandler: { (success, error) in
        if success {
            do {
                let predicate = CNContact.predicateForContactsMatchingName("John")
                self.contactList = try store.unifiedContactsMatchingPredicate(predicate, keysToFetch:
[CNContactGivenNameKey, CNContactFamilyNameKey])
            } catch {
```

```

print("Error")
}
NSOperationQueue.mainQueue().addOperationWithBlock({
self.tableView.reloadData()
})
}
})
}

```

Do It Yourself: Replace the `viewWillAppear()` method in the `ContactsViewController` class of Listing 28-1 by the method of Listing 28-2. Run the application. You should only see the contacts with the name “John”.

Reading Contacts

The `ContactsViewController` class of our example only accesses the contact’s first and last name, but we can read any value we want. Figure 28-3 adds two views to our interface: a detail view to show the contact’s full name and address, and a view with a Text Field to add a new contact.

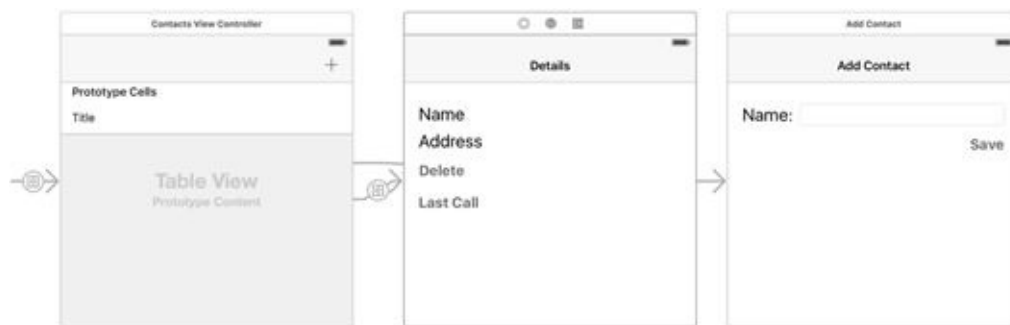


Figure 28-3: Additional views to modify, delete, and create new contacts

Do It Yourself: Add two new views to the Storyboard. Add Navigation Items to the Navigation Bar of each view and change their titles to Details and Add Contact. Connect the Table View’s cell to the Details view and the bar button on the Table View’s Navigation Bar to the Add Contact view with Show Segues (selecting a cell opens the Details view and pressing the + button opens the Add Contact view). Select the Segue that connects the cell with the Details view and assign it the identifier “showDetails”. Add two labels and two buttons to the Details view and a Text Field and a button to the Add Contact view, as shown in Figure 28-3. Select the labels and change the number of lines to 0 from the Attributes Inspector panel, so the labels can

show long names and full addresses.

The main purpose of the Details view is to show additional information about the contact selected in the Table View, so the view controller for this view has to access the contact and show its values on the screen. We call it **DetailsViewController**.

Listing 28-3: Showing contact's details

```
import UIKit
import Contacts

class DetailsViewController: UIViewController {
    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var addressLabel: UILabel!
    var contactID: String!
    var contact: CNContact!

    override func viewDidLoad() {
        super.viewDidLoad()

        if contactID != nil {
            let store = CNContactStore()
            do {
                contact = try store.unifiedContactWithIdentifier(contactID, keysToFetch: [CNContactGivenNameKey, CNContactFamilyNameKey, CNContactPostalAddressesKey, CNContactNoteKey])
                self.nameLabel.text = "\(contact.givenName) \(contact.familyName)"
            } catch {
                print("Error")
            }
        }
    }
}
```

To read the selected contact, we use the identifier provided by the Table View and call the **unifiedContactWithIdentifier()** method. This method returns a **CNContact** object with the contact's information. Notice that this time we ask for four properties: the given name, the family name, postal addresses, and notes.

The Details view needs to receive from the Table View the identifier of the contact selected by the user. For this purpose, the **DetailsViewController** class includes the **contactID** property to store the value. In the **ContactsViewController** class we have to include the **prepareForSegue()** to assign the identifier to this property before the Details view is loaded.

Listing 28-4: Sending the contact’s identifier to the Details view

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?){
    if segue.identifier == “showDetails” {
        let controller = segue.destinationViewController as! DetailsViewController
        if let path = tableView.indexPathForSelectedRow {
            let contact = contactList[path.row]
            controller.contactID = contact.identifier
        }
    }
}
```

Do It Yourself: Create a subclass of the `UIViewController` class, call it `DetailsViewController`, and assign it to the Details view (Figure 28-3, center). Connect the labels in the Details view to the `DetailsViewController` class with Outlets called `nameLabel` and `addressLabel`. Complete the class with the code of Listing 28-3. Add the method of Listing 28-4 to the `ContactsViewController` class of Listing 28-1. Run the application and select a contact. You should see the Details view with the contact’s full name.

At this point the Details view only shows the contact’s full name. To display additional information, we have to learn first how that information is stored. Properties like `givenName` or `familyName` return single values that we can read and print right away, but others like `emailAddresses` and `phoneNumbers` store arrays of `CNLabeledValue` objects that contain the original value and a label that identifies the purpose of the value. For example, a contact can have a phone number for the office and another for the home. Reading the label, we can determine what the value represents. The following are the initializer and properties included in the `CNLabeledValue` class to create and read label objects.

`CNLabeledValue(label: label, value: value)`—This initializer creates a `CNLabeledValue` object from the values provided by the attributes. The **`label`** attribute is a string that determines the purpose of the value, and the **`value`** attribute is the property value.

`label`—This property returns the label of the property value.

`value`—This property returns the property value.

Labels are string values. We can designate our own labels or use the constants defined by the `CNLabeledValue` class. For general purposes, the class defines three constants `CNLabelHome`, `CNLabelWork`, and `CNLabelOther`. The constant available for email accounts is

CNLabelEmailiCloud. And phone numbers can be described as CNLabelPhoneNumberiPhone, CNLabelPhoneNumberMobile, CNLabelPhoneNumberMain, CNLabelPhoneNumberHomeFax, CNLabelPhoneNumberWorkFax, CNLabelPhoneNumberOtherFax, and CNLabelPhoneNumberPager.

For properties like `emailAddresses`, the value of the `CNLabeledValue` object is just a string with the email account, but others like postal addresses and phone numbers require their own objects to store all the values associated with the property. For example, the framework defines the `CNPhoneNumber` class to store phone numbers. The class includes the following initializer and property to create and read these types of values.

CNPhoneNumber(stringValue: `phone`)—This initializer creates a `CNPhoneNumber` object with the phone number specified by the `phone` attribute. The attribute is a string value and may include other characters besides numbers (e.g., (555) 555.5555).

stringValue—This property returns a string with the phone number.

Postal addresses are stored in objects of the `CNPostalAddress` class. The framework defines the `CNPostalAddress` class to return the values and the `CNMutablePostalAddress` class to create new objects, but both classes include the same properties to define and retrieve the values.

street—This property sets or returns a string with the street.

city—This property sets or returns a string with the name of the city.

state—This property sets or returns a string with the name of the state.

postalCode—This property sets or returns a string with the postal code.

country—This property sets or returns a string with the name of the country.

Because of how they are stored, to get a particular value we have to check its label first and then read the properties to get the value we are really looking for. The following example updates the `viewDidLoad()` method of the `DetailsViewController` class to show the home address of the selected contact on the screen.

Listing 28-5: Getting the contact's home address

```
override func viewDidLoad() {
    super.viewDidLoad()
    if contactID != nil {
        let store = CNContactStore()
        do {
```



```

    contact = try store.unifiedContactWithIdentifier(contactID, keysToFetch: [CNContactGivenNameKey,
CNContactFamilyNameKey, CNContactPostalAddressesKey, CNContactNoteKey])

    self.nameLabel.text = "\(contact.givenName) \(contact.familyName)"

    var address: CNPostalAddress!
    for item in contact.postalAddresses {
        if item.label == CNLabelHome {
            address = item.value as! CNPostalAddress
        }
    }

    if address != nil {
        self.addressLabel.text = "\(address.street), \(address.state)"
    } else {
        self.addressLabel.text = "Unknown"
    }
    } catch {
        print("Error")
    }
}
}

```

The code of Listing 28-5 looks for a **CNLabeledValue** object in the array returned by the **postalAddresses** property with a label that matches the constant **CNLabelHome**. When an object is found (which means that there is a home address stored for this contact), the value is casted as a **CNPostalAddress** object and assigned to the **address** variable. This value is used next to extract the street and the state and show them on the screen.

Do It Yourself: Replace the **viewDidLoad()** method in the **DetailsViewController** class of Listing 28-3 by the method of Listing 28-5. Run the application and select a contact. You should see the contact’s name and address on the screen.

Formatters

The way names and addresses are written varies from one country to another. For example, in Japan the family name goes before the given name. To make sure that these values adapt to international conventions, the Contacts framework includes two formatter classes: **CNContactFormatter** to format names and **CNPostalAddressFormatter** to format addresses. Both classes offer type methods to format the values and return a string.

stringFromContact(contact, style: style)—This type method, included in the **CNContactFormatter** class, produces a string with the contact’s full name formatted according to the style specified by the **style** attribute. The **contact** attribute is a reference to the **CNContact** object we want to format, and the **style** attribute is an enumeration of type **CNContactFormatterStyle** with the values **FullName** (original name) and **PhoneticFullName** (phonetic name).

stringFromPostalAddress(postalAddress: address, style: style)—This type method, included in the **CNPostalAddressFormatter** class, produces a string with the contact’s address formatted according to the style specified by the **style** attribute. The **address** attribute is a reference to the **CNPostalAddress** object we want to format, and the **style** attribute is an enumeration of type **CNPostalAddressFormatterStyle** that at the moment only includes the value **MailingAddress**.

To generate the strings, these type methods have to access the contact’s properties, but as we mentioned before, only the properties specified in the request are available. Finding the keys for the address is easy because the formatter only requires the **postalAddresses** property, but the contact’s name may be composed by the values of different properties, including family and given names, prefixes, and suffixes, etc. Depending on where the user is located, the formatted string may include the values of one property or another. The **CNContactFormatter** class includes the **descriptorForRequiredKeysForStyle(style)** method that returns a keys descriptor we can use instead of the keys to get the properties we need in any possible situation.

The following example updates the **viewDidLoad()** method implemented in Listing 28-5 to show formatted values instead of the strings created before from single properties. Notice that, along with the keys we used before to get the addresses and the note, the **unifiedContactWithIdentifier()** method now includes the descriptor returned by the **descriptorForRequiredKeysForStyle()** method.

Listing 28-6: Formatting values

```
override func viewDidLoad() {
    super.viewDidLoad()

    if contactID != nil {
        let store = CNContactStore()
        do {
            let keys = CNContactFormatter.descriptorForRequiredKeysForStyle(.FullName)
            contact = try store.unifiedContactWithIdentifier(contactID, keysToFetch: [keys,
CNContactPostalAddressesKey, CNContactNoteKey])
            let name = CNContactFormatter.stringFromContact(contact, style: .FullName)
            self.nameLabel.text = name
            var address: CNPostalAddress!
```

```

for item in contact.postalAddresses {
    if item.label == CNLabelHome {
        address = item.value as! CNPostalAddress
    }
}

if address != nil {
    let street = CNPostalAddressFormatter.stringFromPostalAddress(address, style: .MailingAddress)
    self.addressLabel.text = street
} else {
    self.addressLabel.text = "Unknown"
}
} catch {
    print("Error")
}
}
}

```

The strings produced by the formatter vary from one country to another, but usually the name is presented in one single line and the address includes break lines to make it more readable, as shown in Figure 28-4.

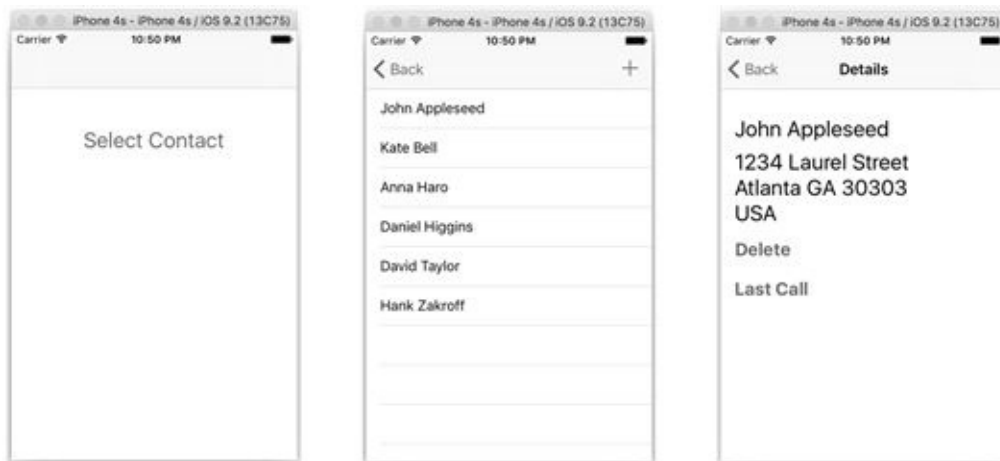


Figure 28-4: Contact with formatted values

Do It Yourself: Replace the `viewDidLoad()` method in the `DetailsViewController` class of Listing 28-3 by the method of Listing 28-6. Run the application. You should see something similar to Figure 28-4.

Working with Contacts

Besides accessing existent contacts, we can delete them, update their values, and create new ones. The processes are set up as requests that are later executed by the contact store. The Contacts framework includes the `CNSaveRequest` class to create these requests. We have to create an instance of this class and then call the method correspondent to the task we want to perform.

addContact(contact, toContainerWithIdentifier: identifier)—This method configures the request to add a new contact to the database. The **contact** attribute is a reference to the `CNContact` object we want to store, and the **identifier** attribute is a string that identifies the container where the contact will be stored (the value `nil` specifies the default container).

updateContact(contact)—This method configures the request to update the contact specified by the **contact** attribute. The attribute is a reference to the `CNContact` object we want to update.

deleteContact(contact)—This method configures the request to delete the contact specified by the **contact** attribute. The attribute is a reference to the `CNContact` object we want to delete.

Once the request is created and configured, we have to execute it from the contacts store. The `CNContactStore` class includes the following method for this purpose.

executeSaveRequest(request)—This method executes the request specified by the **request** attribute. The attribute is a `CNSaveRequest` object.

The `CNContact` objects are immutable, which means they can't be modified. To add, update, or delete a contact we have to work with mutable objects created from a subclass of `CNContact` called `CNMutableContact`. The `CNContact` class, as well as other Objective-C classes, includes the `mutableCopy()` method to turn an immutable object into its mutable version. The following example implements the Action for the Delete button of our application's Details view (see Figure 28-3, center). The code turns the contact selected by the user into a `CNMutableContact` object and then creates the request to delete it.

Listing 28-7: Deleting a contact

```
@IBAction func deleteContact(sender: UIButton) {  
    if contact != nil {  
        let deleteContact = contact.mutableCopy() as! CNMutableContact
```

```

let request = CNSaveRequest()
request.deleteContact(deleteContact)
do {
let store = CNContactStore()
try store.executeSaveRequest(request)
} catch {
print("Error")
}
}
navigationController?.popViewControllerAnimated(true)
}

```

The `executeSaveRequest()` method can throw an error, so we have to execute it inside a `do catch` block. Once the contact is deleted, the Action takes the user back to the Table View.

Do It Yourself: Connect the Delete button in the application’s Details view to the `DetailsViewController` class with an Action called `deleteContact()`. Complete the Action with the code of Listing 28-7. Run the application, select a contact, and press the Delete button. The contact should disappear from the list.

The process to update a contact works the same way. We have to get a mutable copy of the contact selected by the user, modify the properties we want to update, and then create and execute the request. The following example implements the Action for the Last Call button of our application’s Details view. The code assigns a string to the `note` property with the current date. The purpose is to register the last time we made a phone call to the contact. Notice that we are able to modify the `note` property because the `CNContactNoteKey` key was included when the contact was retrieved (see Listing 28-3).

Listing 28-8: Updating a contact

```

@IBAction func updateContact(sender: UIButton) {
if contact != nil {
let updateContact = contact.mutableCopy() as! CNMutableContact
updateContact.note = "Last call \(NSDate())"
let request = CNSaveRequest()
request.updateContact(updateContact)
do {
let store = CNContactStore()
try store.executeSaveRequest(request)
} catch {
print("Error")
}
}
}

```

```

    }
}
navigationController?.popViewControllerAnimated(true)
}

```

Do It Yourself: Connect the Last Call button in the application’s Details view to the **DetailsViewController** class with an Action called **updateContact()**. Complete the Action with the code of Listing 28-8. Run the application, select a contact, and press the Last Call button. Go to the simulator’s Hardware menu and select Home, or press the Home button on your device. Open the Contacts app and select the contact you just updated to confirm that it was modified.

Adding new contacts is also a similar process, but we have included a special view for this purpose in our interface (see Figure 28-3, right). The view includes a Text Field to introduce a name and a button to save the new contact. The view controller for this view only needs an Outlet to access the value introduced in the Text Field and an Action for the Save button. We call it **AddContactViewController**.

Listing 28-9: Adding new contacts

```

import UIKit
import Contacts

class AddContactViewController: UIViewController {
    @IBOutlet weak var nameField: UITextField!
    @IBAction func addContact(sender: UIButton) {
        let text = nameField.text!
        if text != "" {
            let contact = CNMutableContact()
            contact.givenName = text
            let request = CNSaveRequest()
            request.addContact(contact, toContainerWithIdentifier: nil)
            do {
                let store = CNContactStore()
                try store.executeSaveRequest(request)
            } catch {
                print("Error")
            }
        }
        navigationController?.popViewControllerAnimated(true)
    }
}

```

```
}
```

The code of Listing 28-9 creates a new `CNMutableContact` object, assigns the value inserted in the Text Field to its `givenName` property, and creates the request to add the contact to the database. In this example we only introduce the name of the contact, but we can assign all the values we need.

Do It Yourself: Create a subclass of the `UIViewController` class, call it `AddContactViewController`, and assign it to the Add Contact view (Figure 28-3, right). Connect the Text Field to the `AddContactViewController` class with an Action called `addContact` and the Save button with an Outlet called `nameField`. Complete the class with the code of Listing 28-9. Run the application and press the bar button to open the Add Contact view. Insert a name and press the Save button. The new contact should appear in the list.

CNContactViewController

A contact may contain multiple values, including values of the same type associated with different labels, such as phone numbers or addresses. Creating an interface to show, update, or insert all these values can take time and demand dozens of lines of code. If we want to avoid this repetitive task, we can take advantage of the classes included in the ContactsUI framework. This is a complementary framework that includes two handful view controllers: the `CNContactPickerViewController` controller with a view that lets the user select contacts and values, and the `CNContactViewController` controller with a view that lets the user see and modify contacts one by one. The latter works as our Details view but includes all the values available for the contact and the tools necessary to edit and delete them. The class provides the following initializers to create the controller.

`CNContactViewController(forContact: contact)`—This initializer creates a Contact View Controller to present and edit the contact specified by the `contact` attribute. The attribute is a `CNContact` object.

`CNContactViewController(forNewContact: contact)`—This initializer creates a Contact View Controller to insert the values of a new contact. The `contact` attribute is a reference to the `CNMutableContact` object created to represent the new contact.

The class includes two properties to configure the view and a type method to get the keys descriptor we need to make sure we get all the properties the controller needs to process the contact.

allowsEditing—This property sets or returns a Boolean value that determines whether the view will allow the contact’s values to be edited or not.

allowsActions—This property sets or returns a Boolean value that determines if the view will allow the user to perform actions, like initiating a call or sending a message.

descriptorForRequiredKeys()—This type method returns a descriptor for all the keys required by the Contact View Controller.

