



Big Nerd
Ranch

5TH EDITION

iOS Programming

THE BIG NERD RANCH GUIDE

Christian Keur and Aaron Hillegass

iOS Programming: The Big Nerd Ranch Guide

by Christian Keur and Aaron Hillegass

Copyright © 2015 Big Nerd Ranch, LLC

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, contact

Big Nerd Ranch, LLC
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
<http://www.bignerdranch.com/>
book-comments@bignerdranch.com

The 10-gallon hat with propeller logo is a trademark of Big Nerd Ranch, LLC.

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
<http://www.informit.com>

The authors and publisher have taken care in writing and printing this book but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

App Store, Apple, Cocoa, Cocoa Touch, Finder, Instruments, iCloud, iPad, iPhone, iPod, iPod touch, iTunes, Keychain, Mac, Mac OS, Multi-Touch, Objective-C, OS X, Quartz, Retina, Safari, and Xcode are trademarks of Apple, Inc., registered in the U.S. and other countries.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

ISBN-10 0134389379
ISBN-13 978-0134389370

Fifth edition, first printing, December 2015
Release D.5.1.1

Acknowledgments

While our names appear on the cover, many people helped make this book a reality. We would like to take this chance to thank them.

- First and foremost we would like to thank Joe Conway for his work on the earlier editions of this book. He authored the first three editions and contributed greatly to the fourth edition as well. Many of the words in this book are still his, and for that, we are very grateful.
- A few people in particular went above and beyond with their help on this book. They are Mikey Ward, Juan Pablo Claude, and Chris Morris.
- The other instructors who teach the iOS Bootcamp fed us with a never-ending stream of suggestions and corrections. They are Ben Scheirman, Bolot Kerimbaev, Brian Hardy, Chris Morris, JJ Manton, John Gallagher, Jonathan Blocksom, Joseph Dixon, Juan Pablo Claude, Mark Dalrymple, Matt Bezark, Matt Mathias, Mike Zornek, Mikey Ward, Pouria Almassi, Rod Strougo, Scott Ritchie, Step Christopher, Thomas Ward, TJ Usiyan, and Tom Harrington. These instructors were often aided by their students in finding book errata, so many thanks are due to all the students who attend the iOS Bootcamp.
- Thanks to all of the employees at Big Nerd Ranch who helped review the book, provided suggestions, and found errata.
- Our tireless editor, Elizabeth Holaday, took our distracted mumblings and made them into readable prose.
- Anna Bentley jumped in to provide proofing.
- Ellie Volckhausen designed the cover. (The photo is of the bottom bracket of a bicycle frame.)
- Chris Loper at IntelligentEnglish.com designed and produced the print book and the EPUB and Kindle versions.
- The amazing team at Pearson Technology Group patiently guided us through the business end of book publishing.

The final and most important thanks goes to our students whose questions inspired us to write this book and whose frustrations inspired us to make it clear and comprehensible.

This page intentionally left blank

Table of Contents

Introduction	xi
Prerequisites	xi
What Has Changed in the Fifth Edition?	xi
Our Teaching Philosophy	xii
How to Use This Book	xiii
How This Book Is Organized	xiii
Style Choices	xv
Typographical Conventions	xv
Necessary Hardware and Software	xv
1. A Simple iOS Application	1
Creating an Xcode Project	2
Model-View-Controller	5
Designing Quiz	6
Interface Builder	7
Building the Interface	8
Creating view objects	9
Configuring view objects	11
Running on the simulator	12
A brief introduction to Auto Layout	13
Making connections	16
Creating the Model Layer	21
Implementing action methods	22
Loading the first question	22
Building the Finished Application	23
Application Icons	24
Launch Screen	26
2. The Swift Language	27
Types in Swift	27
Using Standard Types	28
Inferring types	30
Specifying types	30
Literals and subscripting	32
Initializers	33
Properties	34
Instance methods	34
Optionals	34
Subscripting dictionaries	36
Loops and String Interpolation	37
Enumerations and the Switch Statement	39
Enumerations and raw values	40
Exploring Apple’s Swift Documentation	40
3. Views and the View Hierarchy	41
View Basics	42
The View Hierarchy	42
Creating a New Project	44

- Views and Frames 45
 - Customizing the labels 53
- The Auto Layout System 55
 - Alignment rectangle and layout attributes 56
 - Constraints 57
 - Adding constraints in Interface Builder 59
 - Intrinsic content size 60
 - Misplaced views 62
 - Adding more constraints 63
 - Bronze Challenge: More Auto Layout Practice 64
- 4. Text Input and Delegation 65
 - Text Editing 65
 - Keyboard attributes 68
 - Responding to text field changes 69
 - Dismissing the keyboard 72
 - Implementing the Temperature Conversion 73
 - Number formatters 75
 - Delegation 76
 - Conforming to a protocol 76
 - Using a delegate 77
 - More on protocols 79
 - Bronze Challenge: Disallow Alphabetic Characters 79
- 5. View Controllers 81
 - The View of a View Controller 82
 - Setting the Initial View Controller 82
 - UITabBarController 85
 - Tab bar items 88
 - Loaded and Appearing Views 91
 - Accessing subviews 92
 - Interacting with View Controllers and Their Views 92
 - Silver Challenge: Dark Mode 93
 - For the More Curious: Retina Display 93
- 6. Programmatic Views 95
 - Creating a View Programmatically 97
 - Programmatic Constraints 98
 - Anchors 99
 - Activating constraints 100
 - Layout guides 101
 - Margins 102
 - Explicit constraints 103
 - Programmatic Controls 104
 - Bronze Challenge: Another Tab 106
 - Silver Challenge: User’s Location 106
 - Gold Challenge: Dropping Pins 106
 - For the More Curious: NSAutoresizingMaskLayoutConstraint 107
- 7. Localization 109
 - Internationalization 110
 - Formatters 110

Base internationalization	113
Preparing for localization	114
Localization	121
NSString and strings tables	124
Bronze Challenge: Another Localization	127
For the More Curious: NSBundle's Role in Internationalization	127
For the More Curious: Importing and Exporting as XLIFF	128
8. Controlling Animations	129
Basic Animations	130
Closures	130
Another Label	132
Animation Completion	135
Animating Constraints	135
Timing Functions	139
Bronze Challenge: Spring Animations	141
Silver Challenge: Layout Guides	141
9. UITableView and UITableViewController	143
Beginning the Homeowner Application	143
UITableViewController	145
Subclassing UITableViewController	146
Creating the Item Class	147
Custom initializers	147
UITableView's Data Source	149
Giving the controller access to the store	150
Implementing data source methods	152
UITableViewCell	153
Creating and retrieving UITableViewCell	155
Reusing UITableViewCell	156
Content Insets	158
Bronze Challenge: Sections	159
Silver Challenge: Constant Rows	159
Gold Challenge: Customizing the Table	159
10. Editing UITableView	161
Editing Mode	161
Adding Rows	166
Deleting Rows	168
Moving Rows	169
Displaying User Alerts	170
Design Patterns	174
Bronze Challenge: Renaming the Delete Button	174
Silver Challenge: Preventing Reordering	174
Gold Challenge: Really Preventing Reordering	174
11. Subclassing UITableViewCell	175
Creating ItemCell	176
Exposing the Properties of ItemCell	178
Using ItemCell	179
Dynamic Cell Heights	180
Dynamic Type	181

- Responding to user changes 184
- Bronze Challenge: Cell Colors 184
- 12. Stack Views 185
 - Using UINavigationController 187
 - Implicit constraints 188
 - Stack view distribution 191
 - Nested stack views 192
 - Stack view spacing 192
 - Segues 194
 - Hooking Up the Content 195
 - Passing Data Around 200
 - Bronze Challenge: More Stack Views 201
- 13. UINavigationController 203
 - UINavigationController 205
 - Navigating with UINavigationController 209
 - Appearing and Disappearing Views 210
 - Dismissing the Keyboard 211
 - Event handling basics 212
 - Dismissing by pressing the Return key 213
 - Dismissing by tapping elsewhere 214
 - UINavigationController 216
 - Adding buttons to the navigation bar 218
 - Bronze Challenge: Displaying a Number Pad 220
 - Silver Challenge: A Custom UITextField 220
 - Gold Challenge: Pushing More View Controllers 220
- 14. Camera 221
 - Displaying Images and UIImageView 222
 - Adding a camera button 224
 - Taking Pictures and UIImagePickerController 226
 - Setting the image picker’s sourceType 226
 - Setting the image picker’s delegate 228
 - Presenting the image picker modally 228
 - Saving the image 230
 - Creating ImageStore 231
 - Giving View Controllers Access to the Image Store 232
 - Creating and Using Keys 233
 - Wrapping Up ImageStore 236
 - Bronze Challenge: Editing an Image 237
 - Silver Challenge: Removing an Image 237
 - Gold Challenge: Camera Overlay 237
 - For the More Curious: Navigating Implementation Files 238
 - // MARK: 239
- 15. Saving, Loading, and Application States 241
 - Archiving 242
 - Application Sandbox 245
 - Constructing a file URL 247
 - NSKeyedArchiver and NSKeyedUnarchiver 248
 - Loading files 251

Application States and Transitions	252
Writing to the Filesystem with NSData	254
Error Handling	257
Bronze Challenge: PNG	259
For the More Curious: Application State Transitions	260
For the More Curious: Reading and Writing to the Filesystem	261
For the More Curious: The Application Bundle	263
16. Size Classes	265
Another Size Class	266
Bronze Challenge: Stacked Text Field and Labels	272
17. Touch Events and UIResponder	273
Touch Events	274
Creating the TouchTracker Application	275
Creating the Line Struct	276
Structs	277
Value types vs. reference types	277
Creating DrawView	278
Drawing with DrawView	279
Turning Touches into Lines	280
Handling multiple touches	281
@IBInspectable	286
Silver Challenge: Colors	287
Gold Challenge: Circles	287
For the More Curious: The Responder Chain	288
For the More Curious: UIControl	289
18. UIGestureRecognizer and UIMenuController	291
UIGestureRecognizer Subclasses	292
Detecting Taps with UITapGestureRecognizer	292
Multiple Gesture Recognizers	294
UIMenuController	297
More Gesture Recognizers	299
UILongPressGestureRecognizer	299
UIPanGestureRecognizer and simultaneous recognizers	300
More on UIGestureRecognizer	304
Silver Challenge: Mysterious Lines	305
Gold Challenge: Speed and Size	305
Platinum Challenge: Colors	305
For the More Curious: UIMenuController and UIResponderStandardEditActions	306
19. Web Services	307
Starting the Photorama Application	308
Building the URL	309
Formatting URLs and requests	309
NSURLComponents	310
Sending the Request	314
NSURLSession	314
Modeling the Photo	317
JSON Data	318
NSJSONSerialization	319

- Enumerations and associated values 320
- Parsing JSON data 321
- Downloading and Displaying the Image Data 327
- The Main Thread 330
- Bronze Challenge: Printing the Response Information 331
- For the More Curious: HTTP 331
- 20. Collection Views 333
 - Displaying the Grid 334
 - Collection View Data Source 335
 - Customizing the Layout 338
 - Creating a Custom UICollectionViewCell 341
 - Downloading the Image Data 345
 - Extensions 347
 - Navigating to a Photo 350
 - Silver Challenge: Updated Item Sizes 352
 - Gold Challenge: Creating a Custom Layout 352
- 21. Core Data 353
 - Object Graphs 353
 - Entities 353
 - Modeling entities 354
 - Transformable attributes 356
 - NSManagedObject and subclasses 356
 - Building the Core Data Stack 358
 - NSManagedObjectModel 359
 - NSPersistentStoreCoordinator 359
 - NSManagedObjectContext 360
 - Updating Items 361
 - Inserting into the context 361
 - Saving changes 363
 - Updating the Data Source 364
 - Fetch requests and predicates 364
 - Saving Images to Disk 367
 - Bronze Challenge: Photo View Count 368
- 22. Core Data Relationships 369
 - Relationships 370
 - Adding Tags to the Interface 372
 - Parent-Child Contexts 382
 - Silver Challenge: Favorites 384
- 23. Afterword 385
 - What to Do Next 385
 - Shameless Plugs 385
- Index 387

Introduction

As an aspiring iOS developer, you face three major tasks:

- *You must learn the Swift language.* Swift is the recommended development language for iOS. The first two chapters of this book are designed to give you a working knowledge of Swift.
- *You must master the big ideas.* These include things like delegation, archiving, and the proper use of view controllers. The big ideas take a few days to understand. When you reach the halfway point of this book, you will understand these big ideas.
- *You must master the frameworks.* The eventual goal is to know how to use every method of every class in every framework in iOS. This is a project for a lifetime: there are hundreds of classes and thousands of methods available in iOS, and Apple adds more classes and methods with every release of iOS. In this book, you will be introduced to each of the subsystems that make up the iOS SDK, but you will not study each one deeply. Instead, our goal is to get you to the point where you can search and understand Apple's reference documentation.

We have used this material many times at our iOS bootcamps at Big Nerd Ranch. It is well tested and has helped thousands of people become iOS developers. We sincerely hope that it proves useful to you.

Prerequisites

This book assumes that you are already motivated to learn to write iOS apps. We will not spend any time convincing you that the iPhone, iPad, and iPod touch are compelling pieces of technology.

We also assume that you have some experience programming and know something about object-oriented programming. If this is not true, you should probably start with *Swift Programming: The Big Nerd Ranch Guide*.

What Has Changed in the Fifth Edition?

All of the code in this book is Swift, and an early chapter is devoted to getting you up to speed with this new language. Throughout the book, you will see how to use Swift's capabilities and features to write better iOS applications. We have come to love Swift at Big Nerd Ranch and believe you will, too.

Other additions include collection views and size classes and improved coverage of Auto Layout, web services, and Core Data.

This edition assumes that the reader is using Xcode 7.1 or later and running applications on an iOS 9 or later device.

Besides these obvious changes, we made thousands of tiny improvements that were inspired by questions from our readers and our students. Every chapter of this book is just a little better than the corresponding chapter from the fourth edition.

Our Teaching Philosophy

This book will teach you the essential concepts of iOS programming. At the same time, you will type in a lot of code and build a bunch of applications. By the end of the book, you will have knowledge *and* experience. However, all the knowledge should not (and, in this book, will not) come first. That is the traditional way we have all come to know and hate. Instead, we take a learn-while-doing approach. Development concepts and actual coding go together.

Here is what we have learned over the years of teaching iOS programming:

- We have learned what ideas people must grasp to get started programming, and we focus on that subset.
- We have learned that people learn best when these concepts are introduced *as they are needed*.
- We have learned that programming knowledge and experience grow best when they grow together.
- We have learned that “going through the motions” is much more important than it sounds. Many times we will ask you to start typing in code before you understand it. We realize that you may feel like a trained monkey typing in a bunch of code that you do not fully grasp. But the best way to learn coding is to find and fix your typos. Far from being a drag, this basic debugging is where you really learn the ins and outs of the code. That is why we encourage you to type in the code yourself. You could just download it, but copying and pasting is not programming. We want better for you and your skills.

What does this mean for you, the reader? To learn this way takes some trust – and we appreciate yours. It also takes patience. As we lead you through these chapters, we will try to keep you comfortable and tell you what is happening. However, there will be times when you will have to take our word for it. (If you think this will bug you, keep reading – we have some ideas that might help.) Do not get discouraged if you run across a concept that you do not understand right away. Remember that we are intentionally *not* providing all the knowledge you will ever need all at once. If a concept seems unclear, we will likely discuss it in more detail later when it becomes necessary. And some things that are not clear at the beginning will suddenly make sense when you implement them the first (or the twelfth) time.

People learn differently. It is possible that you will love how we hand out concepts on an as-needed basis. It is also possible that you will find it frustrating. In case of the latter, here are some options:

- Take a deep breath and wait it out. We will get there, and so will you.
- Check the index. We will let it slide if you look ahead and read through a more advanced discussion that occurs later in the book.
- Check the online Apple documentation. This is an essential developer tool, and you will want plenty of practice using it. Consult it early and often.
- If Swift or object-oriented programming concepts are giving you a hard time (or if you think they will), you might consider backing up and reading our *Swift Programming: The Big Nerd Ranch Guide*.

How to Use This Book

This book is based on the class we teach at Big Nerd Ranch. As such, it was designed to be consumed in a certain manner.

Set yourself a reasonable goal, like “I will do one chapter every day.” When you sit down to attack a chapter, find a quiet place where you will not be interrupted for at least an hour. Shut down your email, your Twitter client, and your chat program. This is not a time for multitasking; you will need to concentrate.

Do the actual programming. You can read through a chapter first, if you like. But the real learning comes when you sit down and code as you go. You will not really understand the idea until you have written a program that uses it and, perhaps more importantly, debugged that program.

A couple of the exercises require supporting files. For example, in the first chapter you will need an icon for your Quiz application, and we have one for you. You can download the resources and solutions to the exercises from [here](#).

There are two types of learning. When you learn about the Peloponnesian War, you are simply adding details to a scaffolding of ideas that you already understand. This is what we will call “Easy Learning.” Yes, learning about the Peloponnesian War can take a long time, but you are seldom flummoxed by it. Learning iOS programming, on the other hand, is “Hard Learning,” and you may find yourself quite baffled at times, especially in the first few days. In writing this book, we have tried to create an experience that will ease you over the bumps in the learning curve. Here are two things you can do to make the journey easier:

- Find someone who already knows how to write iOS applications and will answer your questions. In particular, getting your application onto a device the first time is usually very frustrating if you are doing it without the help of an experienced developer.
- Get enough sleep. Sleepy people do not remember what they have learned.

How This Book Is Organized

In this book, each chapter addresses one or more ideas of iOS development through discussion and hands-on practice. For more coding practice, most chapters include challenge exercises. We encourage you to take on at least some of these. They are excellent for firming up your grasp of the concepts introduced in the chapter and for making you a more confident iOS programmer. Finally, most chapters conclude with one or two “For the More Curious” sections that explain certain consequences of the concepts that were introduced earlier.

Chapter 1 introduces you to iOS programming as you build and deploy a tiny application. You will get your feet wet with Xcode and the iOS simulator along with all the steps for creating projects and files. The chapter includes a discussion of Model-View-Controller and how it relates to iOS development.

Chapter 2 provides an overview of Swift, including basic syntax, types, optionals, initialization, and how Swift is able to interact with the existing iOS frameworks. You will also get experience working in a playground, Xcode’s new code prototyping tool.

In Chapter 3, you will focus on the iOS user interface as you learn about views and the view hierarchy and create an application called WorldTrotter.

Chapter 4 introduces delegation, an important iOS design pattern. You will also add a text field to `WorldTrotter`.

In Chapter 5, you will expand `WorldTrotter` and learn about using view controllers for managing user interfaces. You will get practice working with views and view controllers as well as navigating between screens using a tab bar.

In Chapter 6, you will learn how to manage views and view controllers in code. You will add a segmented control to `WorldTrotter` that will let you switch between various map types.

Chapter 7 introduces the concepts and techniques of internationalization and localization. You will learn about `NSLocale`, strings tables, and `NSBundle` as you localize parts of `WorldTrotter`.