The framework also defines a protocol for the `CNContactViewController` class called `CNContactViewControllerDelegate` that we need to dismiss the view controller when a new contact is created. The protocol defines the following method that we can use to process the new contact and dismiss the view.

contactViewController(viewController, didCompleteWithContact: contact)—This method is called on the delegate when the process is finished (the Cancel or Done button are pressed). The **contact** attribute is a reference to the contact created for the view.

Using the `CNContactViewController` controller we can simplify our interface because we only need to provide a way for the user to select a contact, the rest of the views are entirely replaced by the controller. For the following example we are going to implement the interface introduced in Figure 28-1 (a Table View in a Navigation Controller). The class that controls the Table View now has to create and present a new Contact View Controller every time a contact is selected.

Listing 28-10: Showing the values of the selected contact in a Contact View Controller

```
import UIKit
import Contacts
import ContactsUI

class ContactsViewController: UITableViewController, CNContactViewControllerDelegate {
    var contactList: [CNContact]!

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        let store = CNContactStore()
        store.requestAccessForEntityType(.Contacts, completionHandler: { (success, error) in
            if success {
```



```

let keys = CNContactViewController.descriptorForRequiredKeys()
let request = CNContactFetchRequest(keysToFetch: [keys])
do {
    self.contactList = []
    try store.enumerateContactsWithFetchRequest(request, usingBlock: { (contact, status) in
        self.contactList.append(contact)
    })
} catch {
    print("Error")
}
NSOperationQueue.mainQueue().addOperationWithBlock({
    self.tableView.reloadData()
})
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    if contactList != nil {
        return contactList.count
    }
    return 0
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("contactCell", forIndexPath: indexPath)
    let contact = contactList[indexPath.row]
    cell.textLabel?.text = "\(contact.givenName) \(contact.familyName)"
    return cell
}

override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let contact = contactList[indexPath.row]
    let controller = CNContactViewController(forContact: contact)
    navigationController?.pushViewController(controller, animated: true)
}
}

```

Listing 28-10 presents only two changes regarding the `ContactsViewController` class introduced in Listing 28-1. This time we use the descriptor returned by the `descriptorForRequiredKeys()` method to get a list of contact objects with the properties required by the Contact View Controller, and implement the `tableView(tableView, didSelectRowAtIndexPath: indexPath)` method to create and present the controller when a contact is selected by the user. The controller is created with the `CNContactViewController(forContact:`

`contact`) initializer and then added to the Navigation Controller stack with the `pushViewController()` method (see Chapter 13). Notice that we add the controller to the stack because we need it to be part of the Navigation Controller to provide the Back button the user needs to go back to the Table View, as shown in Figure 28-5.

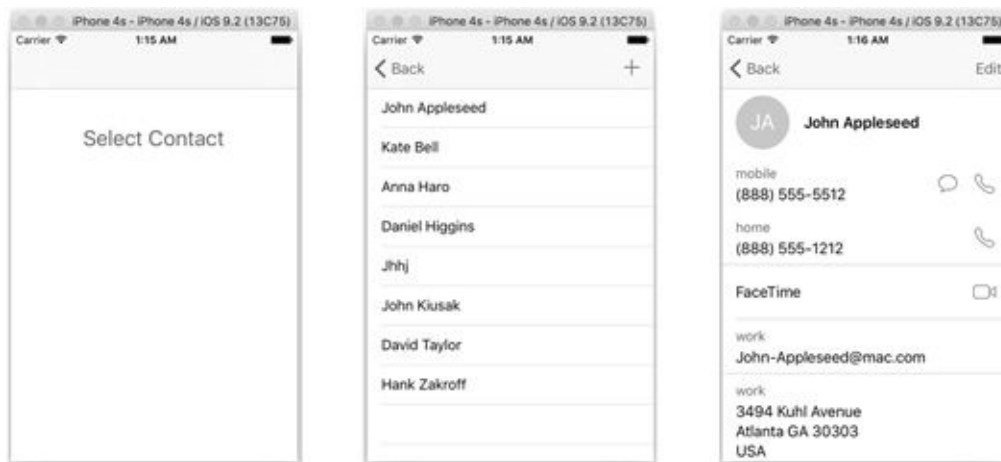


Figure 28-5: Contact displayed in a Contact View Controller

Do It Yourself: Adapt the previous project or create a new one with the interface of Figure 28-1. Update or create a new `ContactsViewController` class with the code of Listing 28-10. Run the application and select a contact. You should see something similar to Figure 28-5.

Adding a new contact with the Contact View Controller is easy. We just have to create a new `CNMutableContact` object and initialize the controller with the `CNContactViewController(forNewContact: contact)` initializer.

Listing 28-11: Adding new contacts with a Contact View Controller

```
@IBAction func addContact(sender: UIBarButtonItem) {  
    let contact = CNMutableContact()  
    let controller = CNContactViewController(forNewContact: contact)  
    controller.delegate = self  
    navigationController?.pushViewController(controller, animated: true)  
}  
  
func contactViewController(viewController: CNContactViewController, didCompleteWithContact contact:  
    CNContact?) {  
    navigationController?.popViewControllerAnimated(true)  
}
```

The Contact View Controller adds the contact to the database automatically when the

user presses the Done button, but we have to declare our view controller as its delegate and implement the `contactViewController(viewController: controller, didCompleteWithContact: contact)` method to remove the controller when the process is over.

Do It Yourself: Connect the bar button in the Table View's Navigation Bar to the `ContactsViewController` class with an Action called `addContact()`. Complete the method and the class with the code of Listing 28-11. Run the application, press the + button, insert some values, and press the Done button. You should see the new contact in the list.

CNContactPickerViewController

A device can store thousands of contacts that users may need to go through to find what they want. The ContactsUI framework includes the `CNContactPickerViewController` class to create a view that lists all the contacts available and lets the user select them along with their values. This is similar to having the Contacts app inside our application. The purpose of the `CNContactPickerViewController` class is to provide views where the user can select one or multiple contacts and also their values to be processed by the application. To send the selected values back to the application, the controller calls delegate methods defined in the `CNContactPickerDelegate` protocol. The following are the methods available.

contactPickerDidCancel(picker)—This method is called on the delegate when the user presses the Cancel button in the controller's view.

contactPicker(picker, didSelectContact: contact)—This method is called on the delegate when the user selects a contact. The **contact** attribute is a `CNContact` object that represents the selected contact.

contactPicker(picker, didSelectContactProperty: contactProperty)—This method is called on the delegate when the user selects a property. The **contactProperty** attribute is a `CNContactProperty` with the values of the selected property.

contactPicker(picker, didSelectContacts: contacts)—This method is called on the delegate when the user presses the Done button after selecting multiple contacts. The **contacts** attribute is an array of `CNContact` objects that represent the selected contacts.

contactPicker(picker, didSelectContactProperties: contactProperties)—This method is called on the delegate when the user presses the Done button after selecting multiple properties. The **contactProperties** attribute is an array of `CNContactProperty` objects with the values of the selected properties.

The process to incorporate this controller in our app is simple. We have to create and present the controller and implement the protocol methods to receive the values selected by the user. The following example assumes that we have an interface with a label connected with an Outlet called `nameLabel` and a button connected with an Action called `selectContact()`. When the button is pressed, the view controller is created and presented on the screen. After the user taps on a contact to select it, the view is closed and the protocol method is executed. In this method we read the contact's `givenName` and `familyName` properties and show their values on the screen.

Listing 28-12: Selecting contacts from a Contact Picker View Controller

```
import UIKit
import ContactsUI

class ViewController: UIViewController, CNContactPickerDelegate {
    @IBOutlet weak var nameLabel: UILabel!

    @IBAction func selectContact(sender: UIButton) {
        let controller = CNContactPickerViewController()
        controller.delegate = self
        presentViewController(controller, animated: true, completion: nil)
    }

    func contactPicker(picker: CNContactPickerViewController, didSelectContact contact: CNContact) {
        if contact.areKeysAvailable([CNContactGivenNameKey, CNContactFamilyNameKey]) {
            let name = "\(contact.givenName) \(contact.familyName)"
            nameLabel.text = name
        }
    }
}
```

The `CNContactPickerViewController` controller includes two views, one to list the contacts available and another to show the values of the selected contact. How the views are presented depends on the protocol methods implemented on the delegate. For example, if we don't implement any method, the controller shows the list of contacts and when a contact is selected it opens a second view to show its values, but if we implement the `contactPicker(picker, didSelectContact: contact)` method on the delegate, as we did in the example of Listing 28-12, when a contact is selected the controller calls the method and closes the view. The result is shown in Figure 28-6.

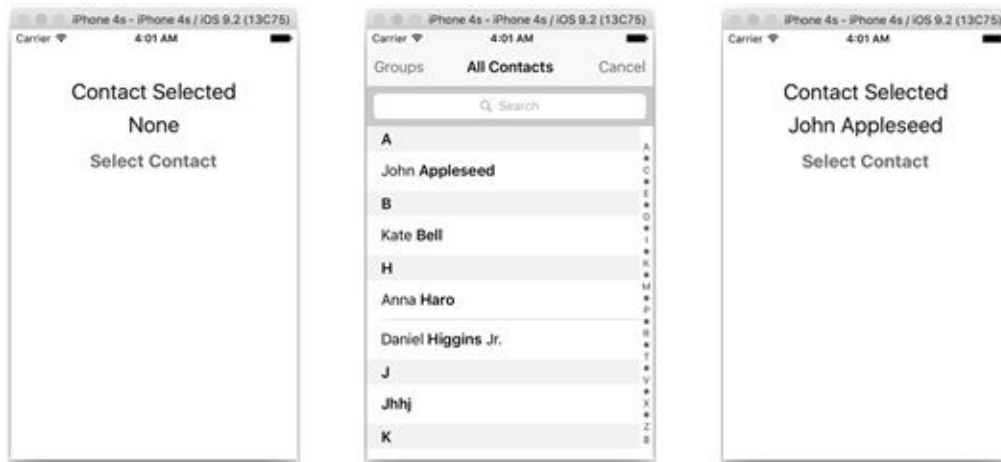


Figure 28-6: Contact selected from a Contact Picker View Controller

The `CNContactPickerViewController` class allows some level of configuration. One of the things we can do is to define predicates that determine what contacts and values are available for selection. The predicates must be created as normal `NSPredicate` objects and assigned to any of the following properties.

predicateForEnablingContact—This property sets or returns a predicate that determines which contacts are selectable.

predicateForSelectionOfContact—This property sets or returns a predicate that determines if the controller should return the selected contact or open the second view to show its values.

predicateForSelectionOfProperty—This property sets or returns a predicate that determines if the controller should return the selected property or perform the action by default (e.g., open the Phone app to make a call to the contact’s phone number).

The following example defines a predicate that allows users to select only contacts of which given names begin with “John”.

Listing 28-13: Assigning predicates to filter contacts

```
import UIKit
import ContactsUI

class ViewController: UIViewController, CNContactPickerDelegate {
    @IBOutlet weak var nameLabel: UILabel!

    @IBAction func selectContact(sender: UIButton) {
        let controller = CNContactPickerViewController()

        controller.predicateForEnablingContact = NSPredicate(format: "givenName = %@",
["John"])
```

```

    controller.delegate = self
    presentViewController(controller, animated: true, completion: nil)
}
func contactPicker(picker: CNContactPickerViewController, didSelectContact contact: CNContact) {
    if contact.areKeysAvailable([CNContactGivenNameKey, CNContactFamilyNameKey]) {
        let name = "\(contact.givenName) \(contact.familyName)"
        nameLabel.text = name
    }
}
}
}

```

Do It Yourself: Update the `ViewController` class of Listing 28-12 with the code of Listing 28-13. Run the application. All the contacts are displayed, but you should only be able to select those with the name “John”.

As we mentioned before, how the controller presents the views depends on the protocol methods implemented by the delegate. For example, if instead of the `contactPicker(picker, didSelectContact: contact)` method we implement the `contactPicker(picker, didSelectContactProperty: property)` method, when the user selects a contact, the controller opens a second view to let the user select a property.

Listing 28-14: Selecting properties

```

import UIKit
import ContactsUI

class ViewController: UIViewController, CNContactPickerDelegate {
    @IBOutlet weak var nameLabel: UILabel!
    @IBAction func selectContact(sender: UIButton) {
        let controller = CNContactPickerViewController()
        controller.predicateForSelectionOfProperty = NSPredicate(format: “key = %@”, argumentArray: [“phoneNumbers”])
        controller.delegate = self
        presentViewController(controller, animated: true, completion: nil)
    }
    func contactPicker(picker: CNContactPickerViewController, didSelectContactProperty contactProperty: CNContactProperty) {
        let phone = contactProperty.value as! CNPhoneNumber
        let number = phone.stringValue
        nameLabel.text = number
    }
}

```

Besides implementing the protocol method, we have to define a predicate that determines which properties are selectable. If the user selects a property that is not defined in the predicate, instead of returning the property's value, the controller performs the action by default (usually nothing).

Property values are returned as `CNContactProperty` objects. The class includes the following properties.

identifier—This property returns a string with the property's identifier.

contact—This property returns a `CNContact` object that represents the contact the property belongs to.

key—This property returns the key that represents the property.

value—This property returns the property's value. It is an `AnyObject` object that we have to cast to the original type.

label—This property returns a string with the label for labeled properties.

In Listing 28-14 we define a predicate to let the user select phone numbers. When a phone number is selected, the controller calls the protocol method to send the value to the delegate. In this method we read the `value` property of the `CNContactProperty` object received through the `contactProperty` attribute to get the property's value, cast it as a `CNPhoneNumber` object, and then assign to the label the value of its `stringValue` property (the phone number).

Do It Yourself: Update the `ViewController` class with the code of Listing 28-14. Run the application, select a contact, and then select a phone number. You should see the selected number printed on the screen.

We can also configure the types of properties that are going to be shown when a contact is selected. The `CNContactPickerViewController` class offers the following property for this purpose.

displayedPropertyKeys—This property sets or returns a set of keys that represent the properties we want to show to the user.

The following example improves the previous `ViewController` class to only show the properties the user is allowed to select (in this case, phone numbers).

Listing 28-15: Selecting phone numbers

```
import UIKit
import ContactsUI

class ViewController: UIViewController, CNContactPickerDelegate {
    @IBOutlet weak var nameLabel: UILabel!
    @IBAction func selectContact(sender: UIButton) {
        let controller = CNContactPickerViewController()
        controller.displayedPropertyKeys = [CNContactPhoneNumbersKey]
        controller.predicateForSelectionOfProperty = NSPredicate(format: "key = %@",
        ["phoneNumbers"])
        controller.delegate = self
        presentViewController(controller, animated: true, completion: nil)
    }
    func contactPicker(picker: CNContactPickerViewController, didSelectContactProperty contactProperty:
    CNContactProperty) {
        let phone = contactProperty.value as! CNPhoneNumber
        let number = phone.stringValue
        nameLabel.text = number
    }
}
```

Do It Yourself: Update the `viewController` class with the code of Listing 28-15. Run the application and select a contact. The view should show only phone numbers to select.

Chapter 29

Sensors

29.1 Core Motion

Apple devices come equipped with multiple sensors. The purpose of these sensors is to provide information about the situation of the device in the real world. Things like the physical location, the position, and orientation, or where the device is heading can all be determined from these sensors. Apple offers the Core Motion framework to have access to data related to motion and movement. The main class of this framework is **CMMotionManager**, which controls three important sensors called Accelerometer, Gyroscope, and Magnetometer. The Accelerometer is similar to a water level, it can detect movement in any direction, including sudden movements like when the user shakes the phone; the Gyroscope, on the other hand, is like a spinning wheel, good to measure rotation and determine orientation; and the Magnetometer is like a compass that measures magnetic fields, good to determine the orientation of the device and where it is heading based on the Earth's magnetic field.

The **CMMotionManager** class includes the following properties to detect if these sensors are available on the device.

accelerometerAvailable—This property returns a Boolean value that indicates whether the device has an Accelerometer or not.

gyroAvailable—This property returns a Boolean value that indicates whether the device has a Gyroscope or not.

magnetometerAvailable—This property returns a Boolean value that determines whether the device has a Magnetometer or not.

Motion Device

The **CMMotionManager** class also includes properties and methods to get information from the three sensors, but the data is not reliable. The sensors introduce interference and bias into their values that over time can produce inaccurate results. This is the reason why the class defines an additional system called *Motion Device* that fuses the information coming from all the three sensors to determine the attitude, rotation rate, and acceleration of the device with more precision and responsiveness. The following are the properties and methods defined in the **CMMotionManager** class to read and process the values returned by this system.

deviceMotionAvailable—This property returns a Boolean value that indicates whether the device includes the Device Motion system or not.

deviceMotionUpdateInterval—This property sets or returns the interval in seconds the system is going to use to provide information to our application. It is a value of type `NSTimeInterval` (a typealias of `Double`).

deviceMotionActive—This property returns a Boolean value that determines whether the Device Motion system is currently sending information to our application or not.

deviceMotion—This property returns a `CMDeviceMotion` object with the data collected from the sensors.

startDeviceMotionUpdatesToQueue(queue, withHandler: block)—This method asks the system to start collecting information. Every time the data is updated, it is sent to the method's handler. The **queue** attribute is a reference to the operation queue we want to use to process the data, and the **block** attribute is the closure that is going to be executed every time new data is received.

startDeviceMotionUpdatesUsingReferenceFrame(reference, toQueue: queue, withHandler: block)—This method asks the system to start collecting information. The values are calculated considering a point of reference and sent to the method's handler. The **reference** attribute is a value of type `CMAAttitudeReferenceFrame` that establishes a frame of reference to calculate the values, the **queue** attribute is a reference to the operation queue we want to use to process the data, and the **block** attribute is the closure that is going to be executed every time new data is received.

startDeviceMotionUpdates()—This method asks the system to start collecting information. After the method is called, we can read the current values directly from the `deviceMotion` property.

stopDeviceMotionUpdates()—This method asks the system to stop collecting information.

The data is returned by the Device Motion system in an object of the `CMDeviceMotion` class. The class offers the following properties to read the values.

gravity—This property returns a value that represents the acceleration produced by the gravity plus the acceleration the user gives to the device. It is a structure of type `CMAcceleration` with the properties `x`, `y`, and `z`.

userAcceleration—This property returns a value that represents the acceleration the user gives to the device. It is a structure of type `CMAcceleration` with the properties `x`, `y`, and `z`.

magneticField—This property returns a value that represents the magnetic field

detected by the sensor. It is a structure of type `CMCalibratedMagneticField` with the properties `field` and `accuracy`. The `field` property is a structure with the properties `x`, `y`, and `z` to store the value of every axis, and the `accuracy` property is a structure of type `CMMagneticFieldCalibrationAccuracy` with the properties `CMMagneticFieldCalibrationAccuracyUncalibrated`, `CMMagneticFieldCalibrationAccuracyLow`, `CMMagneticFieldCalibrationAccuracyMedium`, and `CMMagneticFieldCalibrationAccuracyHigh` that determine the degree of accuracy.

rotationRate—This property returns the rate of rotation around each axis (how fast the device is rotating). It is a structure of type `CMRotationRate` with the properties `x`, `y`, and `z`.

attitude—This property returns a `CMAAttitude` object with the orientation of the device relative to a frame of reference.

The three sensors, Accelerometer, Gyroscope, and Magnetometer, return values for each axis: `x`, `y`, and `z`. The following example uses three labels to show the values of every parameter on the screen.