In Chapter 8, you will learn about and add different types of animations to the Quiz project that you created in Chapter 1.

Chapter 9 introduces the largest application in the book – `Homepwner`. (By the way, “Homepwner” is not a typo; you can find the definition of “pwn” at www.urbandictionary.com.) This application keeps a record of your items in case of fire or other catastrophe. `Homepwner` will take eight chapters to complete.

In Chapter 9 - Chapter 11, you will work with tables. You will learn about table views, their view controllers, and their data sources. You will learn how to display data in a table, how to allow the user to edit the table, and how to improve the interface.

Chapter 12 introduces stack views that will help you create complex interfaces very easily. You will use a stack view to add a new screen to `Homepwner` that displays the details for a single item.

Chapter 13 builds on the navigation experience gained in Chapter 5. You will use `UINavigationController` to give `Homepwner` a drill-down interface and a navigation bar.

Chapter 14 introduces the camera. You will take pictures and display and store images in `Homepwner`.

In Chapter 15, you will add persistence to `Homepwner` using archiving to save and load the application data.

In Chapter 16, you will learn about size classes, and you will use these to update `Homepwner`'s interface to scale well across various screen sizes.

In Chapter 17 and Chapter 18, you will create a drawing application named `TouchTracker` to learn about touch events. You will see how to add multitouch capability and how to use `UIGestureRecognizer` to respond to particular gestures. You will also get experience with the first responder and responder chain concepts and more practice with using structures and dictionaries.

Chapter 19 introduces web services as you create the `Photorama` application. This application fetches and parses JSON from a server using `NSURLSessions` and `NSJSONSerialization`.

In Chapter 20, you will learn about collection views as you build an interface for `Photorama` using `UICollectionView` and `UICollectionViewCell`.

In Chapter 21 and Chapter 22, you will add persistence to `Photorama` using Core Data. You will store and load images and associated data using an `NSManagedObjectContext`.

Style Choices

This book contains a lot of code. We have attempted to make that code and the designs behind it exemplary. We have done our best to follow the idioms of the community, but at times we have wandered from what you might see in Apple’s sample code or code you might find in other books. In particular, you should know up-front that we nearly always start a project with the simplest template project: the single view application. When your app works, you will know it is because of your efforts – not because that behavior was built into the template.

Typographical Conventions

To make this book easier to read, certain items appear in certain fonts. Classes, types, methods, and functions appear in a bold, fixed-width font. Classes and types start with capital letters, and methods and functions start with lowercase letters. For example, “In the **loadView()** method of the **RexViewController** class, create a constant of type **String**.”

Variables, constants, and filenames appear in a fixed-width font but are not bold. So you will see, “In `ViewController.swift`, add a variable named `fido` and initialize it to “Rufus”.”

Application names, menu choices, and button names appear in a sans serif font. For example, “Open Xcode and select New Project... from the File menu. Select Single View Application and then click Choose....”

All code blocks are in a fixed-width font. Code that you need to type in is always bold. For example, in the following code, you would type in the two lines beginning **@IBOutlet**. The other lines are already in the code and are included to let you know where to add the new lines.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet var questionLabel: UILabel!
    @IBOutlet var answerLabel: UILabel!

}
```

Necessary Hardware and Software

To build the applications in this book, you must have a Mac running OS X Yosemite (10.10.5) or later. You will also need Xcode, Apple’s Integrated Development Environment, which is available on the App Store. Xcode includes the iOS SDK, the iOS simulator, and other development tools.

You should join the Apple Developer Program, which costs \$99/year, because:

- Downloading the latest developer tools is free for members.
- You cannot put an app in the store until you are a member.

If you are going to take the time to work through this entire book, membership in the Apple Developer Program is worth the cost. Go to <http://developer.apple.com/programs/ios/> to join.

What about iOS devices? Most of the applications you will develop in the first half of the book are for iPhone, but you will be able to run them on an iPad. On the iPad screen, iPhone applications appear in an iPhone-sized window. Not a compelling use of iPad, but that is OK when you are starting with iOS. In the early chapters, you will be focused on learning the fundamentals of the iOS SDK, and these are the same across iOS devices. Later in the book, you will see how to make applications run natively on both iOS device families.

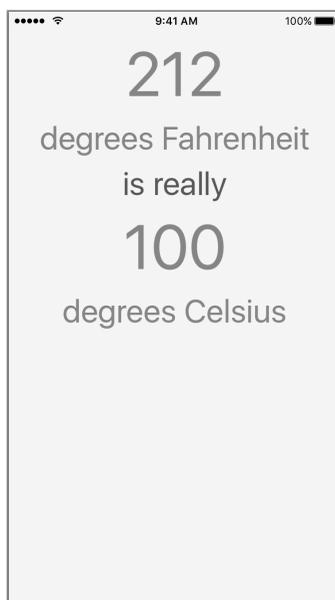
Excited yet? Good. Let's get started.

3

Views and the View Hierarchy

Over the next five chapters, you are going to build an application named WorldTrotter. When it is complete, this app will convert values between degrees Fahrenheit and degrees Celsius. In this chapter, you will learn about views and the view hierarchy through creating WorldTrotter's user interface. At the end of this chapter, your app will look like Figure 3.1.

Figure 3.1 WorldTrotter



Let's start with a little bit of the theory behind views and the view hierarchy.

View Basics

Recall from Chapter 1 that views are objects that are visible to the user, like buttons, text fields, and sliders. View objects make up an application’s user interface. A view

- is an instance of **UIView** or one of its subclasses
- knows how to draw itself
- can handle events, like touches
- exists within a hierarchy of views whose root is the application’s window

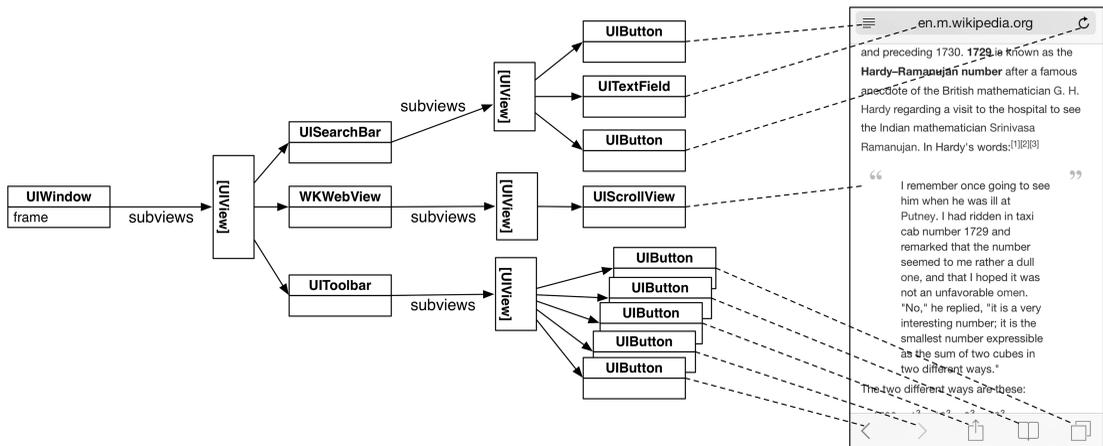
Let’s look at the *view hierarchy* in greater detail.

The View Hierarchy

Every application has a single instance of **UIWindow** that serves as the container for all the views in the application. **UIWindow** is a subclass of **UIView**, so the window is itself a view. The window is created when the application launches. Once the window is created, other views can be added to it.

When a view is added to the window, it is said to be a *subview* of the window. Views that are subviews of the window can also have subviews, and the result is a hierarchy of view objects with the window at its root (Figure 3.2).

Figure 3.2 An example view hierarchy and the interface that it creates

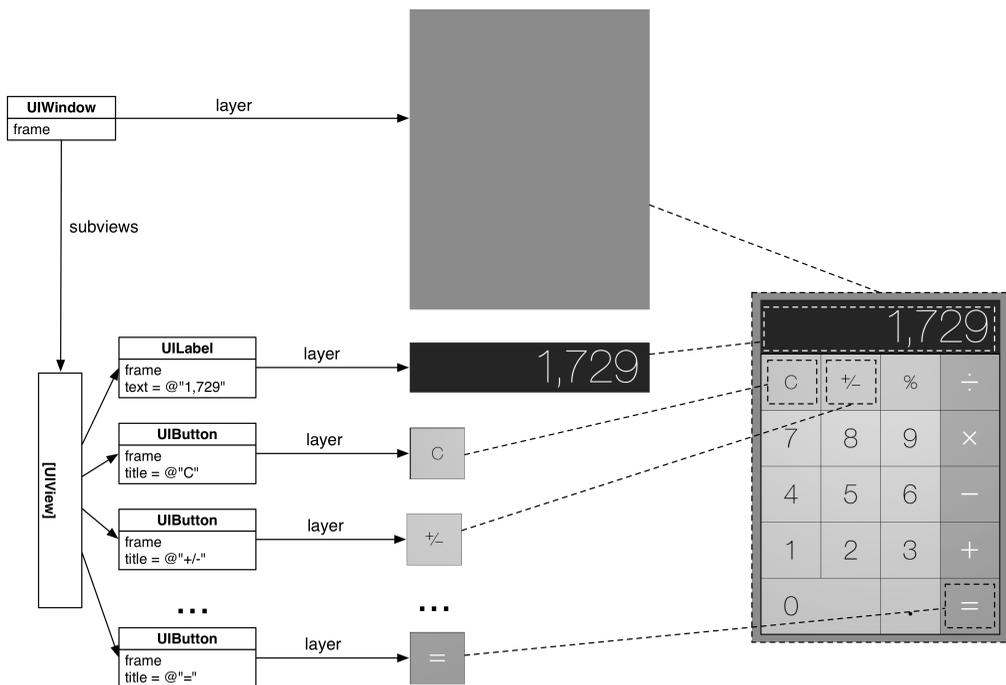


Once the view hierarchy is created, it will be drawn to the screen. This process can be broken into two steps:

- Each view in the hierarchy, including the window, draws itself. It renders itself to its *layer*, which you can think of as a bitmap image. (The layer is an instance of **CALayer**.)
- The layers of all the views are composited together on the screen.

Figure 3.3 shows another example view hierarchy and the two drawing steps.

Figure 3.3 Views render themselves and then are composited together

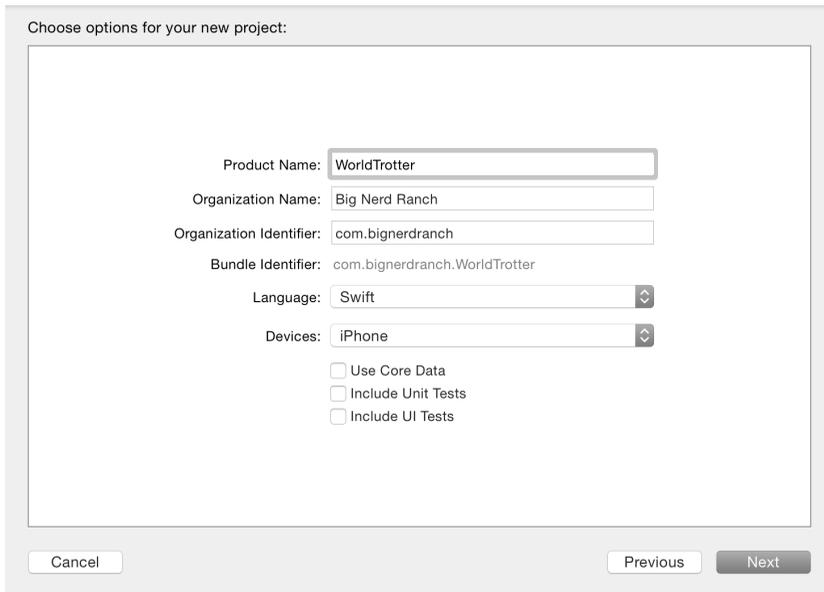


Creating a New Project

In Xcode, select File → New → Project... (or use the keyboard shortcut Command-Shift-N). From the iOS section, select Application, choose the Single View Application template, and click Next.

Enter WorldTrotter for the product name. Make sure that Swift is selected from the Language dropdown and that iPhone is selected from the Devices dropdown. Also make sure the Use Core Data box is unchecked (Figure 3.4). Click Next and then Create on the following screen.

Figure 3.4 Configuring WorldTrotter



Views and Frames

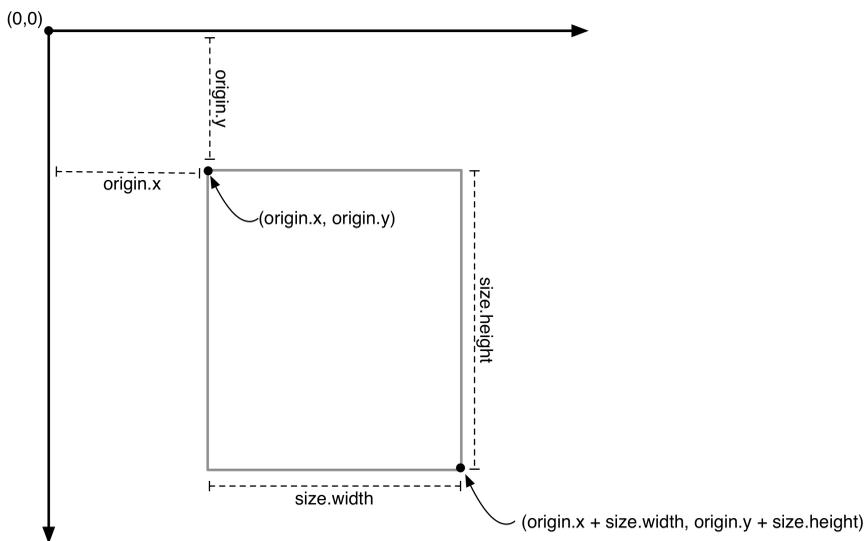
When you initialize a view programmatically, you use its `init(frame:)` designated initializer. This method takes one argument, a `CGRect`, that will become the view's frame, a property on `UIView`.

```
var frame: CGRect
```

A view's frame specifies the view's size and its position relative to its superview. Because a view's size is always specified by its frame, a view is always a rectangle.

A `CGRect` contains the members `origin` and `size`. The `origin` is a structure of type `CGPoint` and contains two `CGFloat` properties: `x` and `y`. The `size` is a structure of type `CGSize` and has two `CGFloat` properties: `width` and `height` (Figure 3.5).

Figure 3.5 CGRect



When the application is launched, the view for the initial view controller is added to the root-level window. This view controller is represented by the `ViewController` class defined in `ViewController.swift`. We will discuss what a view controller is in Chapter 5, but for now, it is sufficient to know that a view controller has a view and that the view associated with the main view controller for the application is added as a subview of the window.

Before you create the views for `WorldTrotter`, you are going to add some practice views programmatically to explore views and their properties and see how the interfaces for applications are created.

Open `ViewController.swift` and delete any methods that the template created. Your file should look like this:

```
import UIKit

class ViewController: UIViewController {
}
```

(Curious about the `import UIKit` line? `UIKit` is a *framework*. A framework is a collection of related classes and resources. The `UIKit` framework defines many of the user interface elements that your users see, as well as other iOS-specific classes. You will be using a few different frameworks as you go through this book.)

Right after the view controller's view is loaded into memory, its `viewDidLoad()` method is called. This method gives you an opportunity to customize the view hierarchy, so it is a great place to add your practice views.

In `ViewController.swift`, override `viewDidLoad()`. Create a `CGRect` that will be the frame of a `UIView`. Next, create an instance of `UIView` and set its `backgroundColor` property to blue. Finally, add the `UIView` as a subview of the view controller's view to make it part of the view hierarchy. (Much of this will not look familiar. That is fine. We will explain more after you enter the code.)

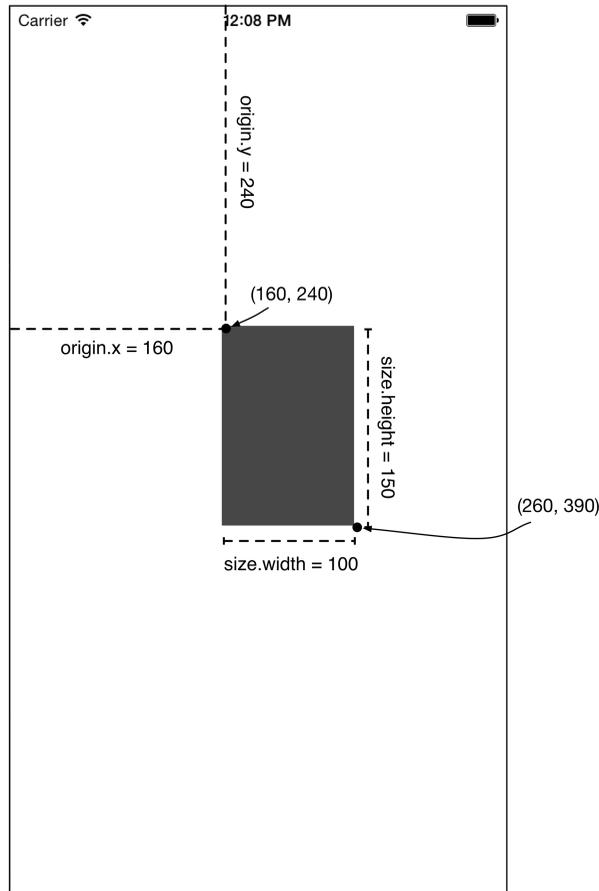
```
class ViewController: UIViewController {  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)  
        let firstView = UIView(frame: firstFrame)  
        firstView.backgroundColor = UIColor.blueColor()  
        view.addSubview(firstView)  
    }  
  