Listing 29-1: Reading values from the sensors

```
import UIKit
import CoreMotion

class ViewController: UIViewController {
    @IBOutlet weak var valueX: UILabel!
    @IBOutlet weak var valueY: UILabel!
    @IBOutlet weak var valueZ: UILabel!
    var manager: CMMotionManager!

    override func viewDidLoad() {
        super.viewDidLoad()

        manager = CMMotionManager()
        if manager.deviceMotionAvailable {
            manager.deviceMotionUpdateInterval = 1.0 / 60.0
            let queue = NSOperationQueue.mainQueue()
            manager.startDeviceMotionUpdatesToQueue(queue, withHandler: { (data, error) in
                if let values = data {
                    self.valueX.text = String(values.gravity.x)
                    self.valueY.text = String(values.gravity.y)
                    self.valueZ.text = String(values.gravity.z)
                }
            })
        }
    }
}
```

```
}  
}
```

The first thing we need is an instance of the `CMMotionManager` class. We have to assign the instance to a property to keep the object alive as long as the view controller is opened. Using the `deviceMotionAvailable` property, we check if the Device Motion system is available and then initialize the process to read the values. Because we want the system to keep reporting the new values to our code, we set the interval as 1/60th of a second and call the `startDeviceMotionUpdatesToQueue()` method. This method requires a reference to a queue where the handler will be executed and a closure to handle the values. In this example we use the main queue, but it is recommended to designate a separate queue when the operations performed by the handler are significant to avoid freezing the interface (see `NSOperationQueue` in Chapter 20). The closure receives two values, `data` with the data gathered from the sensors and `error` to report errors in the process. The `data` attribute is a value of type `CMDeviceMotion`, so we can read any of the properties listed before. This time, we show the values of the `gravity` property because they are easy to track. Depending on how we position the device, we can see the numbers for each axis go from -1.0 to 1.0.

Do It Yourself: Create a new Single View Application project. Add three labels to the initial view. Connect the labels to the `ViewController` class with Outlets called `valueX`, `valueY`, and `valueZ`. Complete the class with the code of Listing 29-1 and run the application. You should see the values changing on the screen while the device is rotated. For better visualization, go to the app's settings (Figure 5-3, number 5) and deactivate the options Landscape Left and Landscape Right, so the application only works in portrait mode.

The Basics: The process of collecting data from the sensors consumes battery and resources. You have to make sure that the data is being collected only when it is necessary. The framework offers methods like `stopDeviceMotionUpdates()` to stop the process. We can call these methods when the data is no longer necessary and also in view controller methods like `viewWillDisappear()` to cancel the process when the view in which the data is required is not visible anymore.

There is an infinite number of applications for the values received from the `CMDeviceMotion` object. We can rotate graphics, adapt the interface, or provide different solutions according to the current state of the device. For instance, the following example reads the `userAcceleration` property to determine if the device was shaken.

Listing 29-2: *Changing the color of the view when the user shakes the device*

```
import UIKit
```

```

import CoreMotion

class ViewController: UIViewController {
    var manager: CMMotionManager!
    var control = false
    var delta = NSDate()
    override func viewDidLoad() {
        super.viewDidLoad()
        view.backgroundColor = UIColor.blueColor()
        manager = CMMotionManager()
        if manager.deviceMotionAvailable {
            manager.deviceMotionUpdateInterval = 1.0 / 60.0
            let queue = NSOperationQueue.mainQueue()
            manager.startDeviceMotionUpdatesToQueue(queue, withHandler: { (data, error) in
                if let values = data {
let today = NSDate()
let seconds = today.timeIntervalSinceDate(self.delta)
let acc = values.userAcceleration.x
if seconds > 0.5 && acc > 2.0 {
self.delta = today
if self.control {
self.view.backgroundColor = UIColor.blueColor()
self.control = false
    } else {
self.view.backgroundColor = UIColor.redColor()
self.control = true
    }
    }
})
}
            })
        }
    }
}

```

In Listing 29-2 we read two values, the acceleration given by the user to the device and the seconds since the last time we checked the value. This is to avoid multiple readings of the same movement. If the acceleration is greater than 2.0 and it's been more than 0.5 seconds since the last reading, we change the main view's background color. The result is that every time the device is shaken from left to right or right to left, the view changes color (we only read the x axis to get the lateral movement).

Do It Yourself: Create a new Single View Application project. Complete the **ViewController** class with the code of Listing 29-2. Run the application and shake the device. Every time the device is shaken, you should see the screen change from red to blue and vice versa.

One of the values we can get from the **CMDDeviceMotion** object is the attitude. The attitude determines how the device sits in a tridimensional space. The attitude values are provided in three mathematical representations: Euler Angles, Rotation Matrix, and Quaternion. The most frequently used are the Euler Angles. These angles are expressed through the properties **pitch**, **roll**, and **yaw**, and determine the orientation of the device in relation to the three axes x, y, and z, respectively. The following example rotates a box on the screen according to the values of **pitch** and **roll**.

Listing 29-3: Rotating a view according to the attitude values

```
import UIKit
import CoreMotion

class ViewController: UIViewController {
    var manager: CMMotionManager!
    var square: UIView!

    override func viewDidLoad() {
        super.viewDidLoad()
        view.setNeedsLayout()
        view.layoutIfNeeded()
        let width = view.frame.size.width
        let height = view.frame.size.height
        square = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
        square.center = CGPoint(x: width / 2, y: height / 2)
        square.backgroundColor = UIColor.redColor()
        view.addSubview(square)

        manager = CMMotionManager()
        if manager.deviceMotionAvailable {
            manager.deviceMotionUpdateInterval = 1.0 / 60.0
            let queue = NSOperationQueue.mainQueue()
            manager.startDeviceMotionUpdatesToQueue(queue, withHandler: { (data, error) in
                if let values = data {
                    let radians = atan(values.attitude.pitch / values.attitude.roll)
                    var matrix = CATransform3DIdentity
                    matrix = CATransform3DRotate(matrix, CGFloat(radians), 0, 0, 1)
                    self.square.layer.transform = matrix
                }
            })
        }
    }
}
```



```

    }
    })
    }
}
}

```

The code of Listing 29-3 creates and adds a small view at the center of the screen to represent a box. When Device Motion sends the values to the handler, we calculate the new angle and rotate the view's layer using a 3D transformation matrix (see Chapter 24). As a result, the box keeps the same orientation while the device is being rotated.

The values of `pitch`, `roll`, and `yaw` correspond to the rotation angles around each axis. This means that if we want to know how many degrees the device was rotated from left to right, in theory we just have to read the value corresponding to that angle (`yaw`). But unfortunately Euler Angles can't determine absolute positions and require some algebra. In our example we calculate the tangent of the angles expressed by the `pitch` and `roll` properties to get the real angle of the box on the two dimensions of the screen.

Do It Yourself: Create a new Single View Application project. Complete the `ViewController` class with the code of Listing 29-3. Run the application and rotate the device. You should see a red square on the screen rotating in the opposite direction. For better visualization, go to the app's settings (Figure 5-3, number 5) and deactivate the options Landscape Left and Landscape Right, so the application only works in portrait mode.

Motion Activity

Combining the information coming from the sensors we can determine what is going on with the device in the real world. Believe it or not, we can determine things like whether the user is walking or running. But calculating what the user is doing based on this data can be tricky. Fortunately, Core Motion was designed to help us with these tasks. One of the systems included by the framework for this purpose is called Motion Activity. This system can estimate what the user is doing (e.g., walking, cycling, etc.) and let us know in real time. Motion Activity is managed by an object of the `CMMotionActivityManager` class, which includes the following methods.

isActivityAvailable()—This method returns a Boolean value that indicates if Motion Activity is available on the device or not (the system is available since the iPhone 5).

startActivityUpdatesToQueue(queue, withHandler: block)—This method

orders the system to start collecting information. The **queue** attribute is a reference to the queue where we want the handler to be executed, and the **block** attribute is the closure that is going to process the values returned. The closure receives an object of type **CMMotionActivity** containing the values produced by the system.

queryActivityStartingFromDate(start, toDate: end, toQueue: queue, withHandler: block)—This method collects historical activity from the system. The **start** and **end** attributes are **NSDate** objects that indicate the period of time we want to get data from, the **queue** attribute is a reference to the queue where we want the handler to be executed, and the **block** attribute is the closure that is going to process the values returned. The closure receives an array of **CMMotionActivity** objects that represent the updates occurred during the specified timeframe. We can collect information up to 7 days prior to the current date.

stopActivityUpdates()—This method orders the system to stop collecting information.

The data returned by Motion Activity is represented by an object of the **CMMotionActivity** class. The class includes the following properties to retrieve the values.

stationary—This property returns a Boolean value that determines if the device is stationary.

walking—This property returns a Boolean value that determines if the user is walking.

running—This property returns a Boolean value that determines if the user is running.

automotive—This property returns a Boolean value that determines if the user is in a moving vehicle.

cycling—This property returns a Boolean value that determines if the user is riding a bicycle.

unknown—This property returns a Boolean value that determines if the current state is unknown.

startDate—This property returns an **NSDate** object with the time when the update of the motion occurred.

confidence—This property returns a value that determines the confidence the system has on the results. It is an enumeration of type **CMMotionActivityConfidence** with the values **Low**, **Medium**, and **High**.

Like previous systems, we have to create an instance of the manager and then call a method to begin collecting data from the system.

Listing 29-4: Detecting the type of motion

```
import UIKit
import CoreMotion

class ViewController: UIViewController {
    @IBOutlet weak var activityLabel: UILabel!
    var manager: CMMotionActivityManager!

    override func viewDidLoad() {
        super.viewDidLoad()
        if CMMotionActivityManager.isActivityAvailable() {
            manager = CMMotionActivityManager()
            let queue = NSOperationQueue.mainQueue()
            manager.startActivityUpdatesToQueue(queue, withHandler: { (data) in
                if let values = data {
                    if values.walking {
                        self.activityLabel.text = "Walking"
                    } else if values.running {
                        self.activityLabel.text = "Running"
                    } else if values.cycling {
                        self.activityLabel.text = "Cycling"
                    } else {
                        self.activityLabel.text = "No Activity"
                    }
                }
            })
        } else {
            print("Motion Activity is not available")
        }
    }
}
```

The code of Listing 29-4 detects if Motion Activity is available on the device and then initializes the manager to get the data from the system. When a new update is received, we check the values of the `CMMotionActivity` object's properties and change a label in the interface to reflect the current activity (if `walking` is `true`, it means that the user is walking, if `running` is `true` it means that the user is running, and so on).

Do It Yourself: Create a new Single View Application project. Add a label to the main view and connect it to the `ViewController` class with an Outlet called `activityLabel`.

Complete the class with the code of Listing 29-4 and run the application. If Motion Activity is available on your device, you should see changes in the label according to the activity you are performing at the moment.

Pedometer

Another system provided by Core Motion to help us understand what's going on with the device is the Pedometer. The main function of the Pedometer is to quantify the amount of activity performed by the user, like the steps the user walked or how many floors ascended. The manager for the Pedometer is provided by the **CMPedometer** class. The following are the methods to check for availability and initiate the collection of data.

isStepCountingAvailable()—This method returns a Boolean value that indicates if the device can count steps.

isDistanceAvailable()—This method returns a Boolean value that indicates if the device can calculate the distance traveled.

isFloorCountingAvailable()—This method returns a Boolean value that indicates if the device can count floors.

isPaceAvailable()—This method returns a Boolean value that indicates if the device can determine the pace of the user in seconds per meter.

isCadenceAvailable()—This method returns a Boolean value that determines if the device can measure the rate at which the user takes steps.

startPedometerUpdatesFromDate(start, withHandler: block)—This method asks the system to start collecting information. The **start** attribute is an **NSDate** object with the time we want to start from, and the **block** attribute is the closure that is going to process the values returned. The closure receives an object of type **CMPedometerData** containing the values produced by the system.

queryPedometerDataFromDate(start, toDate: end, withHandler: block)—This method collects historical data from the system. The **start** and **end** attributes are **NSDate** objects that indicate the period of time we want to get data from, and the **block** attribute is the closure that is going to process the values returned. The closure receives an object of type **CMPedometerData** containing the values produced by the system. We can collect information up to 7 days prior to the current date.

stopPedometerUpdates()—This method asks the system to stop collecting information.

The data returned by the Pedometer is represented by an object of the `CMPedometerData` class. The class includes the following properties to retrieve the values.

numberOfSteps—This property returns an `NSNumber` object with the number of steps taken by the user.

distance—This property returns an `NSNumber` object with the distance traveled by the user in meters.

currentPace—This property returns an `NSNumber` object with the current pace of the user in seconds per meter.

currentCadence—This property returns an `NSNumber` object with the rate at which the steps are taken, measured in steps per second.

floorsAscended—This property returns an `NSNumber` object with the number of floors the user has ascended.

floorsDescended—This property returns an `NSNumber` object with the number of floors the user has descended.

startDate—This property returns an `NSDate` object with the time in which the collection of data started.

endDate—This property returns an `NSDate` object with the time in which the collection of data ended.

Again, to collect data from the sensor we have to initialize the manager and call its methods. The following example counts the steps walked by the user from the moment the view is loaded.

Listing 29-5: Counting the steps

```
import UIKit
import CoreMotion

class ViewController: UIViewController {
    @IBOutlet weak var stepsLabel: UILabel!
    var pedometer: CMPedometer!
    override func viewDidLoad() {
        super.viewDidLoad()
        if CMPedometer.isStepCountingAvailable() {
            pedometer = CMPedometer()
            pedometer.startPedometerUpdatesFromDate(NSDate(), withHandler: { (data, error) in
                if let values = data {
```

```

self.stepsLabel.text = "\(values.numberOfSteps)"
}
})
} else {
print("The system can't count your steps")
}
}
}

```

Do It Yourself: Create a new Single View Application project. Add a label to the main view and connect it to the **ViewController** class with an Outlet called **stepsLabel**. Complete the class with the code of Listing 29-5 and run the application. If the Pedometer is available on your device, you should see on the screen the number of steps taken.

Altimeter

Recent devices include a new sensor called *Altimeter* that can measure the altitude at which the device is located. The system is managed by the **CMAltimeter** class, which includes three methods to detect if the sensor is available and start collecting data from it.

isRelativeAltitudeAvailable()—This method returns a Boolean value that indicates if the device can detect changes in altitude.

startRelativeAltitudeUpdatesToQueue(queue, withHandler: block)—This method asks the system to start collecting information. The **queue** attribute is a reference to the queue where we want the handler to be executed, and the **block** attribute is the closure that is going to process the values returned. The closure receives an object of type **CMAltitudeData** containing the values produced by the system.

stopRelativeAltitudeUpdates()—This method asks the system to stop collecting information.

The class used to communicate the data is called **CMAltitudeData** and includes only two properties.

relativeAltitude—This property returns an **NSNumber** object with the change in altitude since the last time it was reported.

pressure—This property returns an `NSNumber` object with the pressure measured by the Altimeter in kilopascals.

The `relativeAltitude` property doesn't return an absolute altitude but the changes over time from the moment the system begins collecting data. In the following example we use a property called `totalAltitude` to keep track of the changes and show the total altitude on the screen.

Listing 29-6: Calculating the altitude

```
import UIKit
import CoreMotion

class ViewController: UIViewController {
    @IBOutlet weak var altitudeLabel: UILabel!
    var altimeter: CMAltimeter!
    var totalAltitude = 0.0
    override func viewDidLoad() {
        super.viewDidLoad()
        if CMAltimeter.isRelativeAltitudeAvailable() {
            altimeter = CMAltimeter()
            let queue = NSOperationQueue.mainQueue()
            altimeter.startRelativeAltitudeUpdatesToQueue(queue, withHandler: { (data, error) in
                if let values = data {
                    self.totalAltitude += values.relativeAltitude.doubleValue
                    self.altitudeLabel.text = String(self.totalAltitude)
                }
            })
        } else {
            print("The device can't measure altitude")
        }
    }
}
```

Do It Yourself: Create a new Single View Application project. Add a label to the main view and connect it to the `ViewController` class with an Outlet called `altitudeLabel`. Complete the class with the code of Listing 29-6 and run the application. If the Altimeter is available on your device, you should see on the screen the change in altitude since the application is launched.

29.2 Core Location

Another framework developed to help applications determine the situation of the device in the real world is Core Location. This framework doesn't work with its own sensors, but it turns communication devices into sensors to determine the device's location on Earth (including latitude, longitude, and altitude). The system uses cellular towers, the GPS system, and Wi-Fi networks to determine the location. Measuring the signals from cellular towers, the system is able to determine the area in which the device is located. GPS is implemented to provide more accuracy, with data in the range of 3 to 5 meters, but in urban areas the signal is lost and Wi-Fi signals are used to improve precision. With the combination of these three systems, an iOS device can determine its own location with a very low margin of error.

Core Location works in a similar way as the systems studied before. The manager class for this system is called `CLLocationManager`, and includes the following properties and methods.

desiredAccuracy—This property sets or returns a value that determines the accuracy in meters we want the system to achieve. It is of type `CLLocationAccuracy` (a typealias of `Double`). The framework provides some predefined values with the following constants: `kCLLocationAccuracyBestForNavigation`, `kCLLocationAccuracyBest` (set by default), `kCLLocationAccuracyNearestTenMeters`, `kCLLocationAccuracyHundredMeters`, `kCLLocationAccuracyKilometer`, and `kCLLocationAccuracyThreeKilometers`.

distanceFilter—This property sets or returns a value that determines the minimum distance in meters we want the device to move before reporting a new location. It is a value of type `CLLocationDistance` (a typealias of `Double`). The framework defines the `kCLDistanceFilterNone` constant to specify that we want to be notified of every movement (this value is set by default).

activityType—This property sets or returns a value that helps the system understand when the data is required. If the application needs to register the locations while the user is walking, that data becomes irrelevant when the user gets into a car. The property is an enumeration of type `CLActivityType` with the values `Other` (by default), `AutomotiveNavigation`, `Fitness`, and `OtherNavigation`.

allowsBackgroundLocationUpdates—This property sets or returns a Boolean value that determines if the system should get location updates while the app is in the background (the user opened another app).

startUpdatingLocation()—This method orders the system to start collecting information.

stopUpdatingLocation()—This method orders the system to stop collecting information.

requestLocation()—This method requests a one-time delivery of the device's current location. It takes care of starting and finishing the process of collecting the data and makes sure that the most accurate data is returned.

Core Location works with a delegate to report the location data to the application. The framework includes the `CLLocationManagerDelegate` protocol for this purpose. The following are its most frequently used methods.

locationManager(manager, didUpdateLocations: locations)—This method is called on the delegate when a new location is determined. The **locations** attribute is an array with the locations obtained by the system.

locationManager(manager, didFailWithError: error)—This method is called on the delegate when an error occurs.

The location data is stored in objects of the `CLLocation` class. The class includes an initializer to create a `CLLocation` object from raw data and properties to retrieve the information.

CLLocation(latitude: latitude, longitude: longitude)—This initializer creates a `CLLocation` object with the location determined by its attributes. The **latitude** and **longitude** attributes are values of type `CLLocationDegrees` (a typealias of `Double`).

coordinate—This property returns the coordinates of the location. It is a structure of type `CLLocationCoordinate2D` with the properties **latitude** and **longitude**.

horizontalAccuracy—This property returns a value that determines the accuracy of the location's latitude and longitude values. It is of type `CLLocationAccuracy` (a typealias of `Double`).

verticalAccuracy—This property returns a value that determines the accuracy of the location's altitude. It is of type `CLLocationAccuracy` (a typealias of `Double`).

altitude—This property returns the altitude of the location in meters. It is of type `CLLocationDistance` (a typealias of `Double`).

The `CLLocation` class also includes a practical method to calculate the distance between

one `CLLocation` object and another.

distanceFromLocation(location)—This method returns a value that determines the distance between the location in the `CLLocation` object and the location determined by the **location** attribute. The attribute is another object of type `CLLocation` with the location we want to compare. The value returned is of type `CLLocationDistance` (a typealias of `Double`).