}
```

To create a `CGRect`, you use its initializer and pass in the values for `origin.x`, `origin.y`, `size.width`, and `size.height`.

To set the `backgroundColor`, you use the `UIColor` class method `blueColor()`. This is a convenience method that initializes an instance of `UIColor` that is configured to be blue. There are a number of `UIColor` convenience methods for common colors, such as `greenColor()`, `blackColor()`, and `clearColor()`.

Build and run the application (Command-R). You will see a blue rectangle that is the instance of **UIView**. Because the origin of the **UIView**'s frame is (160, 240), the rectangle's top left corner is 160 points to the right and 240 points down from the top left corner of its superview. The view stretches 100 points to the right and 150 points down from its origin, in accordance with its frame's size (Figure 3.6).

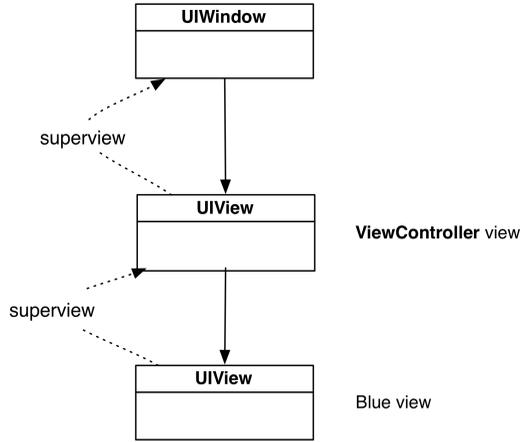
Figure 3.6 WorldTrotter with one **UIView**



Note that these values are in points, not pixels. If the values were in pixels, then they would not be consistent across displays of different resolutions (i.e., Retina vs. non-Retina). A single point is a relative unit of a measure; it will be a different number of pixels depending on how many pixels are in the display. Sizes, positions, lines, and curves are always described in points to allow for differences in display resolution.

Figure 3.7 represents the view hierarchy that you have created.

Figure 3.7 Current view hierarchy



Every instance of **UIView** has a `superview` property. When you add a view as a subview of another view, the inverse relationship is automatically established. In this case, the **UIView**'s `superview` is the **UIWindow**.

Let's experiment with the view hierarchy. First, in `ViewController.swift`, create another instance of **UIView** with a different frame and background color.

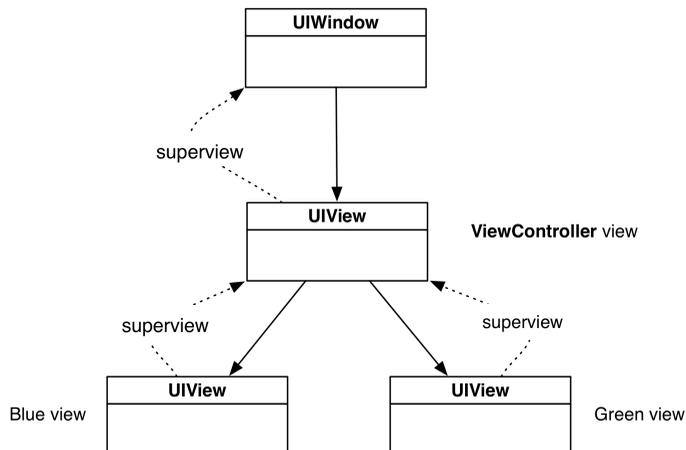
```
override func viewDidLoad() {
    super.viewDidLoad()

    let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)
    let firstView = UIView(frame: firstFrame)
    firstView.backgroundColor = UIColor.blueColor()
    view.addSubview(firstView)

    let secondFrame = CGRect(x: 20, y: 30, width: 50, height: 50)
    let secondView = UIView(frame: secondFrame)
    secondView.backgroundColor = UIColor.greenColor()
    view.addSubview(secondView)
}
```

Build and run again. In addition to the blue rectangle, you will see a green square near the top lefthand corner of the window. Figure 3.8 shows the updated view hierarchy.

Figure 3.8 Updated view hierarchy with two subviews as siblings



Now you are going to adjust the view hierarchy so that one instance of **UIView** is a subview of the other **UIView** instead of the view controller's view. In `ViewController.swift`, add `secondView` as a subview of `firstView`.

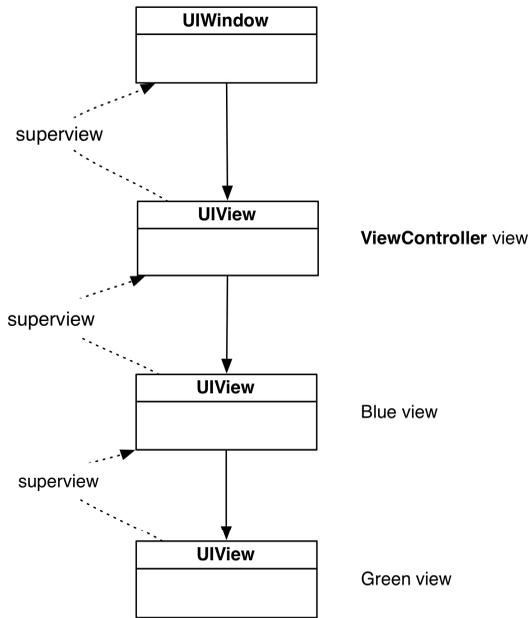
...

```
let secondView = UIView(frame: secondFrame)
secondView.backgroundColor = UIColor.greenColor()
```

```
view.addSubview(secondView)
firstView.addSubview(secondView)
```

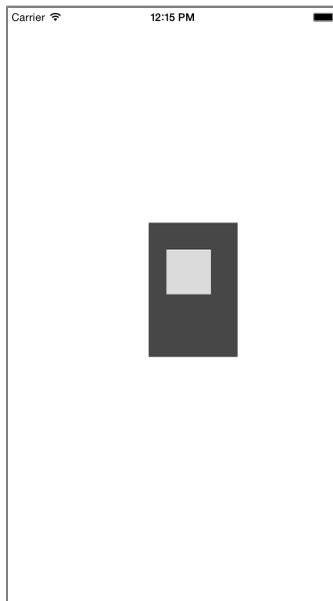
Your view hierarchy is now four levels deep, as shown in Figure 3.9.

Figure 3.9 One **UIView** as a subview of the other



Build and run the application. Notice that `secondView`'s position on the screen has changed (Figure 3.10). A view's frame is relative to its superview, so the top left corner of `secondView` is now inset (20, 30) points from the top left corner of `firstView`.

Figure 3.10 WorldTrotter with new hierarchy



(If the green instance of `UIView` looks smaller than it did previously, that is just an optical illusion. Its size has not changed.)

Now that you have seen the basics of views and the view hierarchy, you can start working on the interface for `WorldTrotter`. Instead of building up the interface programmatically, you will use Interface Builder to visually lay out the interface, as you did in Chapter 1.

In `ViewController.swift`, start by removing your practice code.

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)  
    let firstView = UIView(frame: firstFrame)  
    firstView.backgroundColor = UIColor.blueColor()  
    view.addSubview(firstView)  
  
    let secondFrame = CGRect(x: 20, y: 30, width: 50, height: 50)  
    let secondView = UIView(frame: secondFrame)  
    secondView.backgroundColor = UIColor.greenColor()  
    firstView.addSubview(secondView)  
}
```

Now let's add some views to the interface and set their frames.

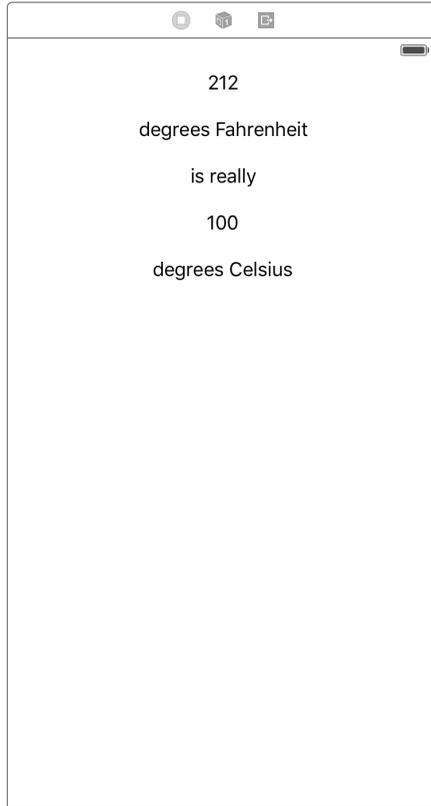
Open `Main.storyboard`. Notice that the interface on the screen is currently a square. While you explore views and their frames, it will be nice to have the size of the interface in Xcode match the screen size of the device that you will be using.

Select the View Controller either in the document outline or by clicking the yellow circle above the interface. Open the *attributes inspector*, which is the fourth tab in the utilities area. You can quickly open this pane using the keyboard shortcut `Command-Option-4`.

At the top of the pane, find the section labeled *Simulated Metrics* and change the *Size* to be `iPhone 4.7-inch`. This will resize the square interface to match the dimensions of the 4.7-inch devices.

From the object library, drag five instances of **UILabel** onto the canvas. Space them out vertically on the top half of the interface and center them horizontally. Set their text to match Figure 3.11.

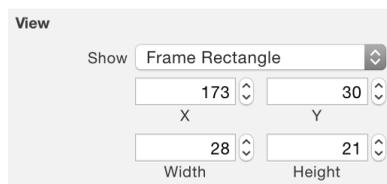
Figure 3.11 Adding labels to the interface



Select the top label so you can see its frame in Interface Builder. Open its *size inspector* – the fifth tab in the utilities area. (As you might have noticed by this point, the keyboard shortcuts for the utilities tabs are Command-Option plus the tab number. Since the size inspector is the fifth tab, its keyboard shortcut is Command-Option-5.)

Under the View section, find Frame Rectangle. (If you do not see it, you might need to select it from the Show pop-up menu.) These values are the view’s frame, and they dictate the position of the view on screen (Figure 3.12).

Figure 3.12 View frame values



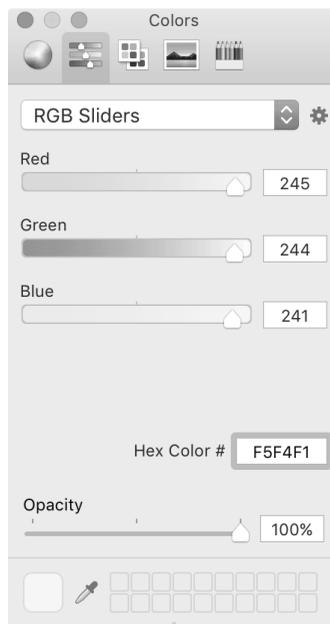
Build and run the application on the iPhone 6s simulator. This corresponds to the 4.7-inch simulated metrics that you specified in the storyboard. The interface on the simulator will look identical to the interface that you laid out in Interface Builder.

Customizing the labels

Let's make the interface look a little bit better by customizing the view properties.

In `Main.storyboard`, select the background view. Open the attributes inspector and give the app a new background color: Find and click the **Background** dropdown and click **Other**. Select the second tab (the **Color Sliders** tab) and enter a Hex Color # of `F5F4F1` (Figure 3.13). This will give the background a warm, gray color.

Figure 3.13 Changing the background color

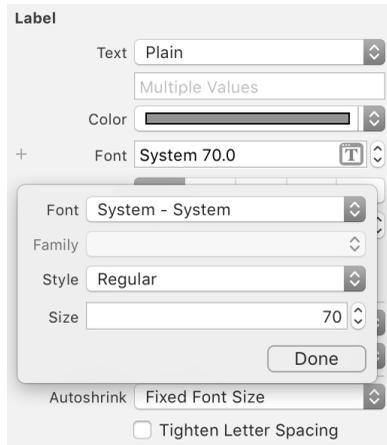


You can customize attributes common to selected views simultaneously. You will use this to give many of the labels a larger font size as well as a burnt orange text color.

Select the top two and bottom two labels by **Command**-clicking them in the document outline and open the attributes inspector. Update the text color: Under the **Label** section, find **Color** and open the pop-up menu. Select the **Color Sliders** tab again and enter a Hex Color # of `E15829`.

Now let's update the font. Select the 212 and 100 labels. Under the Label section in the attributes inspector, find Font and click on the text icon next to the current font. From the popover that appears, make the Font System - System and the Size 70 (Figure 3.14). Select the remaining three labels. Open their Font pop-up and make the Font System - System and the Size 36.

Figure 3.14 Customizing the labels' font



Now that the font size is larger, the text no longer fits within the bounds of the label. You could resize the labels manually, but there is an easier way.

Select the top label on the canvas. From Xcode's Editor menu, select Size to Fit Content (Command=). This will resize the label to exactly fit its text contents. Repeat the process for the other four labels. (You can select all four labels to resize them all at once.) Now move the labels so that they are again nicely aligned vertically and centered horizontally (Figure 3.15).

Figure 3.15 Updating the label frames



Build and run the application on the iPhone 6s simulator. Now build and run the application on the iPhone 6s Plus simulator. Notice that the labels are no longer centered – instead, they appear shifted slightly to the left.

You have just seen two of the major problems with absolute frames. First, when the contents change (like when you changed the font size), the frames do not automatically update. Second, the view does not look equally good on different sizes of screens.

In general, you should not use absolute frames for your views. Instead, you should use Auto Layout to flexibly compute the frames for you based on constraints that you specify for each view. For example, what you really want for *WorldTrotter* is for the labels to remain the same distance from the top of the screen and to remain horizontally centered within their superview. They should also update if the font or text of the labels change. This is what you will accomplish in the next section.

The Auto Layout System

Before you can fix the labels to have them lay out flexibly, you need to learn a little theory about the Auto Layout system.

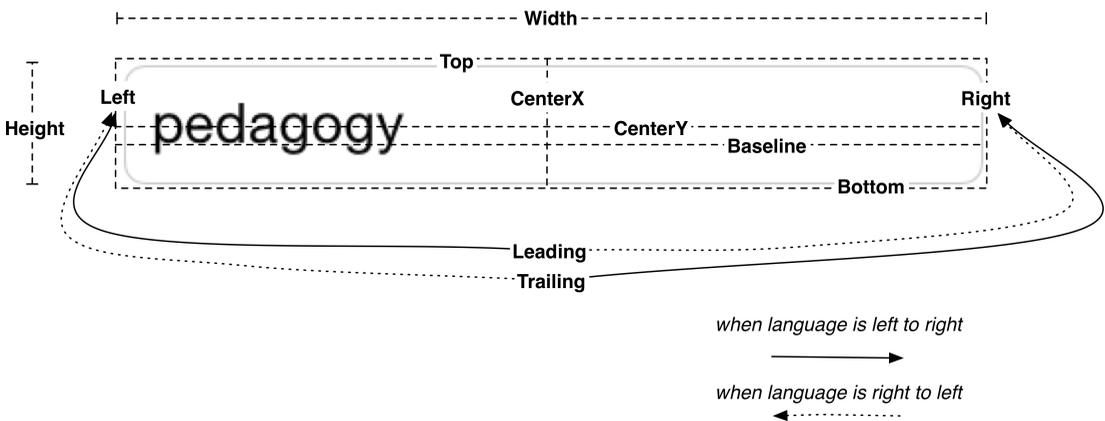
As you saw in Chapter 1, absolute coordinates make your layout fragile because they assume that you know the size of the screen ahead of time.

Using Auto Layout, you can describe the layout of your views in a relative way that enables their frames to be determined at runtime so that the frames' definitions can take into account the screen size of the device that the application is running on.

Alignment rectangle and layout attributes

The Auto Layout system is based on the *alignment rectangle*. This rectangle is defined by several *layout attributes* (Figure 3.16).

Figure 3.16 Layout attributes defining an alignment rectangle of a view



Width/Height	These values determine the alignment rectangle's size.
Top/Bottom/Left/Right	These values determine the spacing between the given edge of the alignment rectangle and the alignment rectangle of another view in the hierarchy.
CenterX/CenterY	These values determine the center point of the alignment rectangle.
Baseline	This value is the same as the bottom attribute for most, but not all, views. For example, UITextField defines its baseline as the bottom of the text it displays rather than the bottom of the alignment rectangle. This keeps "descenders" (letters like 'g' and 'p' that descend below the baseline) from being obscured by a view right below the text field.
Leading/Trailing	These values are language-specific attributes. If the device is set to a language that reads left to right (e.g., English), then the leading attribute is the same as the left attribute and the trailing attribute is the same as the right attribute. If the language reads right to left (e.g., Arabic), then the leading attribute is on the right and the trailing attribute is on the left. Interface Builder automatically prefers leading and trailing over left and right, and, in general, you should as well.

By default, every view has an alignment rectangle, and every view hierarchy uses Auto Layout.

The alignment rectangle is very similar to the frame. In fact, these two rectangles are often the same. Whereas the frame encompasses the entire view, the alignment rectangle only encompasses the content that you wish to use for alignment purposes. Figure 3.17 shows an example where the frame and the alignment rectangle are different.

Figure 3.17 Frame vs. alignment rectangle



You cannot define a view's alignment rectangle directly. You do not have enough information (like screen size) to do that. Instead, you provide a set of *constraints*. Taken together, these constraints enable the system to determine the layout attributes, and thus the alignment rectangle, for each view in the view hierarchy.

Constraints

A *constraint* defines a specific relationship in a view hierarchy that can be used to determine a layout attribute for one or more views. For example, you might add a constraint like, “The vertical space between these two views should always be 8 points,” or, “These views must always have the same width.” A constraint can also be used to give a view a fixed size, like, “This view's height should always be 44 points.”

You do not need a constraint for every layout attribute. Some values may come directly from a constraint; others will be computed by the values of related layout attributes. For example, if a view's constraints set its left edge and its width, then the right edge is already determined (left edge + width = right edge, always). As a general rule of thumb, you need at least two constraints per dimension (horizontal and vertical).

If, after all of the constraints have been considered, there is still an ambiguous or missing value for a layout attribute, then there will be errors and warnings from Auto Layout and your interface will not look as you expect on all devices. Debugging these problems is important, and you will get some practice later in this chapter.

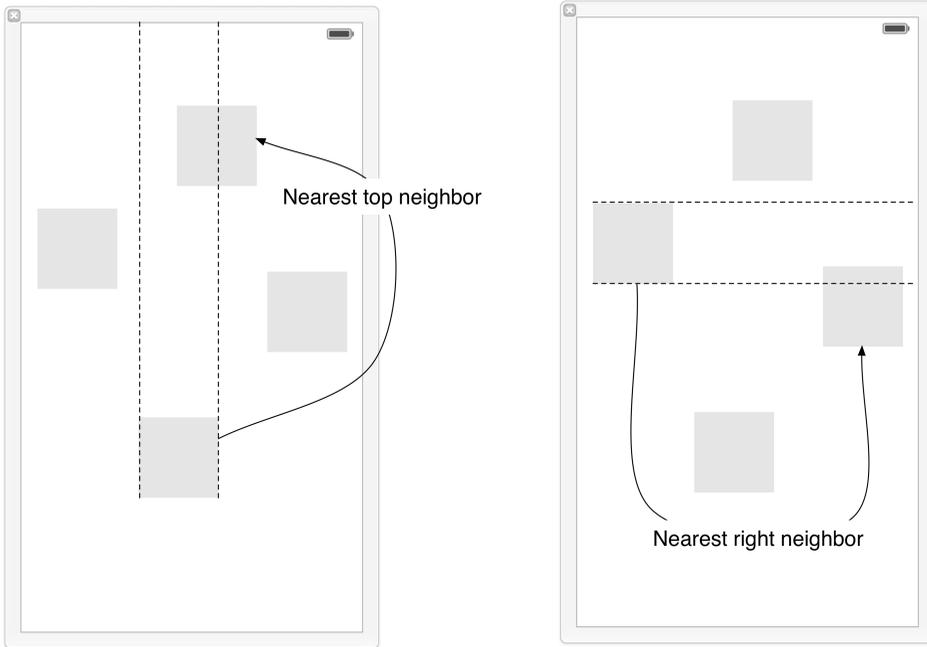
How do you come up with constraints? Let's see how using the labels that you have laid out on the canvas.

First, describe what you want the view to look like independent of screen size. For example, you might say that you want the top label to be:

- 8 points from the top of the screen
- centered horizontally in its superview
- as wide and as tall as its text

To turn this description into constraints in **Interface Builder**, it will help to understand how to find a view's *nearest neighbor*. The nearest neighbor is the closest sibling view in the specified direction (Figure 3.18).

Figure 3.18 Nearest neighbor



If a view does not have any siblings in the specified direction, then the nearest neighbor is its superview, also known as its container.