Authorization

Before even trying to get the current location we have to ask permission to the user. There are two types of authorization we may request. We can ask permission to get updates only while the app is active (the app is being used by the user at the moment), or all the time (even when the app moves to the background). The `CLLocationManager` class offers the following methods to process the authorization.

requestWhenInUseAuthorization()—This method asks authorization to get the location while the app is in use.

requestAlwaysAuthorization()—This method asks authorization to get the location when the app is active or in the background (when it was closed or the user switched to another app).

authorizationStatus()—This method returns the current authorization status. The value returned is an enumeration of type `CLAuthorizationStatus` with the values `NotDetermined`, `Restricted`, `Denied`, `AuthorizedAlways`, and `AuthorizedWhenInUse`.

The `CLLocationManagerDelegate` protocol also defines a method to get the authorization status every time it is changed.

locationManager(manager, didChangeAuthorizationStatus: status)—This method is called on the delegate when the user changes the level of authorization. The **status** attribute is an enumeration of type `CLAuthorizationStatus` that determines the new authorization status. The possible values are `NotDetermined`, `Restricted`, `Denied`, `AuthorizedAlways`, and `AuthorizedWhenInUse`.

There is one more step we have to take to complete the configuration. Unlike other systems, Core Location requires us to specify two keys in the application's info.plist file to

provide the text that the system is going to show in the Alert View when it asks the user to confirm if our application is allowed to access the device's location. We already explained how to add keys and values to this configuration file in Chapter 27 (see Figure 27-4). The framework defines two keys, one for each type of authorization: `NSLocationAlwaysUsageDescription` and `NSLocationWhenInUseUsageDescription`. We have to add these keys to the `info.plist` with a string that explains why our app wants the user's permission, as shown in Figure 29-1.

Key	Type	Value
Information Property List	Dictionary	(18 items)
NSLocationAlwaysUsageDescription	String	We need this information to show your location
NSLocationWhenInUseUsageDescription	String	We need this information to show your location
Localization native development region	String	en
Executable file	String	\$(EXECUTABLE_NAME)

Figure 29-1: Authorization keys added to the `info.plist` file

Getting the Location

With a location manager we can get the current location to know where the user is at the moment or multiple successive locations to determine where the user is going. In any case, we have to create an instance of the `CLLocationManager` class, configure the manager, and implement at least two protocol methods to get the values returned by the system and report errors.

Listing 29-7: Getting user's location

```
import UIKit
import CoreLocation

class ViewController: UIViewController, CLLocationManagerDelegate {
    @IBOutlet weak var latitudeLabel: UILabel!
    @IBOutlet weak var longitudeLabel: UILabel!
    var manager: CLLocationManager!

    override func viewDidLoad() {
        super.viewDidLoad()

        manager = CLLocationManager()
        manager.delegate = self
        manager.distanceFilter = kCLHeadingFilterNone
        manager.desiredAccuracy = kCLLocationAccuracyBest
        manager.requestWhenInUseAuthorization()
        manager.startUpdatingLocation()
    }
}
```

```

func locationManager(manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
    if let location = locations.last {
        latitudeLabel.text = String(location.coordinate.latitude)
        longitudeLabel.text = String(location.coordinate.longitude)
    }
}

func locationManager(manager: CLLocationManager, didFailWithError error: NSError) {
    print("Error")
}
}

```

The code of Listing 29-7 checks location changes over time and shows the values on the screen. The example assumes that we have added two labels to the main view to show the latitude and longitude. We create the `CLLocationManager` object first, assign the view controller as its delegate, configure the manager to get all the locations determined by the system with maximum accuracy, request authorization, and finally initiate the process to collect the data with the `startUpdatingLocation()` method. The authorization is requested with the `requestWhenInUseAuthorization()` because we only need to know the current location when the app is opened and the user is watching the values on the screen. When new locations are determined, the system calls the `locationManager(manager, didUpdateLocations: locations)` method on the delegate with an array of all the locations found. The `CLLocation` objects received by the method represent all the values found for the current location. We can use the last one obtained or calculate an average for a better approximation. In our example, we just read the last one and get the values of the latitude and longitude from its `coordinate` property.

Do It Yourself: Create a new Single View Application project. Add two labels to the main view and connect them to the `ViewController` with Outlets called `latitudeLabel` and `longitudeLabel`. Complete the class with the code of Listing 29-7. Add the `NSLocationAlwaysUsageDescription` and `NSLocationWhenInUseUsageDescription` keys to the `info.plist` file, as shown in Figure 29-1. Run the application. After allowing the app to get your location, you should see the values of your longitude and latitude on the screen.

When we monitor location changes as we did in the previous example, we have to remember to stop the process with the `stopUpdatingLocation()` method if the information is no longer necessary. If all we want is to get the current location, instead of the `startUpdatingLocation()` method we can call the `requestLocation()` method. This method starts and stops the process automatically and gets an average of all the locations found to provide the most accurate result.

Listing 29-8: Getting only the user’s current location

```
import UIKit
import CoreLocation

class ViewController: UIViewController, CLLocationManagerDelegate {
    @IBOutlet weak var latitudeLabel: UILabel!
    @IBOutlet weak var longitudeLabel: UILabel!
    var manager: CLLocationManager!

    override func viewDidLoad() {
        super.viewDidLoad()
        manager = CLLocationManager()
        manager.delegate = self
        manager.distanceFilter = kCLHeadingFilterNone
        manager.desiredAccuracy = kCLLocationAccuracyBest
        manager.requestWhenInUseAuthorization()
        manager.requestLocation()
    }

    func locationManager(manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
        if let location = locations.last {
            latitudeLabel.text = String(format: "%.2f", location.coordinate.latitude)
            longitudeLabel.text = String(format: "%.2f", location.coordinate.longitude)
        }
    }

    func locationManager(manager: CLLocationManager, didFailWithError error: NSError) {
        print("Error")
    }
}
```

In Listing 29-8 we follow the same procedure to initialize and configure the location manager, but we ask the system to only get the current location and then stop. We format the values to have only two decimal digits and show them on the screen (see **String** structures in Chapter 3).

Background Location

So far we have read the location while the app is opened and active (it is being shown on the screen), but when the user presses the Home button or switches to another app, our app moves to the background and the system stops updating the location. There are situations in which this is not the expected behaviour. For example, we may have an application that

needs to perform certain tasks while the user is running to be able to show results later when it is reopened. iOS allows certain activities to be performed in the background, but we must declare it explicitly from the Capabilities panel in the app's settings (see Chapter 23, Figure 23-1). The panel offers an option called Background Modes. When we turn this option on, Xcode shows a list of processes that we may allow to perform while the application is in the background. For Core Location, we have to select the option called Location updates, as shown in Figure 29-2.

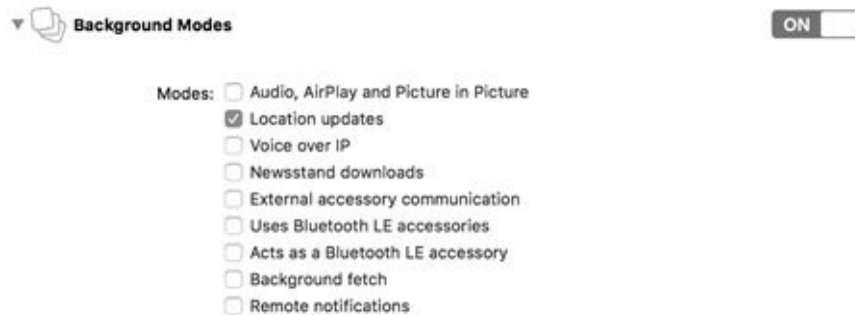


Figure 29-2: Authorization keys added to the info.plist file

Because this setting is permanent, we also have to set the `allowsBackgroundLocationUpdates` property to tell the manager when our application needs to get locations while in the background. The following example calculates the distance between two locations with the `distanceFromLocation()` method to show a message to the user every time he or she achieves the goal of walking 10 meters.

Listing 29-9: Getting notifications while the app is in the background

```
import UIKit
import CoreLocation

class ViewController: UIViewController, CLLocationManagerDelegate {
    @IBOutlet weak var distanceLabel: UILabel!
    var manager: CLLocationManager!

    var initial: CLLocation!
    var lastDistance = 0.0

    override func viewDidLoad() {
        super.viewDidLoad()
        manager = CLLocationManager()
        manager.delegate = self
        manager.distanceFilter = kCLHeadingFilterNone
        manager.desiredAccuracy = kCLLocationAccuracyBest
        manager.allowsBackgroundLocationUpdates = true
        manager.requestAlwaysAuthorization()
```

```

    manager.startUpdatingLocation()
}

func locationManager(manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
    if let location = locations.last {
        if initial != nil {
            let distance = location.distanceFromLocation(initial)
            let last = lastDistance + 10
            if distance > last {
                showNotification()
                lastDistance += 10
            }
            distanceLabel.text = String(format: "%.2f", distance)
        } else {
            initial = location
        }
    }
}

func locationManager(manager: CLLocationManager, didFailWithError error: NSError) {
    print("Error")
}

func showNotification() {
    let notification = UILocalNotification()
    notification.fireDate = NSDate()
    notification.timeZone = NSTimeZone.defaultTimeZone()
    notification.alertBody = "New Achievement!"
    let app = UIApplication.sharedApplication()
    app.scheduleLocalNotification(notification)
}
}

```

In Listing 29-9 we define two properties called **initial** and **lastDistance** to store the initial location and the next goal the user must achieve. Every time a new location is received, we calculate the distance between this and the initial location and compare it to the value of **lastDistance**. If the goal is achieved, we call our **showNotification()** method to post a notification and increment the value of **lastDistance** to set the new goal.

Notifications also require the user's authorization. As we did in Chapter 18 (see Listing 18-8), we can ask for permission to post notifications as soon as the app is launched by calling the **registerUserNotificationSettings()** method of the **UIApplication** object in the app's delegate.

Listing 29-10: Asking for permission to show notifications

```

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let app = UIApplication.sharedApplication()
        let setNotifications = UIUserNotificationSettings(forTypes: [.Alert], categories: nil)
        app.registerUserNotificationSettings(setNotifications)
        return true
    }
}

```

Do It Yourself: Create a new Single View Application project. Add a label to the initial view to show the distance on the screen and connect it to the **ViewController** class with an Outlet called **distanceLabel**. Go to the app's settings (Figure 5-3, number 5), select the Capabilities panel (Figure 6-97), turn on the Background Modes option, and select Location updates from the list, as shown in Figure 29-2. Add the keys to the info.plist file (Figure 29-1). Update the **AppDelegate** class with the code of Listing 29-10 and complete the **ViewController** class with the code of Listing 29-9. Run the application on your device, press the Home button and take a walk. You should see a notification on the screen every time you walk 10 meters.

Region

Another useful feature of Core Location is the possibility to establish a region and then be notified when the user enters or exits that region. The framework includes a generic class called **CLRegion** and subclasses that define specific types of regions. The most useful is the **CLCircularRegion** class, used to define a circular region. The following is the initializer for this class.

CLCircularRegion(center: **center, radius: **radius**, identifier: **identifier**)**

—This initializer creates a **CLCircularRegion** object that defines the location and boundaries of a circular geographic region. The **center** attribute is a structure of type **CLLocationCoordinate2D** that determines the latitude and longitude of the center of the region, the **radius** attribute is a **CLLocationDistance** value (a typealias of **Double**) that determines the radius of the region, and the **identifier** attribute is a string that identifies the region.

The center of the region is determined by a structure of type `CLLocationCoordinate2D`. This structure contains two properties to store the latitude and longitude. The following is the structure's initializer.

`CLLocationCoordinate2D(latitude: latitude, longitude: longitude)`—This initializer creates a `CLLocationCoordinate2D` structure with the location specified by its attributes. The **`latitude`** and **`longitude`** attributes are values of type `CLLocationDegrees` (a typealias of `Double`).

The `CLLocationManager` class includes the following methods to monitor a region.

`startMonitoringForRegion(region)`—This method asks the system to monitor the region specified by the **`region`** attribute. The attribute is an object of the `CLRegion` class or any of its subclasses.

`stopMonitoringForRegion(region)`— This method asks the system to stop monitoring the region specified by the **`region`** attribute. The attribute is an object of the `CLRegion` class or any of its subclasses.

When the manager determines, based on the current location, that the user has entered or exited a region, it calls the following protocol methods on its delegate to let our app know what happened.

`locationManager(manager, didEnterRegion: region)`—This method is called on the delegate when the system determines that the user enters a region.

`locationManager(manager, didExitRegion: region)`—This method is called on the delegate when the system determines that the user exited a region.

The following example stores the initial location and posts a notification when the device is 20 meters away from that location. The application assumes that the initial location is the user's home and therefore he or she has to be reminded to take the keys when leaving the place.

Listing 29-11: Monitoring a region

```
import UIKit
import CoreLocation
```

```

class ViewController: UIViewController, CLLocationManagerDelegate {
    var manager: CLLocationManager!
    var initial: CLLocation!

    override func viewDidLoad() {
        super.viewDidLoad()

        manager = CLLocationManager()
        manager.delegate = self
        manager.distanceFilter = kCLHeadingFilterNone
        manager.desiredAccuracy = kCLLocationAccuracyBest
        manager.allowsBackgroundLocationUpdates = true
        manager.requestAlwaysAuthorization()
        manager.startUpdatingLocation()
    }

    func locationManager(manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
        if let location = locations.last {
            if initial == nil {
                initial = location

                let coordinates = CLLocationCoordinate2D(latitude: location.coordinate.latitude, longitude:
location.coordinate.longitude)
                let region = CLCircularRegion(center: coordinates, radius: 20, identifier: "Home")
                manager.startMonitoringForRegion(region)
            }
        }
    }

    func locationManager(manager: CLLocationManager, didExitRegion region: CLRegion) {
        if region.identifier == "Home" {
            let notification = UILocalNotification()
            notification.fireDate = NSDate()
            notification.timeZone = NSTimeZone.defaultTimeZone()
            notification.alertBody = "Don't forget the keys!"
            let app = UIApplication.sharedApplication()
            app.scheduleLocalNotification(notification)
        }
    }
}

```

The code of Listing 29-11 creates a region as soon as the current location is determined. The region is identified with the string “Home” so it can be set apart from other regions we may need to monitor later. When the delegate method is called by the system to tell the application that the user has exited the region, we check if the region is Home and then

post a notification to remind the user to take the keys.

Do It Yourself: Replace the `ViewController` class of Listing 29-9 created for the previous project by the class of Listing 29-11. Remove the label in the main view (we are not showing the distance anymore) and run the application. Walk away from your current location. You should see a notification posted when you are 20 meters or more away from the initial location.

Geocoding

Geocoding is the name of a process in which location coordinates (latitude and longitude) are turned into an address or vice-versa. Users can't understand or interpret the values 40.7637825011971 and -73.9731328627541, but they can easily figure out how to get to 767 5th Ave, New York, NY 10153, United States, which is the address of one of the Apple stores in New York, located at those coordinates. Core Location offers the `CLGeocoder` class to convert these values. The following methods are provided by the class to process addresses one way or another.

`reverseGeocodeLocation(location, completionHandler: block)`—This method creates a string with the address that corresponds to the location specified by the **`location`** attribute. The **`location`** attribute is a `CLLocation` object with the coordinates of the location, and the **`block`** attribute is a closure that is executed when the operation is over. The closure receives two values: an array with `CLPlacemark` objects that represent the addresses found and an `NSError` object to report for errors.

`geocodeAddressString(address, completionHandler: block)`—This method determines the location of the address specified by the **`address`** attribute. The **`address`** attribute is a string with the human-readable address (preferably with values separated by comma), and the **`block`** attribute is a closure that is executed when the operation is over. The closure receives two values: an array with `CLPlacemark` objects that represent the locations found and an `NSError` object to report for errors.

`geocodeAddressString(address, inRegion: region, completionHandler: block)`—This method determines the location of the address specified by the **`address`** attribute. The **`address`** attribute is a string with the human-readable address (preferably with values separated by comma), the **`region`** attribute is a `CLRegion` object that gives a hint to the system of where the address is located, and the **`block`** attribute is a closure that is executed when the operation is over. The closure receives two values: an array with `CLPlacemark` objects that represent the locations found and an `NSError` object to report for errors.

No matter if the conversion is from address to location or location to address, the results are always returned as **CLPlacemark** objects. The following are the most frequently used properties of this class.

location—This property returns a **CLLocation** object with the location’s latitude and longitude.

name—This property returns a string with the name of the placemark (usually the name of the street).

country—This property returns a string with the name of the country.

postalCode—This property returns a string with the postal code.

administrativeArea—This property returns a string with the state or province.

subAdministrativeArea—This property returns a string with additional information of the administrative area.

locality—This property returns a string with the name of the city.

subLocality—This property returns a string with information of the city.

thoroughfare—This property returns a string with the street address.

subThoroughfare—This property returns a string with additional information of the street.

areasOfInterest—This property returns an array of strings with names of areas of interest close to the placemark (e.g., museums, parks, etc.).

To get a string with the address, we just have to generate a **CLLocation** object with the right coordinates and call the **reverseGeocodeLocation()** method. For example, we can get the address of the current location.

Listing 29-12: Getting the address of the current location

```
import UIKit
import CoreLocation

class ViewController: UIViewController, CLLocationManagerDelegate {
    @IBOutlet weak var streetLabel: UILabel!
    @IBOutlet weak var cityLabel: UILabel!
    @IBOutlet weak var stateLabel: UILabel!
    @IBOutlet weak var countryLabel: UILabel!
```

```

var manager: CLLocationManager!
override func viewDidLoad() {
    super.viewDidLoad()
    manager = CLLocationManager()
    manager.delegate = self
    manager.requestWhenInUseAuthorization()
    manager.requestLocation()
}

func locationManager(manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
    if let location = locations.last {
        let geocoding = CLGeocoder()
        geocoding.reverseGeocodeLocation(location, completionHandler: { (placemarks, error) in
            if let place = placemarks?.first {
                let street = "\(place.thoroughfare!) \(place.subThoroughfare!)"
                self.streetLabel.text = street
                self.cityLabel.text = place.locality
                self.stateLabel.text = place.administrativeArea
                self.countryLabel.text = place.country
            }
        })
    }
}

func locationManager(manager: CLLocationManager, didFailWithError error: NSError) {
    print("Error")
}
}

```

The code of Listing 29-12 assumes that we have four labels in the initial view to show the values of the street, city, state, and country of a location. When the view is loaded, the code asks permission to get the user’s location and calls the `reverseGeocodeLocation()` method to get the address. The method generates a `CLPlacemark` object with the address found and sends it to the closure. The value is an array that in this case contains only one item, so we extract the first item and assign its values to the labels.

Do It Yourself: Create a new Single View Application project. Add four labels to the initial view and connect them to the `ViewController` class with the Outlets `streetLabel`, `cityLabel`, `stateLabel`, and `countryLabel`. Complete the `ViewController` class with the code of Listing 29-12. Remember to add the keys to the `info.plist` file as we did for previous projects in this chapter. Run the application. You should see your address on the screen.

The opposite process, converting an address to latitude and longitude values, works the same way. We have to generate a string with the address and then call the `geocodeAddressString()` method.

Listing 29-13: Getting the location of an address

```
import UIKit
import CoreLocation

class ViewController: UIViewController {
    @IBOutlet weak var latitudeLabel: UILabel!
    @IBOutlet weak var longitudeLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        let address = "767 5th Ave, New York, NY 10153, United States"
        let geocoding = CLGeocoder()
        geocoding.geocodeAddressString(address, completionHandler: { (places, error) in
            if places != nil {
                let place = places?.first
                if let coordinates = place?.location?.coordinate {
                    self.latitudeLabel.text = String(coordinates.latitude)
                    self.longitudeLabel.text = String(coordinates.longitude)
                }
            }
        })
    }
}
```

If the address is not specific enough, the `geocodeAddressString()` method can generate multiple locations. In Listing 29-13 we just take the first location found and show its coordinate values on the screen, but a professional application will probably list the values and let the user select the appropriate one for the case.