Now you can spell out the constraints for the label:

1. The label's top edge should be 8 points away from its nearest neighbor (which is its container – the view of the **ViewController**).
2. The label's center should be the same as its superview's center.
3. The label's width should be equal to the width of its text rendered at its font size.
4. The label's height should be equal to the height of its text rendered at its font size.

If you consider the first and fourth constraints, you can see that there is no need to explicitly constrain the label's bottom edge. It will be determined from the constraints on the label's top edge and the label's height. Similarly, the second and third constraints together determine the label's right and left edges.

Now that you have a plan for the top label, you can add these constraints. Constraints can be added using **Interface Builder** or in code. Apple recommends that you add constraints using **Interface Builder** whenever possible, and that is what you will do here. However, if your views are created and

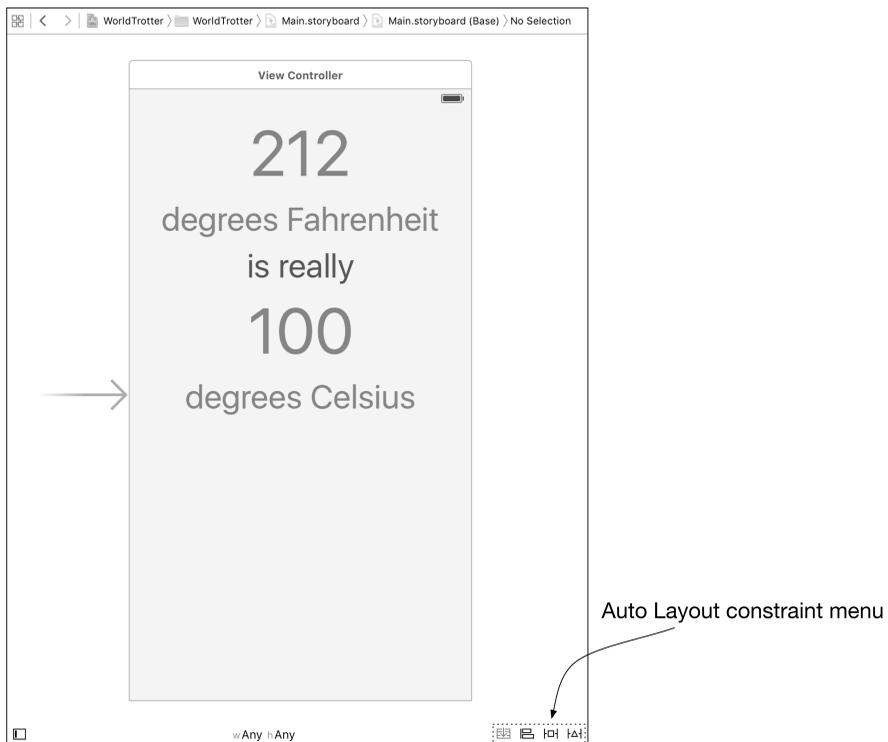
configured programmatically, then you can add constraints in code. In Chapter 6, you will practice that approach.

Adding constraints in Interface Builder

Let's get started constraining that top label.

Select the top label on the canvas. In the bottom righthand corner of the canvas, find the Auto Layout constraint menu (Figure 3.19).

Figure 3.19 Using the Auto Layout constraint menu



Click the  icon (the third from the left) to reveal the Pin menu. This menu shows you the current size and position of the label.

At the top of the Pin menu are four values that describe the label's current spacing from its nearest neighbor on the canvas. For this label, you are only interested in the top value.

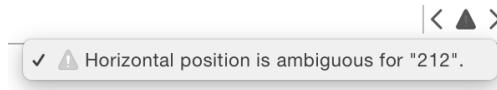
To turn this value into a constraint, click the top red strut separating the value from the square in the middle. The strut will become a solid red line.

In the middle of the menu, find the label's Width and Height. The values next to Width and Height indicate the current canvas values. To constrain the label's width and height to the current canvas values, check the boxes next to Width and Height. The button at the bottom of the menu reads Add 3 Constraints. Click this button.

At this point, you have not specified enough constraints to fully determine the alignment rectangle. Interface Builder will help you determine what the problem is.

In the top right corner of Interface Builder, notice the yellow warning sign (Figure 3.20). Click on this icon to reveal the issue: “Horizontal position is ambiguous for "212".”

Figure 3.20 Horizontal ambiguity



You have added two vertical constraints (a top edge constraint and a height constraint), but you have only added one horizontal constraint (a width constraint). Having only one constraint makes the horizontal position of the label ambiguous. You will fix this issue by adding a center alignment constraint between the label and its superview.

With the top label still selected, click the  icon (the second from the left in the Auto Layout constraints menu) to reveal the Align menu. If you have multiple views selected, this menu will allow you to align attributes among the views. Since you have only selected one label, the only options you are given are to align the view within its container.

In the Align menu, select Horizontally in Container (do not click Add 1 Constraint yet). Once you add this constraint, there will be enough constraints to fully determine the alignment rectangle. To ensure that the frame of the label matches the constraints specified, open the Update Frames pop-up menu from the Align menu and select Items of New Constraints. This will reposition the label to match the constraints that have been added. Now click on Add 1 Constraint to add the centering constraint and reposition the label.

The label’s constraints are all blue now that the alignment rectangle for the label is fully specified. Additionally, the warning at the top right corner of Interface Builder is now gone.

Build and run the application on the iPhone 6s simulator and the iPhone 6s Plus simulator. The top label will remain centered in both simulators.

Intrinsic content size

Although the top label’s position is flexible, its size is not. This is because you have added explicit width and height constraints to the label. If the text or font were to change, you would be in the same position you were in earlier. The size of the frame is absolute, so the frame would not hug to the content.

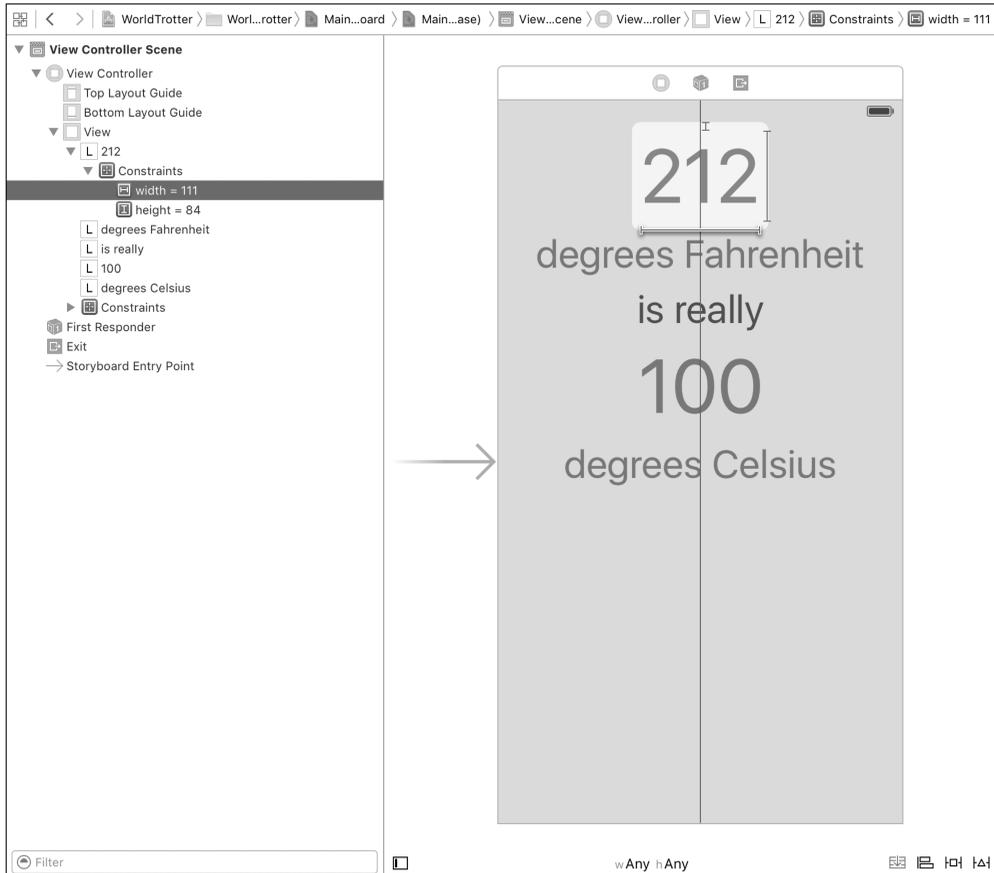
This is where the *intrinsic content size* of a view comes into play. You can think of the intrinsic content size as the size that a view “wants” to naturally be. For labels, this size is the size of the text rendered at the given font. For images, this is the size of the image itself.

A view’s intrinsic content size acts as implicit width and height constraints. If you do not specify constraints that explicitly determine the width, the view will be its intrinsic width. The same goes for the height.

With this knowledge, let the top label have a flexible size by removing the explicit width and height constraints.

In `Main.storyboard`, select the width constraint on the label. You can do this by clicking on the constraint on the canvas. Alternatively, in the document outline, you can click on the disclosure triangle next to the 212 label, then disclose the list of constraints for the label (Figure 3.21). Once you have selected the width constraint, press the Delete key. Do the same for the height constraint.

Figure 3.21 Selecting the width constraint



Notice that the constraints for the label are still blue. Since the width and height are being inferred from the label's intrinsic content size, there are still enough constraints to determine the label's alignment rectangle.