Do It Yourself: Create a new Single View Application project or modify the previous one. Add two labels to the initial view and connect them to the `ViewController` class with the Outlets `latitudeLabel` and `longitudeLabel`. Complete the class with the code of Listing 29-13. Remember to add the keys to the `info.plist` file as we did for previous projects in this chapter. Run the application. You should see the latitude and longitude values corresponding to the address on the screen.

IMPORTANT: Apple offers an additional service that combines Core Motion with

Core Location to determine indoor locations. The topic is beyond the scope of this book. For more information visit our website and follow the links for this chapter.

The Basics: Because addresses are retrieved from a database online, the device has to be connected to the Internet for the process to work. The system can identify addresses in different formats. If you provide the address from code, as we did in Listing 29-13, write the values in order of relevance and separated by commas.

Chapter 30

MapKit

30.1 Map View

The geocoding feature of Core Location introduced in Chapter 29 allows us to get a more user-friendly description of a location, but sometimes this is not enough. Often, users need to visualize their location or the places where they want to go on a map to position themselves in the world. Apple offers the MapKit framework to create and configure maps that users can interact with to look for places and get information. Maps are presented on the screen with a view created from a subclass of `UIView` called `MKMapView`. The view takes care of loading the map, managing user interaction and displaying custom content generated by our application. The view may be added to the interface from an option in the Object Library.

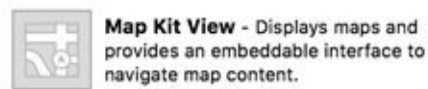


Figure 30-1: Map Kit View option in the Object Library

Like a regular view, the Map View can be constrained to the sides of the main view or other elements. Figure 30-2 shows what this view looks like on a simple interface.

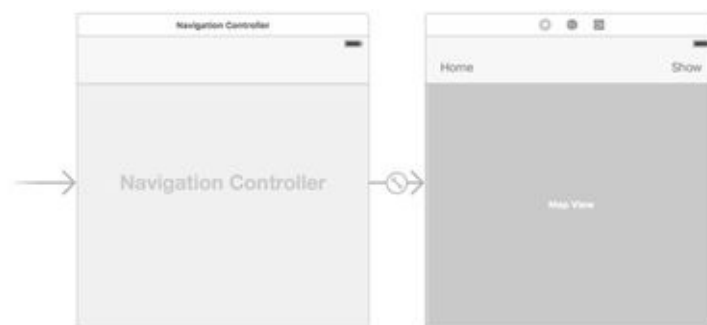


Figure 30-2: Interface with a Map View

By default, the map shows the entire world, but it can only work after the MapKit framework is imported in the view controller, as shown next.

Listing 30-1: Setting up the Map View

```
import UIKit
import MapKit

class ViewController: UIViewController {
    @IBOutlet weak var mapView: MKMapView!
```

```

override func viewDidLoad() {
    super.viewDidLoad()
}
}

```



Figure 30-3: Map View with the configuration by default

Do It Yourself: Create a new Single View Application project. Embed the initial view in a Navigation Controller. Add a Map Kit View to the initial view, pin it to the sides, and connect it to the **ViewController** class with an Outlet called **mapView**. Add two bar buttons to the Navigation Bar called Home and Show, as shown in Figure 30-2. We will use these buttons later to select locations. Complete the **ViewController** class with the code of Listing 30-1 and run the application. You should see something similar to Figure 30-3.

Configuring the Map

The Map View offers different map styles and can also let the user zoom in and out, pan, or rotate the map to find a location. The **MKMapView** class includes some properties to configure these features.

mapType—This property sets or returns a value that determines the style of the map. It is an enumeration of type **MKMapType** with the values **Standard** (by default), **Satellite**, **Hybrid**, **SatelliteFlyover**, and **HybridFlyover**.

zoomEnabled—This property sets or returns a Boolean value that determines if the user can zoom the map (**true** by default).

scrollEnabled—This property sets or returns a Boolean value that determines if the

user can scroll the map (**true** by default).

rotateEnabled—This property sets or returns a Boolean value that determines if the user can rotate the map (**true** by default).

We can define the characteristics of the map any time we want. For example, we can change the map style and determine what the user will be able to do with the map before it is shown on the screen.

Listing 30-2: Configuring the map

```
import UIKit
import MapKit

class ViewController: UIViewController {
    @IBOutlet weak var mapView: MKMapView!

    override func viewDidLoad() {
        super.viewDidLoad()
        mapView.mapType = .Satellite
        mapView.rotateEnabled = false
    }
}
```

Users can zoom in and out to find a specific place on the map, but we can also do it from code. The **MKMapView** class offers the following properties and methods to get the values of the area currently displayed on the screen or set a new one.

region—This property returns an **MKCoordinateRegion** structure with values that determine the area currently displayed on the screen.

centerCoordinate—This property returns a **CLLocationCoordinate2D** structure with the coordinates at the center of the visible area.

setRegion(region, animated: animated)—This method sets the map’s visible region. The **region** attribute is a structure of type **MKCoordinateRegion**, which contains a **CLLocationCoordinate2D** value that determines the region’s center coordinates, and a value of type **MKCoordinateSpan** that determines the region’s size in degrees. The **animated** attribute is a Boolean value that determines if the process is going to be animated.

setCenterCoordinate(coordinates, animated: animated)—This method sets the region’s coordinates. The **coordinates** attribute is a value of type

`CLLocationCoordinate2D` with the latitude and longitude, and the **animated** attribute is a Boolean value that determines if the process is going to be animated.

The `MKCoordinateRegion` structure required to set the region is created from a `CLLocationCoordinate2D` structure that contains the latitude and longitude (see Chapter 29), and an `MKCoordinateSpan` value that contains the size of the region in degrees. Because the degrees of an area can be difficult to calculate, the framework defines a function that lets us set the size of the region in meters.

MKCoordinateRegionMakeWithDistance(coordinates, latitude, longitude)—This function returns an `MKCoordinateRegion` structure with the values determined by its attributes. The **coordinates** attribute is an `CLLocationCoordinate2D` value with the region’s coordinates, and the **latitude** and **longitude** are values of type `CLLocationDistance` (a typealias of `Double`) that determine the vertical and horizontal size of the region in meters.

The following example implements this function to show an area of 2000 meters around the location of one of the Apple Stores in New York.

Listing 30-3: Displaying a specific region

```
import UIKit
import MapKit

class ViewController: UIViewController {
    @IBOutlet weak var mapView: MKMapView!

    override func viewDidLoad() {
        super.viewDidLoad()

        let location = CLLocationCoordinate(latitude: 40.7637825011971, longitude: -73.9731328627541)
        let region = MKCoordinateRegionMakeWithDistance(location.coordinate, 2000, 2000)
        mapView.setRegion(region, animated: false)
    }
}
```

The code of Listing 30-3 creates a `CLLocation` object with the coordinates of the Apple Store, defines a region around that area, and finally sets the map’s visible region with the `setRegion()` method. This time, instead of the entire world, the Map View shows the area of New York where the store is located.

Do It Yourself: Update the `ViewController` class with the code of Listing 30-3. Run the application. You should see an area of New York City with the Apple store in the center.

Annotations

The previous example establishes the visible area of the map and sets the center to be the location of the Apple Store. Because this is a relevant location, the map shows an icon with the name of the store, but this is not always the case. Most locations don't show any reference at all and the user has to guess where the location actually is. To add our own pins or graphics to the map, we have to work with annotations. Annotations are marks on the map that provide additional information of a particular location. They are associated with a view that can show an image to represent the annotation, such as a pin, and a subview, called *Callout*, that opens when the user taps on the view to show additional information, such as an image or a title. Figure 30-4 illustrates how an annotation might look like on the map.



Figure 30-4: Annotation on the map

What the framework describes as annotations are actually objects that define the basic aspects of an annotation, like the coordinates in which the annotation view will appear and the title and subtitle the callout is going to display. The MapKit framework includes the `MKAnnotation` protocol that we have to adopt to create these objects. The protocol defines the following properties.

coordinate—This property sets or returns a `CLLocationCoordinate2D` structure that determines the annotation's latitude and longitude.

title—This property sets or returns a string with the title to show on the callout bubble.

subtitle—This property sets or returns a string with the subtitle to show on the callout bubble.

To create our annotations, we have to define a class that adopts this protocol and implements at least its three properties.

Listing 30-4: Defining an annotation

```
import MapKit

class MyAnnotation: NSObject, MKAnnotation {
    var coordinate: CLLocationCoordinate2D
    var title: String?
    var subtitle: String?
    init(coordinate: CLLocationCoordinate2D) {
        self.coordinate = coordinate
    }
}
```

Annotations are created from our custom class and then added to the map. The **MKMapView** class offers the following properties and methods to add, remove, and manage annotations.

annotations—This property returns an array of **MKAnnotation** objects that represent the annotations already added to the map.

addAnnotation(annotation)—This method adds an annotation to the map. The **annotation** attribute is an object of a class that adopts the **MKAnnotation** protocol.

addAnnotations(annotations)—This method adds multiple annotations to the map. The **annotations** attribute is an array of objects of a class that adopts the **MKAnnotation** protocol.

removeAnnotation(annotation)—This method removes an annotation. The **annotation** attribute is a reference to the object that represents the annotation.

removeAnnotations(annotations)—This method removes multiple annotations. The **annotations** attribute is an array of objects that represent the annotations we want to remove.

showAnnotations(annotations, animated: animated)—This method sets a region that includes the annotations specified by the **annotations** attribute. The **annotations** attribute is an array of objects that represent the annotations we want to include in the region, and the **animated** attribute is a Boolean value that indicates if the process should be animated.

selectAnnotation(annotation, animated: animated)—This method selects an annotation (it shows the subview, as if the user had tapped on the annotation view). The **annotation** attribute is an object that represents the annotation we want to select, and the **animated** attribute is a Boolean values that indicates if the process should be animated.

deselectAnnotation(annotation, animated: animated)—This method deselects an annotation (it hides the subview). The **annotation** attribute is an object that represents the annotation we want to deselect, and the **animated** attribute is a Boolean values that indicates if the process should be animated.

The following code expands the **viewDidLoad()** method from the previous example to create an annotation from our **MyAnnotation** class and adds it to the map.

Listing 30-5: Adding an annotation

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    let location = CLLocation(latitude: 40.7637825011971, longitude: -73.9731328627541)  
    let region = MKCoordinateRegionMakeWithDistance(location.coordinate, 2000, 2000)  
    mapView.setRegion(region, animated: false)  
  
    let annotation = MyAnnotation(coordinate: location.coordinate)  
    annotation.title = "Apple Store"  
    mapView.addAnnotation(annotation)  
}
```

By default, the Map View generates a view to show the annotation on the map. The view contains an image that represents a pin pointing to the annotation’s location and a subview to display the callout bubble when the user taps on the pin. Figure 30-5 shows what this annotation looks like.

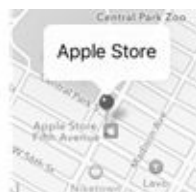


Figure 30-5: Annotation with a view by default

Do It Yourself: Add to your project a new Swift file called **MyAnnotation.swift** and copy the code of Listing 30-4 inside. Update the **ViewController** class with the code of

Listing 30-5. Run the application and tap on the pin pointing to the Apple Store. You should see something similar to Figure 30-5.

IMPORTANT: The MapKit framework includes the **MKPointAnnotation** class to create basic annotations. If your annotations only require the values of the **coordinate**, **title**, and **subtitle** properties, you can create them from this class instead of defining your own.

An annotation object determines where the annotation is located, but the annotation view is the element responsible for displaying on the map the graphic pointing to the annotation's coordinates (usually in the form of a pin or small icon). By default, the Map View provides its own view with a small pin and a simple callout bubble (Figure 30-5), but we can define our own with the **MKAnnotationView** class or a subclass of it. The class includes the following initializer and properties to create and configure the view.

MKAnnotationView(annotation: **annotation, reuseIdentifier: **identifier**)**

—This initializer creates an annotation view for the annotation specified by the **annotation** attribute. The **annotation** attribute is a reference to the annotation object we want to associate with the view, and the **identifier** attribute is a string the Map View needs to be able to reuse the view to display multiple annotations.

image—This property sets or returns the image for the view. It is an object of type **UIImage**.

enabled—This property sets or returns a Boolean value that determines if the view can be selected.

leftCalloutAccessoryView—This property sets or returns the view displayed on the left side of the callout bubble.

rightCalloutAccessoryView—This property sets or returns the view displayed on the right side of the callout bubble.

Annotations are added to the map and then the Map View checks which ones are inside the visible area and requests the views to display. The Map View works like a Table View; when it needs a view, it calls a delegate method to get it. The method is defined in the **MKMapViewDelegate** protocol. To provide a custom view for each annotation, we have to adopt this protocol and implement the following method.

mapView(mapView, viewForAnnotation: **annotation)**—This method is called

on the delegate when the Map View needs a view to show an annotation on the map. The **annotation** attribute is a reference to the object that represents the annotation the map has to show.

Like cells, annotation views are reusable. We have to create the view inside the delegate method, assign an identifier to it, and then use the same view again for other annotations. The **MKMapView** class includes the following method to get a reusable view.

dequeueReusableAnnotationViewWithIdentifier(identifier)—This method returns an **MKAnnotationView** object with the view identified by the **identifier** attribute. The attribute is the same string declared when the view was created.

The **ViewController** class in the following example adopts the **MKMapViewDelegate** protocol and implements the delegate method to create a custom view for our annotation.

Listing 30-6: Adding a custom annotation view

```
import UIKit
import MapKit

class ViewController: UIViewController, MKMapViewDelegate {
    @IBOutlet weak var mapView: MKMapView!
    override func viewDidLoad() {
        super.viewDidLoad()

        let location = CLLocation(latitude: 40.7637825011971, longitude: -73.9731328627541)
        let region = MKCoordinateRegionMakeWithDistance(location.coordinate, 2000, 2000)
        mapView.setRegion(region, animated: false)

        let annotation = MyAnnotation(coordinate: location.coordinate)
        annotation.title = "Apple Store"
        mapView.addAnnotation(annotation)
        mapView.delegate = self
    }

    func mapView(mapView: MKMapView, viewForAnnotation annotation: MKAnnotation) ->
MKAnnotationView? {
        if let temp = annotation as? MyAnnotation {
            var aView = mapView.dequeueReusableAnnotationViewWithIdentifier("Pins")
            if aView == nil {
                aView = MKAnnotationView(annotation: temp, reuseIdentifier: "Pins")
                aView?.image = UIImage(named: "iconmap")
            }
        }
    }
```

```

aView?.canShowCallout = true
} else {
aView?.annotation = annotation
}
return aView
}
return nil
}
}

```

The code of Listing 30-6 sets the region we want to show, adds an annotation to the map for the coordinates where the Apple Store is located, and finally designates the **ViewController** class as the Map View’s delegate. The protocol method is implemented next to provide a custom view for this annotation. When the Map View is already showing the selected area, it detects that there is an annotation inside that area and calls the `mapView(mapView, viewForAnnotation: annotation)` method to get its view. This method checks whether the annotation is one of our custom annotations and creates the view for it. Before creating a new view, we first check whether a view with the “Pins” identifier already exists. If not, we create a new view for the annotation with the `iconmap.png` image. Notice that the `dequeueReusableAnnotationViewWithIdentifier()` method returns a view that was previously used for another annotation, so we have to assign the new annotation to it before returning it. Figure 30-6 shows how the custom view looks like.

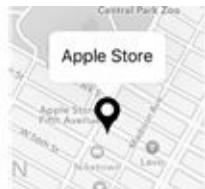


Figure 30-6: Custom annotation view

Do It Yourself: Download the `iconmap.png` image from our website and add it to your project’s Assets Catalog. Update the **ViewController** class from the previous example with the code of Listing 30-6. Run the application and tap on the icon. You should see something similar to Figure 30-6.

The annotation view is responsible for the subview that presents the annotation’s callout bubble. Besides the title and subtitle provided by the annotation, this subview can also include two small views or controls on the sides. Because these views usually contain information specific for each annotation, we have to define the data required to create the views in the annotation object. This may be images, text, decoration views, etc. For example, we might add a property to the **MyAnnotation** class to include a thumbnail that

helps the user identify the location.

Listing 30-7: Adding custom data to an annotation

```
import MapKit
```

```
class MyAnnotation: NSObject, MKAnnotation {  
    var coordinate: CLLocationCoordinate2D  
    var title: String?  
    var subtitle: String?  
    var picture: UIImage?  
  
    init(coordinate: CLLocationCoordinate2D) {  
        self.coordinate = coordinate  
    }  
}
```

Now, besides the title and subtitle, we can also define a thumbnail for each annotation and then display that image in the callout bubble. Depending on which side we want the thumbnail to appear, we have to assign it to the `leftCalloutAccessoryView` or the `rightCalloutAccessoryView` properties.

Listing 30-8: Configuring the callout bubble

```
import UIKit
```

```
import MapKit
```

```
class ViewController: UIViewController, MKMapViewDelegate {  
    @IBOutlet weak var mapView: MKMapView!  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        let location = CLLocationCoordinate(latitude: 40.7637825011971, longitude: -73.9731328627541)  
        let region = MKCoordinateRegionMakeWithDistance(location.coordinate, 2000, 2000)  
        mapView.setRegion(region, animated: false)  
        let annotation = MyAnnotation(coordinate: location.coordinate)  
        annotation.title = "Apple Store"  
        annotation.picture = UIImage(named: "appstore")  
        mapView.addAnnotation(annotation)  
  
        mapView.delegate = self  
    }  
    func mapView(mapView: MKMapView, viewForAnnotation annotation: MKAnnotation) -> MKAnnotationView? {  
        if let temp = annotation as? MyAnnotation {
```

```

var aView = mapView.dequeueReusableAnnotationViewWithIdentifier("Pins")
if aView == nil {
    aView = MKAnnotationView(annotation: temp, reuseIdentifier: "Pins")
    aView?.image = UIImage(named: "iconmap")
    aView?.canShowCallout = true

    let leftImage = UIImageView(image: temp.picture)
    aView?.leftCalloutAccessoryView = leftImage
} else {
    aView?.annotation = annotation
}
return aView
}
return nil
}
}

```

The delegate method of Listing 30-8 adds a small image to the callout's left side called `appstore.png` with a picture of the Apple Store. The result is shown in Figure 30-7.

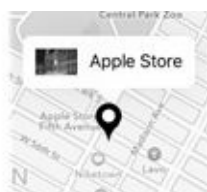


Figure 30-7: *Callout with a thumbnail*

Do It Yourself: Update the `MyAnnotation` class with the code of Listing 30-7 and the `ViewController` class with the code of Listing 30-8. Run the application and tap on the pin to open the callout bubble.

If we don't need to provide a custom view and image, we can implement the view assigned by default to the annotations (Figure 30-5). This view is created from the `MKPinAnnotationView` class, which includes an image of a pin that may adopt three different colors. The class includes the following properties to configure the view.

animatesDrop—This property sets or returns a Boolean value that determines if the insertion of the pin is going to be animated.

pinTintColor—This property sets or returns a `UIColor` object that determines the color

of the pin. The `MKPinAnnotationView` class includes the type methods `redPinColor()`, `greenPinColor()` and `purplePinColor()` to return the three standard colors: red, green, and purple.

Listing 30-9: Using a Pin Annotation View

```
func mapView(mapView: MKMapView, viewForAnnotation annotation: MKAnnotation) -> MKAnnotationView? {
    if let temp = annotation as? MyAnnotation {
        var aView = mapView.dequeueReusableAnnotationViewWithIdentifier("Pins") as? MKPinAnnotationView
        if aView == nil {
            aView = MKPinAnnotationView(annotation: annotation, reuseIdentifier: "Pins")
            aView?.pinTintColor = MKPinAnnotationView.redPinColor()
            aView?.canShowCallout = true
            aView?.animatesDrop = true

            let leftImage = UIImageView(image: temp.picture)
            aView?.leftCalloutAccessoryView = leftImage
        } else {
            aView?.annotation = annotation
        }
        return aView
    }
    return nil
}
```

To take advantage of the Pin Annotation View we just have to replace the `MKAnnotationView` instance by an instance of the `MKPinAnnotationView` class. Notice that the `dequeueReusableAnnotationViewWithIdentifier()` method always returns an `MKAnnotationView` object that we have to cast as our class (in this case, `MKPinAnnotationView`).

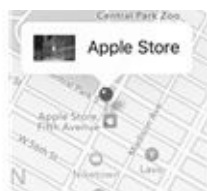


Figure 30-8: Pin Annotation View

Do It Yourself: Replace the `mapView(mapView, viewForAnnotation: annotation)` method by the method of Listing 30-9. Run the application and tap on the icon. You should see something similar to Figure 30-8.

User Location

Displaying the user location on a map is easy with Map Views. We don't even have to determine the location ourselves, as we did in the examples of Chapter 29, we just have to set the properties provided by the **MKMapView** class and the Map View takes care of everything. The following are the properties available.

showsUserLocation—This property sets or returns a Boolean value that determines if we want the Map View to determine the user location.

userLocationVisible—This property returns a Boolean value that determines if the user's location is currently visible on the map.

userLocation—This property returns an object of the **MKUserLocation** class with the user's location.

To get the Map View to determine the user's current location and show it on the map, we just have to set the **showsUserLocation** property to **true**, but if we want the map to show the area around the location, we have to set the map's visible region with the **setRegion()** method, as we did before. The following example configures the Map View to get the user's location and then defines an Action for the Home button on the Navigation Bar to set the region around it.

Listing 30-10: Getting the user location

```
import UIKit
import MapKit

class ViewController: UIViewController, MKMapViewDelegate {
    @IBOutlet weak var mapView: MKMapView!
    var manager: CLLocationManager!

    override func viewDidLoad() {
        super.viewDidLoad()
        manager = CLLocationManager()
        manager.requestWhenInUseAuthorization()

        mapView.showsUserLocation = true
        mapView.delegate = self
    }

    @IBAction func showHome(sender: UIBarButtonItem) {
        let location = mapView.userLocation
```

```

let region = MKCoordinateRegionMakeWithDistance(location.coordinate, 2000, 2000)
mapView.setRegion(region, animated: true)
}
}

```

No matter the process we use to determine the user's location, we always have to ask permission to do it. Once we have requested authorization, we can set the `showsUserLocation` property to ask the Map View to get the current location and display it on the map. The location is shown on the map represented by a view with a small icon, as shown in Figure 30-9.

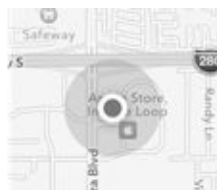


Figure 30-9: *User's location on the map*

Do It Yourself: Connect the Home button with an Action called `showHome()`. Update the `ViewController` class with the code of Listing 30-10. Remember to add the `NSLocationAlwaysUsageDescription` and `NSLocationWhenInUseUsageDescription` keys to the `info.plist` file, as we did for the examples of Chapter 29 (see Figure 29-1). Run the application and press the Home button to see your location on the map.

The location is returned as an object of the `MKUserLocation` class. This class defines a special type of annotation to represent the user's location. As any other annotation class, it includes the `title`, `subtitle`, and `coordinate` properties that we can read to get the annotation's values, but also the following specific properties.

location—This property returns a `CLLocation` object with the user's location. If the location was not determined yet, the property returns `nil`.

updating—This property returns a Boolean value that indicates if the Map View is currently in the process of determining the user's location.

heading—This property returns a value that determines where the user is heading. It is an object of the `CLHeading` class, which, among others, includes the `magneticHeading` and `trueHeading` properties to return a value in degrees that determines the cardinal point to which the user is heading (0 = north, 90 = east, 180 = south and 270 = west).

Using these properties we can respond to the location's current value, as we did in Listing 30-10 or configure the annotation to show information that helps the user identify the location.

Listing 30-11: Defining a title for the user's location

```
override fun viewDidLoad() {  
    super.viewDidLoad()  
    manager = CLLocationManager()  
    manager.requestWhenInUseAuthorization()  
  
    mapView.showsUserLocation = true  
    mapView.userLocation.title = "You are here"  
    mapView.delegate = self  
}
```

The Map View constantly updates the user's location. To help us keep track of these changes over time and also detect errors, the **MKMapViewDelegate** protocol defines the following methods.

mapView(mapView, didUpdateUserLocation: **userLocation)**—This method is called on the delegate every time a new location is determined. The **userLocation** attribute is an object of the **MKUserLocation** class with the user's current location.

mapView(mapView, didFailToLocateUserWithError: **error)**—This method is called on the delegate when the Map View can't determine the user's location. The **error** attribute is an **NSError** value with information of the error produced.

When the Map View detects a new location, it calls the **mapView(mapView, didUpdateUserLocation: location)** method to report the result. Here we can displace the center of the region to the new location or perform any other task we need.

Listing 30-12: Keeping track of the user's location

```
func mapView(mapView: MKMapView, didUpdateUserLocation userLocation: MKUserLocation) {  
    mapView.setCenterCoordinate(userLocation.coordinate, animated: true)  
}
```

Search

With the Geocoding service provided by Core Location we can turn an address into a location and create an annotation to show it on the map, as we have seen in Chapter 29, but the MapKit framework incorporates its own service to not only translate addresses into locations but also find places of interest. The service is called *Local Search* and can take a freeform query string and return an array with the results. The query is created from the `MKLocalSearchRequest` class, which includes the following properties for configuration.

naturalLanguageQuery—This property sets or returns a string with the term or address we want to search.

region—This property sets or returns an `MKCoordinateRegion` structure that determines the region in which we want to perform the search.

To perform the search, the framework defines the `MKLocalSearch` class. The class includes the following initializer and method.

MKLocalSearch(request: request)—This initializer creates an `MKLocalSearch` object to perform a search request. The **request** attribute is an `MKLocalSearchRequest` object.

startWithCompletionHandler(block)—This method performs the search and sends the results to the closure specified by the **block** attribute. The **block** attribute is a closure that receives two values: an `MKLocalSearchResponse` object with the results and an `NSError` object with the errors found.

The search returns an `MKLocalSearchResponse` object that contains the following properties.

mapItems—This property returns an array of `MKMapItem` objects that represent the results returned by the search.

boundingRegion—This property returns an `MKCoordinateRegion` structure that determines the region occupied by the results returned by the search.

The Local Search service was designed to find all the places that match the query. The framework defines the `MKMapItem` to represent each place. This class includes multiple properties to return the data from the place, including the following.

name—This property sets or returns a string with the place’s name.

phoneNumber—This property sets or returns a string with the place’s phone number.

url—This property sets or returns an `NSURL` value with the place’s URL.

placemark—This property sets or returns an `MKPlacemark` object (a subclass of the `CLPlacemark` studied in Chapter 29) with additional information of the place.

There are different ways an app can perform a search and display the places returned. For our example, we have decided to connect the Show button introduced in the interface of Figure 30-2 to an Action that sets the region around the user’s location, finds places that sell pizza, and creates the annotation to show them on the map.

Listing 30-13: Searching for pizza places

```
import UIKit
import MapKit

class ViewController: UIViewController, MKMapViewDelegate {
    @IBOutlet weak var mapView: MKMapView!
    var manager: CLLocationManager!

    override func viewDidLoad() {
        super.viewDidLoad()
        manager = CLLocationManager()
        manager.requestWhenInUseAuthorization()
        mapView.showsUserLocation = true
        mapView.userLocation.title = “You are here”
        mapView.delegate = self
    }

    @IBAction func showLocations(sender: UIBarButtonItem) {
        let location = mapView.userLocation
        let region = MKCoordinateRegionMakeWithDistance(location.coordinate, 2000, 2000)
        mapView.setRegion(region, animated: true)

        setAnnotations()
    }

    func setAnnotations() {
        let request = MKLocalSearchRequest()
        request.naturalLanguageQuery = “Pizza”
        request.region = mapView.region
        let search = MKLocalSearch(request: request)
        search.startWithCompletionHandler({ (results, error) in
```

```

if let items = results?.mapItems {
    self.mapView.removeAnnotations(self.mapView.annotations)
    for item in items {
        if let coordinates = item.placemark.location?.coordinate {
            let annotation = MKPointAnnotation()
            annotation.coordinate = coordinates
            annotation.title = item.name
            annotation.subtitle = item.phoneNumber
            self.mapView.addAnnotation(annotation)
        }
    }
}
}
}
}
}
}
}
}
}

```

The code of Listing 30-13 defines the `setAnnotations()` method to search for places associated with the term “Pizza” and set the annotations for each location found. When the `startWithCompletionHandler()` method returns the results, we remove the current annotations, get the coordinates of the location for each place, and set the new annotations with the place’s name and phone number as the callout’s title and subtitle. Every time the user taps on an annotation view, these values are displayed on the screen, as shown in Figure 30-10.



Figure 30-10: Pizzerias close to the user’s location

Do It Yourself: Connect the Show button on the Navigation Bar with an Action called `showLocations()`. Update the `ViewController` class in your project with the code of Listing 30-13. Remember to disconnect the Home button if you don’t use it anymore. Run the application and press the Show button. You should see your location and the pizzerias in the area.

The `setAnnotations()` method looks for places only in the current visible region. If the user scrolls the map to find new places, our application does not update the information. In cases like this, we can take advantage of two methods defined by the `MKMapViewDelegate` protocol that are called when the region changes.

mapView(mapView, regionWillChangeAnimated: animated)—This method is called on the delegate when the visible region is about to change. The **animated** attribute is a Boolean value that indicates if the process is going to be animated.

mapView(mapView, regionDidChangeAnimated: animated)—This method is called on the delegate after the visible region changed. The **animated** attribute is a Boolean value that indicates if the process was animated.

Implementing the `mapView(mapView, regionDidChangeAnimated: animated)` method we can update the annotations every time the user scrolls the map (but only after the user's location was already determined).

Listing 30-14: Updating the annotations

```
func mapView(mapView: MKMapView, regionDidChangeAnimated animated: Bool) {  
    if mapView.userLocation.location != nil {  
        setAnnotations()  
    }  
}
```

Directions

Maps are not only used to find places but also to find routes to go from one place to another. The MapKit framework includes a set of classes to calculate a route and generate the graphics to show it on the map. The first class we need to implement is called `MKDirectionsRequest`. This class generates a request for a route between two locations. The following are the properties available to configure the request.

source—This property sets or returns the route's starting point. It is an object of type `MKMapItem`.

destination—This property sets or returns the route's destination. It is an object of type `MKMapItem`.

requestsAlternateRoutes—This property sets or returns a Boolean value that determines whether or not multiple routes will be returned when available.

transportType—This property sets or returns a value that determines the type of transportation used to travel the route. It is an **MKDirectionsTransportType** structure with the properties **Automobile**, **Walking**, **Transit**, and **Any**.

departureDate—This property sets or returns an **NSDate** object that determines the date of departure and helps the system estimate the better route.

arrivalDate—This property sets or returns an **NSDate** object that determines the date of arrival and helps the system estimate the better route.

The request has to be sent to Apple servers for processing. The framework defines the **MKDirections** class to perform the request and process the results. The class includes the following initializer and method.

MKDirections(request: request)—This initializer creates an **MKDirections** object with the specified request. The **request** attribute is an **MKDirectionsRequest** object.

calculateDirectionsWithCompletionHandler(block)—This method performs the request and sends the results to the closure specified by the **block** attribute. The closure receives two values: an **MKDirectionsResponse** object with the routes found and an **NSError** value to report errors.

The routes are returned as objects of the **MKRoute** class. The class includes the following properties to get the route's information.

polyline—This property sets or returns the route's geometry that we can use to draw the route on the map. It is an object of the **MKPolyline** class.

steps—This property sets or returns an array of **MKRouteStep** objects that describe every step the user has to take to reach the destination, including instructions on how to follow the route.

advisoryNotices—This property sets or returns an array of strings with additional information that the user may need to travel the route, such as traffic jams or interruptions.

distance—This property sets or returns a **CLLocationDistance** value (a typealias of **Double**) with the route distance in meters.

expectedTravelTime—This property sets or returns an **NSTimeInterval** value (a

typealias of **Double**) with the expected travel time in seconds.

After the server returns the **MKRoute** object describing the route, we have to present its values to the user. There are different types of information we can extract from these objects, but the most interesting is the route's geometry provided by the **polyline** property, which allow us to draw the route on the map.

The Map View displays the graphics in layers. There is a layer for the map, another for the labels and roads, and we can also add custom layers to display our own graphics, including the route. The value returned by the **polyline** property contains all the information required to create a layer that we can add to the Map View to draw the route. The **MKMapView** class offers several methods to add and remove layers. The following are the most frequently used.

addOverlay(overlay, level: level)—This method adds a layer to the Map View. The **overlay** attribute is an object that adopts the **MKOverlay** protocol and includes the code necessary to define the layer, and the **level** attribute is an enumeration of type **MKOverlayLevel** that defines the level in which the layer will be placed. The possible values are **AboveRoads** (places the layer above roads but below labels) and **AboveLabels** (places the layer above roads and labels).

removeOverlay(overlay)—This method removes a layer from the Map View. The **overlay** attribute is a reference to the object that represents the layer we want to remove.

Adding the layer to the Map View is like adding an annotation; it provides the information necessary to display the layer but it doesn't draw anything on the map. As with annotations, the Map View calls a delegate method to get the information necessary to draw the layer. The following is the method defined in the **MKMapViewDelegate** protocol for this purpose.

mapView(mapView, rendererForOverlay: overlay)—This method is called on the delegate when the Map View needs a renderer to draw a layer. The method has to return an object of the **MKOverlayRenderer** class (or any of its subclasses) with the renderer we want to use to render the graphics. The **overlay** attribute is a reference to the object that represents the layer we want to draw.

The method must return a renderer that we have to configure according to how we want the layer to be drawn. The framework defines the **MKPolylineRenderer** class (a subclass of a subclass of **MKOverlayRenderer**) to create a renderer for a polyline overlay. The class

includes the following initializer.

MKPolylineRenderer(overlay: **overlay)**—This initializer creates a renderer for a polyline overlay. The **overlay** attribute is a reference to the object that represents the layer we want to draw.

The **MKPolylineRenderer** class inherits from its superclasses a set of properties we can use to configure the renderer.

fillColor—This property sets or returns a **UIColor** object with the color to fill the path.

strokeColor—This property sets or returns a **UIColor** object with the color for the stroke.

lineWidth—This property sets or returns a **CGFloat** value that determines the width of the path.

lineJoin—This property sets or returns a value that determines the style for the corners of the path. It is an enumeration of type **CGLineJoin** with the values **Miter** (joins the lines with a sharp end), **Round** (joins the lines with a rounded end), and **Bevel** (joins the lines with a square end).

lineCap—This property sets or returns a value that determines the style for the ending of the lines. It is an enumeration of type **CGLineCap** with the values **Butt** (squared end), **Round** (round end), and **Square** (squared end extended half the width of the line).

miterLimit—This property sets or returns a value of type **Double** that determines how long the lines in the corners will extend.

lineDashPhase—This property sets or returns a **CGFloat** value that determines the offset at which the renderer starts drawing a dashed line.

lineDashPattern—This property sets or returns an array of numbers that specify the pattern used to create a dashed line.

The route's origin and destination are set with **MKMapItem** objects (the same type of objects we receive when we perform a search). The class includes the following initializer to create an **MKMapItem** object from an **MKPlacemark** object.

MKMapItem(placemark: **placemark)**—This initializer creates an **MKMapItem** object with the information provided by the **placemark** attribute. The attribute is an object of the **MKPlacemark** class (a subclass of **CLPlacemark**) with the information about the

place (location, name, etc.).

The **MKPlacemark** class offers its own initializer to create the **MKPlacemark** object we need to initialize the **MKMapItem** object.

MKPlacemark(coordinate: **coordinate, addressDictionary: **dictionary**)**
—This initializer creates an **MKPlacemark** object with the coordinates specified by the **coordinate** attribute. The **coordinate** attribute is a **CLLocationCoordinate2D** structure with the place’s latitude and longitude, and the **dictionary** attribute is a dictionary that specifies the rest of the values for the placemark, but it was deprecated and it must be declared as **nil**.

The first step to create a route is to define the **MKMapItem** objects for the starting point and destination.

Listing 30-15: Setting the route’s starting point and destination

```
import UIKit
import MapKit

class ViewController: UIViewController, MKMapViewDelegate {
    @IBOutlet weak var mapView: MKMapView!
    var origin: MKMapItem!
    var destination: MKMapItem!
    override func viewDidLoad() {
        super.viewDidLoad()
        let coordOrigin = CLLocationCoordinate2D(latitude: 40.7637825011971, longitude: -73.9731328627541)
        let placeOrigin = MKPlacemark(coordinate: coordOrigin, addressDictionary: nil)
        origin = MKMapItem(placemark: placeOrigin)

        let coordDestination = CLLocationCoordinate2D(latitude: 40.7523809365088, longitude: -73.9778321046893)
        let placeDestination = MKPlacemark(coordinate: coordDestination, addressDictionary: nil)
        destination = MKMapItem(placemark: placeDestination)
        let region = MKCoordinateRegionMakeWithDistance(coordOrigin, 2000, 2000)
        mapView.setRegion(region, animated: false)
        mapView.delegate = self
    }
}
```

The code of Listing 30-15 defines two properties, **origin** and **destination**, to store the

information for the route's starting point and destination. When the view is loaded, these values are initialized with the coordinates of two places in New York: the Apple Store and Grand Central Terminal, and the visible region is set around the Apple Store's area. For this example, we decided to calculate the route when the Show button is pressed. The following is the Action for the button.

Listing 30-16: Calculating the route

```
@IBAction func showDirections(sender: UIBarButtonItem) {  
    let request = MKDirectionsRequest()  
    request.source = origin  
    request.destination = destination  
    request.requestsAlternateRoutes = false  
    let directions = MKDirections(request: request)  
    directions.calculateDirectionsWithCompletionHandler({ (results, error) in  
        if let routes = results?.routes {  
            let route = routes.first!  
            self.mapView.addOverlay(route.polyline, level: .AboveRoads)  
        }  
    })  
}
```

The `showDirections()` method starts by creating the request. We configure the request with the `origin` and `destination` properties and then create the `MKDirections` object to process it. The `calculateDirectionsWithCompletionHandler()` method returns the results in an `MKDirectionsResponse` object, which contains the `routes` property with the `MKRoute` object for the route. Because we set the `requestsAlternateRoutes` property to `false`, the array returned by this property only contains one route object. If the array contains a route, we get the first element and add its polyline overlay to the map. The next step is to implement the `MKMapViewDelegate` protocol method to provide the renderer for the polyline overlays.

Listing 30-17: Drawing the route

```
func mapView(mapView: MKMapView, rendererForOverlay overlay: MKOverlay) -> MKOverlayRenderer {  
    let render = MKPolylineRenderer(overlay: overlay)  
    render.strokeColor = UIColor.redColor()  
    return render  
}
```

The method of Listing 30-17 creates an `MKPolylineRenderer` object for each layer with a stroke in color red. This renderer draws a red line between one location and another, as

illustrated by Figure 30-11.



Figure 30-11: Route on the map

Do It Yourself: Disconnect the Show button from the previous Action and connect it to a new Action called `showDirections()`. Update the `ViewController` class with the codes of Listings 30-15, 30-16, and 30-17. Run the application and press the Show button. You should see the route from the Apple Store to Grand Central Terminal drawn on the map.

Because we centered the visible area at the Apple Store, the user can't see the entire route unless the map is zoomed out or scrolled. There are several things we can do to improve our application, but it always depends on what we want to achieve and how the application is organized. One interesting alternative is to define the annotations for the starting point and destination and then call the `showAnnotations()` method provided by the Map View to set a visible region that includes both annotations. This way not only will the entire route be visible, but we also make sure that the user can identify the places where the trip begins and ends.

Listing 30-18: Zooming out to show the route

```
@IBAction func showDirections(sender: UIBarButtonItem) {
    let request = MKDirectionsRequest()
    request.source = origin
    request.destination = destination
    request.requestsAlternateRoutes = false
    let directions = MKDirections(request: request)
    directions.calculateDirectionsWithCompletionHandler({ (results, error) in
        if let routes = results?.routes {
            let route = routes.first!
```

```

self.mapView.addOverlay(route.polyline, level: .AboveRoads)
let annotation1 = MKPointAnnotation()
annotation1.coordinate = self.origin.placemark.coordinate
annotation1.title = "Apple Store"
self.mapView.addAnnotation(annotation1)
let annotation2 = MKPointAnnotation()
annotation2.coordinate = self.destination.placemark.coordinate
annotation2.title = "Grand Central Terminal"
self.mapView.addAnnotation(annotation2)
self.mapView.showAnnotations([annotation1, annotation2], animated: true)
}
})
}

```

In Listing 30-18 the annotations are added to the map when the results are received from the server. Because of this, the annotations and route are shown at the same time and the map zooms in or out to show the full route on the screen.



Figure 30-12: Entire route on the map

Do It Yourself: Update the `showDirections()` method with the code of Listing 30-18. Run the application and press the show button. You should see the entire route, as shown in Figure 30-12.

How we present information and draw the graphics on the map depends on the requirements of our application, but there are two more methods defined in the `MKMapViewDelegate` protocol that could be useful when working with routes and destinations.

mapView(mapView, didSelectAnnotationView: view)—This method is called on the delegate when an annotation view is selected.

mapView(mapView, didDeselectAnnotationView: view)—This method is called on the delegate when an annotation view is deselected.

Using these methods we can draw the route to the destination selected by the user. The following example looks for coffee shops nearby the Apple Store in New York City and implements the **mapView(mapView, didSelectAnnotationView: view)** method to get the route between the Apple Store and the selected coffee shop.

***Listing 30-19:** Calculating routes to the destinations selected by the user*

```
import UIKit
import MapKit

class ViewController: UIViewController, MKMapViewDelegate {
    @IBOutlet weak var mapView: MKMapView!
    var appleCoord: CLLocationCoordinate2D!
    var route: MKRoute?

    override func viewDidLoad() {
        super.viewDidLoad()
        appleCoord = CLLocationCoordinate2D(latitude: 40.7637825011971, longitude: -73.9731328627541)
        let region = MKCoordinateRegionMakeWithDistance(appleCoord, 2000, 2000)
        mapView.setRegion(region, animated: false)
        mapView.delegate = self

        let request = MKLocalSearchRequest()
        request.naturalLanguageQuery = "Coffee"
        request.region = region
        let search = MKLocalSearch(request: request)
        search.startWithCompletionHandler({ (results, error) in
            if let items = results?.mapItems {
                for item in items {
                    if let coordinates = item.placemark.location?.coordinate {
                        let annotation = MKPointAnnotation()
                        annotation.coordinate = coordinates
                        annotation.title = item.name
                        annotation.subtitle = item.phoneNumber
                        self.mapView.addAnnotation(annotation)
                    }
                }
            }
        })
    }
}
```

```

    }
    })
}
func mapView(mapView: MKMapView, didSelectAnnotationView view: MKAnnotationView) {
    if let destinationCoord = view.annotation?.coordinate {
        let placeOrigin = MKPlacemark(coordinate: appleCoord, addressDictionary: nil)
        let origin = MKMapItem(placemark: placeOrigin)
        origin.name = "Apple Store"
        let placeDestination = MKPlacemark(coordinate: destinationCoord, addressDictionary: nil)
        let destination = MKMapItem(placemark: placeDestination)
        destination.name = view.annotation?.title!
        let request = MKDirectionsRequest()
        request.source = origin
        request.destination = destination
        request.transportType = .Walking
        request.requestsAlternateRoutes = false
        let directions = MKDirections(request: request)
        directions.calculateDirectionsWithCompletionHandler({ (results, error) in
            if let routes = results?.routes {
                if let oldRoute = self.route {
                self.mapView.removeOverlay(oldRoute.polyline)
                self.route = nil
                }
                self.route = routes.first!
                self.mapView.addOverlay(self.route!.polyline, level: .AboveRoads)
            }
        })
    }
}

func mapView(mapView: MKMapView, rendererForOverlay overlay: MKOverlay) -> MKOverlayRenderer {
    let render = MKPolylineRenderer(overlay: overlay)
    render.strokeColor = UIColor.redColor()
    return render
}
}

```

The code of Listing 30-19 doesn't introduce anything new. We search for places related to the word "Coffee" around the Apple Store, create annotations for every place found, and add them to the map. When the user selects a pin, the delegate method is executed and we use the pin's location to get the route between the Apple Store and the place selected. Because the user can select a different place after a route was added to the map, we store a

reference to the route in the `route` property to be able to remove the overlay later. The result is shown in Figure 30-13.



Figure 30-13: Route between the Apple Store and the place selected by the user

Do It Yourself: Update the `ViewController` class with the code of Listing 30-19. Remember to disconnect the buttons to the Actions you don't implement anymore. Run the application and tap on a pin. You should see something similar to Figure 30-13.

There is additional information provided by the route object that may be relevant to the user, including the steps the user has to follow to travel the route and the total distance. The next example is an Action we can connect to the Show button on our interface to display an Alert View with the distance and the estimated time it takes to get to the destination of the current route.

Listing 30-20: Showing route's additional information

```
@IBAction func showRouteDetails(sender: UIBarButtonItem) {
    if let current = route {
        let distance = current.distance
        let time = current.expectedTravelTime
        var message = "Distance: \(distance) meters \r\n"
        message += "Time: \(time) seconds"
        let alert = UIAlertController(title: "Route", message: message, preferredStyle: .Alert)
        let action = UIAlertAction(title: "OK", style: .Default, handler: nil)
        alert.addAction(action)
        presentViewController(alert, animated: true, completion: nil)
    }
}
```

Do It Yourself: Connect the Show button with an Action called `showRouteDetails()`. Complete the method with the code of Listing 30-20. Run the application, select a pin, wait for the route to appear, and press the Show button. You should see an Alert View with the distance and estimated time in seconds.

Chapter 31

Internationalization

31.1 Internationalization

Apple has presence in more than a hundred countries and apps are distributed in dozens of languages and dialects, but we can't publish multiple copies of our application, we have to create only one version and adapt it to each market. This includes not only translations but also organizing the elements in the interface, formatting numbers, etc. The adaptation process is divided in two parts called *Internationalization* and *Localization*. Internationalization is the process of preparing our application to be able to work in any region and language, while Localization is the process of translating the interface and resources to each language.

Fortunately, most of the internationalization process is done automatically for us. For example, the constraints defined with Auto Layout organize the interface according to the direction of the language (right-to-left or left-to-right) without us having to do anything about it. But we still have to format values, such as dates and units of measurement, and translate strings to the respective languages. Usually the translation is done by professional translators and then incorporated into our application, but values have to be formatted from code with the formatter classes provided by the Foundation framework.

Formatters

In Chapter 4 we already introduced two formatters, `NSDateFormatter` and `NSNumberFormatter`, which allow us to turn a value into a string with a specific format. These formatters automatically create the string according to the language and region where the user is located. For example, if we format the value of an `NSDate` object with the `NSDateFormatter` class and set the style for the date as `FullStyle`, the formatter will create the string “Saturday, January 2, 2016” for users located in the US that speak English and the string “sábado, 2 de enero de 2016” for those located in Spain that speak Spanish (see Chapter 4, Listings 4-35 and 4-36).

Configuring the formatters as we did in the examples of Chapter 4 is enough to get the expected results, but the classes provide convenient methods to create localized strings. The following is the type method available for the `NSDateFormatter` class.

`localizedStringFromDate(date, dateStyle: style1, timeStyle: style2)`—This type method returns a string with the value of the `NSDate` object specified by the **`date`** attribute and with a format according to the values specified by the **`style1`** and **`style2`** attributes. The **`style1`** and **`style2`** attributes are enumerations of type `NSDateFormatterStyle` with the values `NoStyle`, `ShortStyle`, `MediumStyle`, `LongStyle`, and `FullStyle`.

The `NSNumberFormatter` class also includes its own type method to localize numbers.

localizedStringFromNumber(number, numberStyle: style)—This type method returns a string with the value of the **number** attribute and with a format according to the value specified by the **style** attribute. The **style** attribute is an enumeration of type `NSNumberFormatterStyle` with the values `NoStyle`, `DecimalStyle`, `CurrencyStyle`, `PercentStyle`, `ScientificStyle`, and `SpellOutStyle`.

Besides these classes, we also have classes for specific units or periods of time. For instance, there are two more classes to format dates and time: `NSDateComponentsFormatter` and `NSDateIntervalFormatter`. The `NSDateComponentsFormatter` class provides the following type method to create a localized string from date components.

localizedStringFromDateComponents(components, unitsStyle: style)—This type method returns a string with the components specified by the **components** attribute and with a format according to the value specified by the **style** attribute. The **components** attribute is an `NSDateComponents` object with the date components we want to include in the string, and the **style** attribute is an enumeration of type `NSDateComponentsFormatterUnitsStyle` that defines how the components are going to be described. The possible values are `Positional`, `Abbreviated`, `Short`, `Full`, and `SpellOut`.

We can also localize an interval of time with the `NSDateIntervalFormatter` class. For example, we may have two dates seven days apart, January 2nd and January 9th, and want to create a string that reflects the interval. For users in the US that speak English, the string will be something like “2 – 9 Jan 2016” and users in Spain that speak Spanish will see “2–9 ene 2016”. The class defines the following properties and methods to configure the interval and create the string.

dateStyle—This property sets or returns the style for the date. It is an enumeration of type `NSDateIntervalFormatterStyle` with the values `NoStyle`, `ShortStyle`, `MediumStyle`, `LongStyle`, and `FullStyle`.

timeStyle—This property sets or returns the style for the time. It is an enumeration of type `NSDateIntervalFormatterStyle` with the values `NoStyle`, `ShortStyle`, `MediumStyle`, `LongStyle`, and `FullStyle`.

stringFromDate(date1, toDate: date2)—This method returns a localized string that represents the interval between the dates specified by the **date1** and **date2**

attributes.

Foundation also includes formatters for numbers that are associated to unit of measurement, such as kilograms, meters, etc. The class to format energy (calories) is called `NSEnergyFormatter`, and includes the following methods to create a localized string.

stringFromJoules(joules)—This method returns a localized string with the value specified by the **joules** attribute converted to units appropriate for the user’s locale. The **joules** attribute is a value of type `Double` that determines the energy in joules.

stringFromValue(value, unit: unit)—This method returns a localized string with the value and unit specified by the attributes. The **value** attribute is a value of type `Double` that determines the amount of energy, and the **unit** attribute is an enumeration of type `NSEnergyFormatterUnit` that defines the unit of measurement. The possible values are `Joule`, `Kilojoule`, `Calorie`, and `Kilocalorie`.

The class to format lengths is called `NSLengthFormatter`, and includes the following methods to create the localized string.

stringFromMeters(meters)—This method returns a localized string with the value specified by the **meters** attribute converted to units appropriate for the user’s locale. The **meters** attribute is a value of type `Double` that determines the length in meters.

stringFromValue(value, unit: unit)—This method returns a localized string with the value and unit specified by the attributes. The **value** attribute is a value of type `Double` that determines the length, and the **unit** attribute is an enumeration of type `NSLengthFormatterUnit` that defines the unit of measurement. The possible values are `Millimeter`, `Centimeter`, `Meter`, `Kilometer`, `Inch`, `Foot`, `Yard`, and `Mile`.

Another formatter class in the Foundation framework is `NSMassFormatter`. This class can format numbers that represent mass. The following are the methods provided by the class to create localized strings.

stringFromKilograms(kilograms)—This method returns a localized string with the value specified by the **kilograms** attribute converted to units appropriate for the user’s locale. The **kilograms** attribute is a value of type `Double` that determines the mass in kilograms.

stringFromValue(value, unit: unit)—This method returns a localized string with the value and unit specified by the attributes. The **value** attribute is a value of type

Double that determines the mass, and the **unit** attribute is an enumeration of type **NSMassFormatterUnit** that defines the unit of measurement. The possible values are **Gram**, **Kilogram**, **Ounce**, **Pound**, and **Stone**.

Values are only localized when we use the appropriate formatter, so we have to implement these classes every time we want the values to be shown with the right format in every region and language. As an example we have created a simple interface that shows a list of localized values.

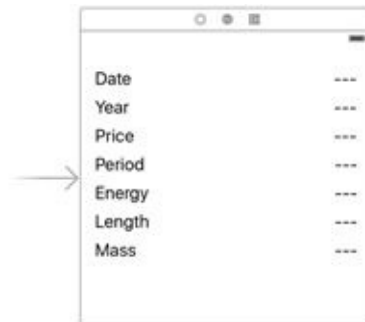


Figure 31-1: Localized interface

The view controller for this interface has to get the strings for each value and assign them to the labels on the right.

Listing 31-1: Creating localized strings

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var dateLabel: UILabel!
    @IBOutlet weak var yearLabel: UILabel!
    @IBOutlet weak var priceLabel: UILabel!
    @IBOutlet weak var periodLabel: UILabel!
    @IBOutlet weak var energyLabel: UILabel!
    @IBOutlet weak var lengthLabel: UILabel!
    @IBOutlet weak var massLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        let today = NSDate()
        let date = NSDateFormatter.localizedStringFromDate(today, dateStyle: .MediumStyle, timeStyle: .NoStyle)
        dateLabel.text = date

        let calendar = NSCalendar.currentCalendar()
        let components = calendar.components(NSCalendarUnit.Day, fromDate: today)
```

```

let year = NSDateComponentsFormatter.localizedStringFromDateComponents(components, unitsStyle: .Full)
yearLabel.text = year

let price = NSNumberFormatter.localizedStringFromNumber(120.59, numberStyle: .CurrencyStyle)
priceLabel.text = price

let later = NSDate(timeInterval: 60 * 60 * 24 * 7, sinceDate: today)
let formatterInterval = NSDateIntervalFormatter()
formatterInterval.dateStyle = .MediumStyle
formatterInterval.timeStyle = .NoStyle

let interval = formatterInterval.stringFromDate(today, toDate: later)
periodLabel.text = interval

let formatterEnergy = NSNumberFormatter()
let energy = formatterEnergy.stringFromJoules(2000)
energyLabel.text = energy

let formatterLength = NSNumberFormatter()
let length = formatterLength.stringFromMeters(150)
lengthLabel.text = length

let formatterMass = NSNumberFormatter()
let mass = formatterMass.stringFromKilograms(95)
massLabel.text = mass
}
}

```

The code of Listing 31-1 creates localized strings with all the formatter classes studied in this chapter and random values. Figure 31-2 illustrates how the values are presented in different regions and languages. The picture on the left is what the user will see when the device is located in the US and configured for the English language, the picture in the center shows the values in Spanish for people in Spain, and the picture on the right will be seen in Egypt by people that speak Arabic.

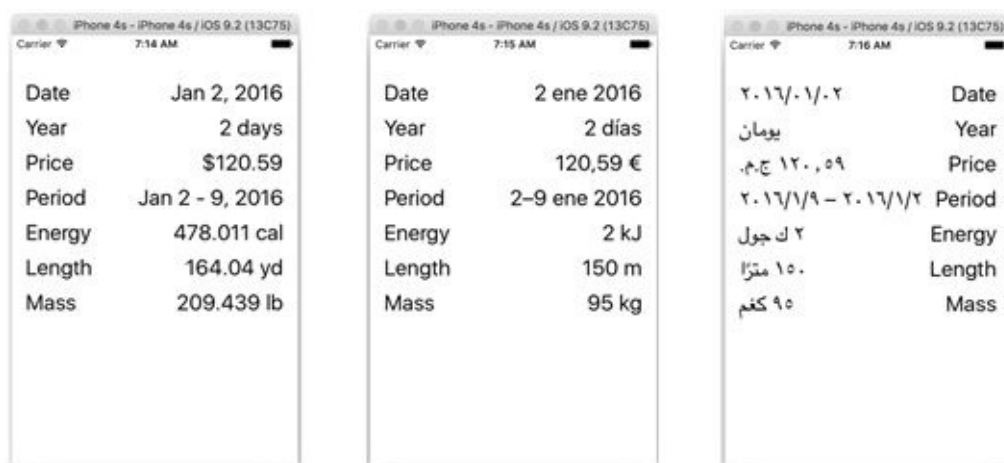


Figure 31-2: Interface in different regions and languages

Do It Yourself: Create a new Single View Application project. Add seven labels on the left side of the initial view with the texts Date, Year, Price, Period, Energy, Length, and Mass. Add another seven labels on the right with placeholders, as shown in Figure 31-1, and connect them to the **ViewController** class with Outlets called **dateLabel**, **yearLabel**, **priceLabel**, **periodLabel**, **energyLabel**, **lengthLabel**, and **massLabel**. Remember to pin all the labels to the sides of the view and to each other with Leading, Trailing, and Space constraints. Complete the **ViewController** class with the code of Listing 31-1. Run the application. You should see the labels on the right with values formatted according to your location and language.

Languages

Formatting the values to adapt to every language doesn't make our app ready for international markets. Some of the information, like the currency symbol, is region-dependent, but some, like the name of days and months, is language-dependent. By default, the app is distributed to every market with the Base language (the language used in development), but we can add the rest of the languages we want our app to work with in the project's settings. The project's settings are available in the app's settings panel. We have to open the app settings (Figure 31-3, number 1), click on the button at the top-left corner of the Editor Area to open the menu (Figure 31-3, number 2), and, instead of selecting the Target as we did before to configure the app, we have to select the project (Figure 31-3, number 3).

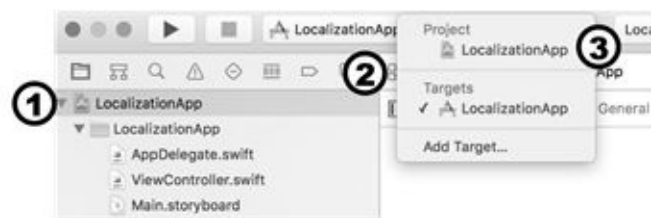


Figure 31-3: Project's settings

After the project is selected, the Editor Area lists a few sections with the project's settings. The section that allows us to edit the languages available for our app is called *Localizations*, as shown in Figure 31-4.



Figure 31-4: Languages available

The Localizations section shows the languages currently available (number 1). If we didn't add a new language before, the only one on the list will be the Base language (the language used in development). To add a new language we have to press the + button at the bottom of the section (number 2) and select the language from the popup menu. After the language is selected, Xcode displays a window where we have to select the resources the language is going to be associated with (usually all the resources are selected) and press Finish to add the language. Figure 31-4 shows what we see when the process is over (for this example we added the languages Spanish and Arabic).



Language	Resources
Arabic	2 Files Localized
English — Development Language	2 Files Localized
Spanish	2 Files Localized

Figure 31-5: New languages available

Now our app is ready to work with all the languages on the list, but if we run it, it will still show the strings according to the region and language set on the simulator or the device. This is because the application automatically detects these settings and formats the strings accordingly. To see what the application will look like when it is executed by our international users, we can simulate a region and a language from the Scheme.

We introduced Schemes in Chapter 5. A Scheme determines where our application is going to be executed and the settings it will use. The advantage of using a Scheme is that we can configure the target with the values we want. The Scheme settings are accessible through a popup menu that opens when we click on the Scheme button in the Xcode's Toolbar (Figure 5-4). The menu offers an option to edit the current Scheme or create a new one, as shown in Figure 31-6.

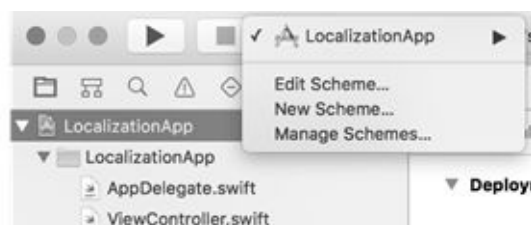


Figure 31-6: Scheme's menu

When we select the Edit Scheme option, Xcode displays a window where we can select the process for which we want to configure the Scheme. Figure 31-7 shows the options

available.

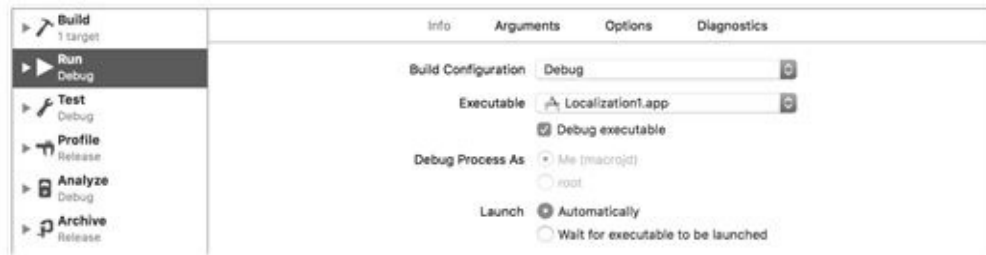


Figure 31-7: *Configuring the Scheme*

If what we want is to see how the interface adapts to a different region or language when the app is executed, we have to change the settings for the Run option. When this option is selected, the panel on the right offers four tabs: Info, Arguments, Options, and Diagnostics. The values for the region and language are shown in the Options tab. At the bottom of the list, there are two controls with this information called Application Language and Application Region. Figure 31-8 shows what the controls look like when we change the values to Arabic and Egypt.



Figure 31-8: *Scheme's region and language*

After the new region and language are selected, we can execute the app and check the strings produced to confirm that the app is ready for that particular market.

Do It Yourself: Open the app's settings (Figure 31-3, number 1). Click on the button at the top-left corner of the Editor Area and select the project from the popup menu (Figure 31-3, number 3). Go down to Localizations and press the + button to add more languages. Add the Arabic language and press Finish. Repeat the process for Spanish (Figure 31-5). Press the Scheme button and select the Edit Scheme option. Select the Run option and activate the Options tab (Figure 31-7). Set the language and region to the ones you want to test. Run the application. You should see the interface adapted to the new settings (Figure 31-2).

Translation

The project developed so far formats the values according to the region and language set

on the device, and even reorganizes the elements on the interface according to the direction of the language (right-to-left for Arabic), but it doesn't translate the text we defined in the Storyboard for the labels on the left. The word Date is shown always as Date, no matter the language selected in the app's settings (see Figure 31-2). The texts are not automatically translated; we have to translate them ourselves one by one.

The process of translating every single text in our application could be exhausting and error prone, but fortunately Xcode offers a tool that extracts the texts and puts them inside a single file we can send to the translators. The option is available in the Editor menu. We have to open the app's settings (Figure 31-3, number 1), go to the Editor menu, and select the option Export for Localization, as shown in Figure 31-9 (the option is only shown when the app's settings panel is open).

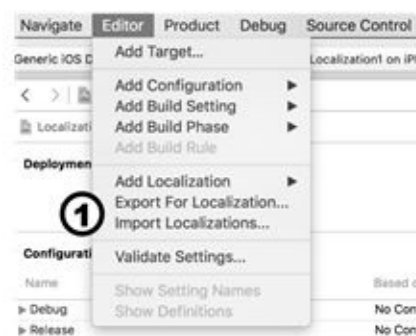


Figure 31-9: Localization options in the Editor menu

A window pops up asking us to select the languages we want to translate the application to and where we want the files to be stored. The system creates a folder and one file per language. For example, if we select the languages added to our project in Figure 31-5, the folder will include the files ar.xliff for Arabic and es.xliff for Spanish (English is not included because it is the language used in development).

The Basics: The xliff files contain XML code. As we have seen before, this is a language that uses tags to enclose the values and determine their purpose. The original texts are enclosed in <source> tags, and the translations are enclosed in <target> tags. Translators have special software that lets them access and modify these values, but you can do it yourself from Xcode. Open the Finder application, find the folder where the files were stored, and double click the file you want to open. Search for the <target> tag, translate the word, and save the file.

Once the files are translated, we have to import them back to our application. The Editor menu offers the Import Localizations option for this purpose (see Figure 31-9, number 1). When this option is selected, Xcode shows a window that exposes the errors found (e.g., texts that the translator forgot to translate or differences between the new file and an old file already in the project) and lets us decide whether to import the file or not.

After the files are imported, the translations become available for our application. Figure 31-10 shows what the app looks like after the translations for Arabic and Spanish are imported.



Figure 31-10: Interface translated to Spanish and Arabic

Do It Yourself: Open the app’s settings (Figure 31-3, number 1). Go to the Editor menu and select the option Export for Localization. Select the folder where you want to store the files, mark all the languages you want to translate the app to (Arabic and Spanish), and press the Save button. Xcode generates two files with the xcliff extension (one for each language). Open the files in Xcode and translate the strings in between <target> tags, or download the files already translated from our website. Go to the Editor menu again and select the Import Localizations to import the translated files back. Press the Scheme button and select the Edit Scheme option from the popup menu (Figure 31-6). Select the Run option, open the Options tab, and change the language and region you want to test. Run the application. You should see one of the interfaces shown in Figure 31-9 depending on the language and region selected.

NSLocalizedString

All the strings in the interface are automatically included in the translation files, but the strings defined in the code are not. The system doesn’t know which are the strings in the code that are going to be shown to the user and therefore require translation. To turn the strings into localizable strings, Foundation includes a function called `NSLocalizedString()`. The most frequently used version of this function takes a string with the original text and a second string with a comment that provides additional information about the string to help the translator get the most accurate translation.

NSLocalizedString(string, comment)—This function returns a localized string. The **string** attribute is a string with the original text, and the **comment** attribute is a string with a comment that describes the purpose of the text to help the translator.

All the strings in our code that we want to include in the translation files have to be created with the `NSLocalizedString()` function. The following example creates a localized string and assigns it to a label.

Listing 31-2: Creating localized strings

```
import UIKit
```

```
class ViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!
    override func viewDidLoad() {
        super.viewDidLoad()

        let text = NSLocalizedString("Hello User!", comment: "Welcome message for new users")
        titleLabel.text = text
    }
}
```

The `NSLocalizedString()` method returns a `String` value with the string specified by the first attribute ("Hello User!"). If a translation is found for the user's language, the `String` value includes the translation instead.

The Basics: The process of translating the texts is the same as before. When we export the files, Xcode takes the strings from the interface and all the strings in the code that are localized with the `NSLocalizedString()` function and includes them in the translation files for each language.

The `NSLocalizedString()` method can also work with formatted strings, but the placeholders are adapted to be translated and therefore require a method that can process these types of format. The `String` structure includes the following type method for this purpose.

localizedStringWithFormat(string, arguments)—This type method creates string from a localized string. The **string** attribute is the localized string we want to format, and the **arguments** attribute is the list of values we want to include in the string separated by comma.

To format a localized string, we have to create the string with the placeholders first and then call the `localizedStringWithFormat()` method with the values we want to use to replace the placeholders.

Listing 31-3: Formatting localized strings

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var titleLabel: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()

        let localizedText = NSLocalizedString("%d hours and %d minutes", comment: "Time the app was running")
        let text = String.localizedStringWithFormat(localizedText, 2, 35)
        titleLabel.text = text
    }
}
```

Do It Yourself: Create a new Single View Application project. Add a label to the initial view and connect it to the `ViewController` class with an Outlet called `titleLabel`. Complete the class with the code of Listing 31-2 or 31-3. If you want to see how the strings are translated, you can follow the same procedure we used before to create and import the translation files.

Resources

Besides text, applications usually contain images, videos, and audio files that need translation. Of course, the way we work with our translator in this case is different; we have asked the translator to translate the texts and then create different versions of the media for each language, but we can use localized texts in our code to load the file corresponding to the user's language. For instance, we can localize the image's file name to load one image or another according to the device's settings.

Listing 31-4: Localizing files

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var logoImage: UIImageView!
```

```

override func viewDidLoad() {
    super.viewDidLoad()

    let logoFile = NSLocalizedString("logoen", comment: "File name. Replace suffix")
    logoImage.image = UIImage(named: logoFile)
}
}

```

The code of Listing 31-4 localized the string “logoen”. In the comments we suggest to the translator to replace the suffix “en” by the one corresponding to the language and therefore we have to provide a file for each translation. For this example we have to prepare three files, one for English (logoen.png), another for Spanish (logoes.png), and one more for Arabic (logoar.png). If we export the translation files and translate the string to logoes and logoar, we will see an image on the screen depending on the device’s settings, as shown in Figure 31-11.



Figure 31-11: Localized images

Do It Yourself: Create a new Single View Application project. Add an Image View to the initial view and connect it to the **ViewController** class with an Outlet called **logoImage**. Complete the **ViewController** class with the code of Listing 31-4. Add to the project the languages Spanish and English (see Figure 31-5). Export the translation files (Figure 31-9). Translate the “logoen” string into “logoes” and “logoar” and import the files back into the project. Download the logoen.png, logoes.png, and logoar.png files from our website and add them to the Assets Catalog. Set the language you want to try (Figure 31-8) and run the application. You should see the image corresponding to the selected language (Figure 31-11).

Chapter 32

App Store

32.1 Publishing

At the beginning of this book we talked about Apple's strict control over the applications the users have access to. Because of Apple's politics, we can't sell applications on our own; we have to publish them in the App Store. The applications are published from Xcode, but there are a series of requirements we need to satisfy for our app to be published and become available to users.

- We need an Apple Developer Program membership.
- We need a Distribution Certificate.
- We need a Provisioning Profile for distribution.
- We need an App ID for each application.
- We have to register the app in the iTunes Connect website.
- We have to create an archive with our app to send to Apple's servers.
- We have to upload the archive to iTunes Connect for review.

Apple Developer Program

Developing and testing can be done with a free account, but publishing our app requires a membership to the Apple Developer Program. The option to enroll in this program is available in the **developer.apple.com** website. We have to click on the Program option at the top of the screen, press the Enroll button, and follow the instructions to register an account for an Individual or an organization. At the time of this writing, the membership costs 99 USD a year.

Certificates, Provisioning Profiles, and Identifiers

Apple wants to make sure that only authorized apps are running on its devices, so it requests developers to add a cryptographic signature to each application. There are three values necessary to authorize the app: certificates, provisioning profiles, and identifiers. Basically, a certificate identifies the developer that publishes the application, the provisioning profile identifies the device that is allowed to run the application, and an identifier, called *App ID*, identifies the application. These values are packed along with the application's files and therefore Apple always knows who developed the app, who is authorized to run it, and in which devices.

As we have seen in Chapter 5, Xcode can generate certificates and provisioning profiles for us. Sometimes the program finds that a value doesn't exist and offers to fix the problem for us, but other times we will have to do it ourselves. The options are available from the Preferences option in the Xcode's menu at the top of the screen. We have to open the Accounts tab (see Figure 5-7, number 1), select our account under Team Name, and press the View Details button (our account has to be registered in Xcode as explained in Chapter 5). The next window shows the list of certificates and provisioning profiles already available.

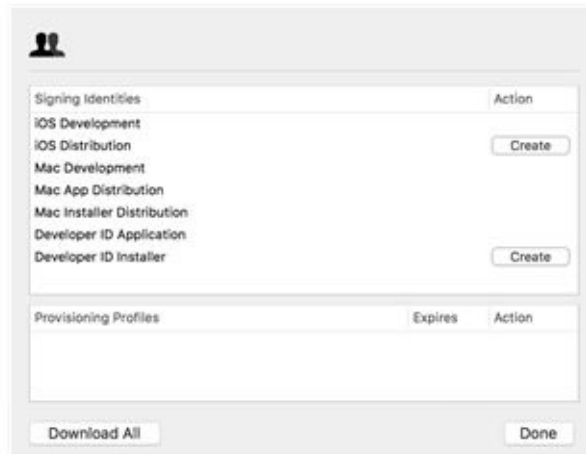


Figure 32-1: Certificates and Provisioning Profiles

There are several types of certificates available, but we usually need only two: one for development and another for distribution. The **development certificate** is used to test the application in our devices and the **distribution certificate** is used to run it in users' devices. As shown by Figure 32-1, Xcode includes both certificates at the top of the list (iOS Development and iOS Distribution). These certificates are automatically created by Xcode when we register our account, but sometimes they may be deactivated and need to be created again. If some of the certificates are not available, Xcode shows a Create button on the right-hand side that we have to press to create it.

Two **provisioning profiles** are also required for development and distribution, but Xcode also takes care of this. When we try to run the app on a device or create the files to submit the application to the App Store, Xcode checks whether or not the respective provisioning profile is available and creates a new one if necessary.

Although the process has been automatized by Xcode, there are times in which we need to create or edit the certificates and provisioning profiles ourselves. The home screen of the **developer.apple.com** website includes an icon called Certificates, Identifiers & Profiles to create and modify these values (the option is not available for free members). Figure 32-2 shows the menu we see after we click on this icon.



Figure 32-2: Options to manage certificates, provisioning profiles, and identifiers

No matter the option we select on this screen, the website opens the same page where we can create, edit, or remove certificates, provisioning profiles and identifiers. The page contains two panels. The left panel offers a list of options to select the type of values we want to work with and the right panel shows the list of values selected and buttons to create new ones. For example, if we select the option to see all the certificates available, we will see the development and distribution certificates created from Xcode and a + button to add another one.

iOS Certificates		
2 Certificates Total		
Name	Type	Expires
John D Gauchat	iOS Development	Oct 17, 2016
John D Gauchat	iOS Distribution	Jan 06, 2017

Figure 32-3: Managing certificates

We usually don't need to create certificates and provisioning profiles from these panels, but we have to use it to create the **App ID** necessary to publish our app. The process is simple: we have to select App IDs from the left panel and press the + button at the top of the right panel. The system will ask for a name (any custom value we want that can help us identify later the ID on the list of IDs available), the type of ID we want to create (Explicit App ID is the one required for distribution), the Bundle ID assigned to our application (this value is available in the app's settings in Xcode), and the services we want our app to be able to use (e.g., iCloud).

iTunes Connect

With the App ID ready, the next step is to create a record on Apple's server to upload the application's files. Apple has designated a special website for this purpose. Because

iTunes is the system Apple uses to manage user's applications, the website is called iTunes Connect, and it is available at **itunesconnect.apple.com**.

iTunes Connect is integrated with Apple's developing system and therefore it already contains information of our developer account. To login, we have to use the same Apple ID and password we use to access our account at developer.apple.com. Figure 32-4 illustrates the options available on this website.



Figure 32-4: iTunes Connect menu

From this panel we can insert our financial information, required to put our apps on the market (Agreements, Tax, and Banking), publish our apps (My Apps), and see how the business is going (Sales and Trends). The first step is to create a record of the app we want to publish from the My Apps option. When we click on this icon, a new window shows the list of our apps and a + button at the top to add new ones.

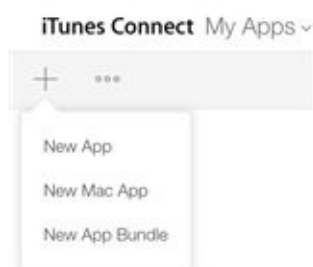


Figure 32-5: Menu to add apps to our account

To add a new app, we have to click the New App button and insert the app's information. The first window asks for the platform (iOS or tvOS), the application's name, the primary language, the bundle ID (this is the App ID we have created before from the developer website), and a custom ID that can help you identify the app. After this information is inserted we have to press the Create button and complete the rest of the information required. This includes the app's description, screenshots, and also personal information. We also have to select the option Pricing and Availability on the left to set the price and where the application will be available. Once all the information is provided, we can finally press the Save button and prepare the files for uploading.

IMPORTANT: iTunes Connect requires a copy of our app’s icon to show in iTunes and the App Store. This version of the icon must have a minimum size of 1024 x 1024 pixels. Remember to also add the rest of the icons to the app and design the launch screen before submitting your app. See Chapter 8 for more details.

Submitting the Application

The application and resources must be compiled in a single archive and then submitted to iTunes Connect. But before we have to make sure that the app is ready for distribution. The options to configure the application are available in the Build Settings tab of the app’s settings editor. We have to open the app’s settings (Figure 31-3, number 1), select the Build Settings tab, and then assign the distribution certificate (iOS Distribution) to the Code Signing Identity options, as shown in Figure 32-6, number 1.



Figure 32-6: Distribution settings

Once the settings are ready, we have to create the archive. The option is available on Xcode’s Product menu. The option is only available when a device is selected on the Schemes, so we have to select the device (it could be a generic device or the device currently connected to the computer) and click on the Archive option inside the Product menu. Xcode compiles the application and creates the archive. The next window shows the archive and offers options to validate and submit the app.



Figure 32-7: List of archives created for our apps

Figure 32-7 shows an archive created for an application called Books Collector (number 1). The item representing the archive includes the date it was created, the app's version, and the number of the archive (we can send multiple archives to iTunes Connect and later decide which one we want to send for review).

The Basics: The app's version is determined from the app's settings editor (by default it is set as 1.0). The version's number has to be specified as integers separated by one or two periods (e.g., 1.0 or 1.2.5). The values represent different revisions of our app, with the order of relevance from left to right. The actual meaning of the value is arbitrary, but we are required to change it every time an update is published to the App Store to reflect how big the update was.

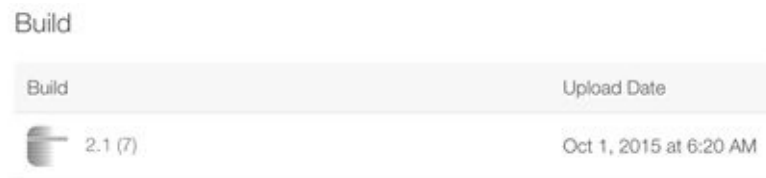
Although it is not required, we should always validate the archive before submitting the app to iTunes Connect. The process allows Xcode to detect errors and suggest a fix or do it for us. When we press the Validate button (Figure 32-7, number 2) a dialog asks for the developing team to associate to the application and shows the app's provisioning profile and entitlements. We have to check this information and press the Validate button to initiate the validation process.



Figure 32-8: Ready to validate

During validation, Xcode reads the project's configuration and checks that everything is fine. If no errors are found, the app is ready for submission. We can press the Upload to App Store button (Figure 32-7, number 3), check again that the values are correct, and press Submit.

As we mentioned before, we may submit multiple archives to iTunes Connect. For this reason, we have to go back to the iTunes Connect website, open the description of our application, and select the archive we just uploaded (it may take a few minutes to be available). Figure 32-9 shows the option with the archive (build) we just uploaded for the Books Collector application.



Build	Upload Date
 2.1 (7)	Oct 1, 2015 at 6:20 AM

Figure 32-9: *Selecting the build to send to the App Store*

After the archive is selected, we can press the Save button to save the app’s description. If all the required information was provided, we can finally press the Submit for Review link at the top of the page to submit the application. The system asks us a few questions and the application is sent for review (the message Waiting for Review is shown below the app’s title).

The process takes a few days to be completed. If everything is right and the app is accepted, Apple sends us an email to let us know that the app has become available in the App Store.

IMPORTANT: Because of the multiple systems involved in the publication and submission processes, you may experience unexpected results that will force you to follow a different set of steps than those presented in this chapter. Apple offers a thorough distribution guide that explains the process in detail and can help you solve any problem you find on the way. For more information, visit our website and follow the links for this chapter.

For Masterminds

Book Series

for more books visit

www.formasterminds.com