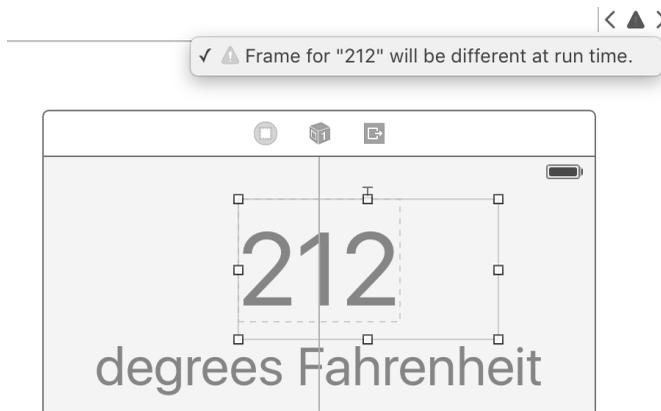
Misplaced views

As you have seen, blue constraints indicate that the alignment rectangle for a view is fully specified. Orange constraints often indicate a *misplaced view*. This means that the frame for the view in Interface Builder is different than the frame that Auto Layout has computed.

A misplaced view is very easy to fix. That is good, because it is also a very common issue that you will encounter when working with Auto Layout.

Give your top label a misplaced view so that you can see how to resolve this issue. Resize the top label on the canvas using the resize controls and look for the yellow warning in the top right corner of the canvas. Click on this warning icon to reveal the problem: “Frame for "212" will be different at run time” (Figure 3.22).

Figure 3.22 Misplaced view warning



As the warning says, the frame at runtime will not be the same as the frame specified on the canvas. If you look closely, you will see an orange dotted line that indicates what the runtime frame will be.

Build and run the application. Notice that the label is still centered despite the new frame that you gave it in Interface Builder. This might seem great – you get the result that you want, after all. But the disconnect between what you have specified in Interface Builder and the constraints computed by Auto Layout will cause problems down the line as you continue to build your views. Let’s fix the misplaced view.

Back in the storyboard, select the top label on the canvas. Click the $\text{⌘}+⌘$ icon (the right-most icon) to reveal the Resolve Auto Layout Issues menu. Select Update Frames from the Selected Views section. This will update the frame of the label to match the frame that the constraints will compute.

You will get very used to updating the frames of views as you work with Auto Layout. One word of caution: if you try to update the frames for a view that does not have enough constraints, you will almost certainly get unexpected results. If that happens, undo the change and inspect the constraints to see what is missing.

At this point, the top label is in good shape. It has enough constraints to determine its alignment rectangle, and the view is laying out the way you want.

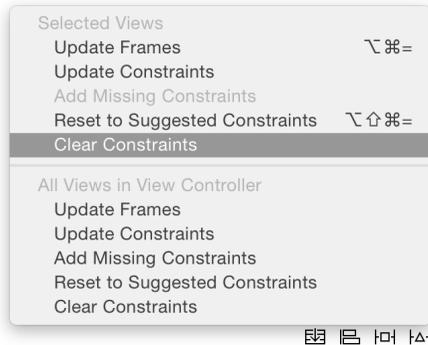
Becoming proficient with Auto Layout takes a lot of experience, so in the next section you are going to remove the constraints from the top label and then add constraints to all of the labels.

Adding more constraints

Let's flesh out the constraints for the rest of the views. Before you do that, you will first remove the existing constraints from the top label.

Select the top label on the canvas. Open the **Resolve Auto Layout Issues** menu and select **Clear Constraints** from the **Selected Views** section (Figure 3.23).

Figure 3.23 Clearing constraints



You are going to add the constraints to all of the views in two steps. First you will center the top label horizontally within the superview. Then you will add constraints that pin the top of each label to its nearest neighbor while aligning the centers of all of the labels.

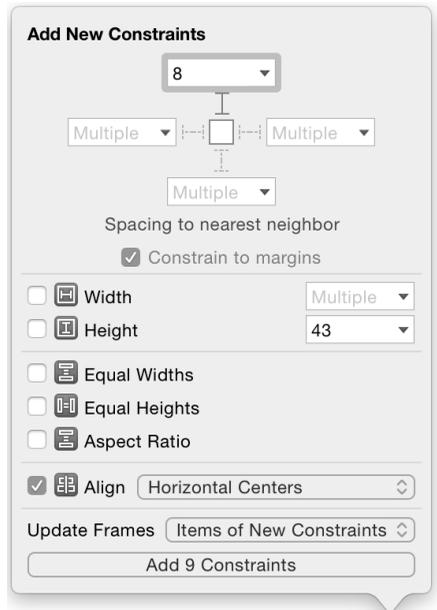
Select the top label. Open the **Align** menu and choose **Horizontally in Container** with a constant of 0. Make sure that **Update Frames** has **None** selected; remember that you do not want to update the frame of a view that does not have enough constraints, and this one constraint will certainly not provide enough information to compute the alignment rectangle. Go ahead and **Add 1 Constraint**.

Now select all five labels on the canvas. It can be very convenient to add constraints to multiple views simultaneously. Open the **Pin** menu and make the follow choices:

1. Select the top strut and make sure it has a constant of 8.
2. From the **Align** menu, choose **Horizontal Centers**.
3. From the **Update Frames** menu, choose **Items of New Constraints**.

Your menu should match Figure 3.24. Once it does, click Add 9 Constraints. This will add the constraints to the views and update their frames to reflect the Auto Layout changes.

Figure 3.24 Adding more constraints with the Pin menu



Build and run the application on the iPhone 6s simulator. The views will be centered within the interface. Now build and run the application on the iPhone 6s Plus simulator. Unlike earlier in the chapter, all of the labels remain centered on the larger interface.

Auto Layout is a crucial technology for every iOS developer. It helps you create flexible layouts that work across a range of devices and interface sizes. It also takes a lot of practice to master. You will get a lot of experience working with Auto Layout as you work through this book.

Now that the interface for WorldTrotter is using Auto Layout to adapt to various screen sizes, there is no need for you to specify an iPhone screen size when working in the storyboard.

In Main.storyboard, select the View Controller and open its attributes inspector. Find the Simulated Metrics section and change the Size to Inferred. The interface updates to be the square shape that it was initially. Notice that the labels still remain centered in this square interface due to the constraints that you added.

Designing interfaces using the inferred square shape helps to force you to think about designing adaptive interfaces that work with a variety of screen sizes instead of designing for one particular screen size.

Bronze Challenge: More Auto Layout Practice

Remove all of the constraints from the `ViewController` interface and then add them back in. Try to do this without consulting the book.

This page intentionally left blank

Index

Symbols

.xcassets (asset catalog), 24
.xcdatamodeld (data model file), 354
// MARK:, 239, 240
@IBInspectable, 286

A

access control, 311
accessory indicator (**UITableViewCell**), 153
action methods
 connecting in interface file, 226
 defining, 18
 implementing, 22
 and **UIControl**, 289
active state, 252
addSubview(_:), 49
alerts, displaying, 170-173
alignment rectangles, 56, 57
anchors, 99
animateWithDuration:animations:, 130-132
animations
 animating constraints, 135-139
 basic, 130-132
 marking completion of, 135
 spring-like, 141
 timing functions, 139, 140
anti-aliasing, 93
API Reference, 244
append(_:), 34
application bundle
 explained, 263, 264
 and internationalization, 113, 127
application sandbox, 245-247, 263
application states, 252-254, 260, 261
applicationDidBecomeActive:, 254, 260
applicationDidEnterBackground(_:), 248, 260
applicationDidEnterBackground:, 254
applications
 (see also application bundle, debugging, projects)
 building, 12, 123
 cleaning, 123
 data storage, 246, 247
 directories in, 246, 247

 icons for, 24, 25
 launch images for, 26
 multiple threads in, 330
 running on iPad, 3
 running on simulator, 12
applicationWillEnterForeground:, 254, 260
applicationWillResignActive:, 254, 260
archiving
 vs. Core Data, 353
 described, 242
 implementing, 242-245
 with **NSKeyedArchiver**, 248-251
arrays
 about, 31, 32
 append(_:), 34
 count, 34
 reverse(), 34
 subscripting, 32
 and traps, 32
 writing to filesystem, 261
asset catalogs (Xcode), 24
assistant editor (Xcode), 226
attributes (Core Data), 354
Auto Layout
 (see also constraints, Interface Builder)
 alignment rectangles, 56, 57
 autoresizing masks and, 107, 108
 dynamic cell heights, 180
 introduction to, 13-15
 layout attributes, 56, 57
 purpose of, 55
autoresizing masks, 98, 107, 108
awakeFromInsert, 358

B

background state, 252-254, 260, 261
Base internationalization, 113
baselines, 56
basic animations, 130-132
becomeFirstResponder, 72
Bool, 31
boolean types, 31
bundles
 application (see application bundle)
 NSBundle, 127, 263
buttons
 adding to navigation bars, 218

camera, 224

C

CALayer, 43

callbacks, 76

(see also delegation, target-action pairs)

camera

(see also images)

taking pictures, 224-230

`cancelTouchesInView`, 300

`canPerformAction(_:withSender:)`, 306

cells

(see also **UITableViewCell**)

adding padding to, 158

changing cell class, 177

customizing layout of, 338

dynamic cell heights, 180

prototype, 157, 342

reusing, 158

CGPoint, 45

CGRect, 45-47

CGSize, 45

closures, 130-132, 247

collection view

customizing layout of, 338

displaying, 334

downloading image data, 345-347

layout object, 333

navigating to/displaying photos, 350-352

setting data source, 335-338

colors

background, 46, 335

customizing, 53

common ancestor, 100

concurrency, 330

conditionals

if-let, 36

switch, 39

connections (in Interface Builder), 16-20

connections inspector, 20

console

printing to, 78

viewing in playground, 35

constants, 29

constraints (Auto Layout)

activating programmatically, 100, 101

adding to labels, 116

animating, 135-139

clearing, 15, 63

collection view, 334

creating explicit constraints, 103, 104

creating in Interface Builder, 57-64

creating programmatically, 98-104

implicit, 188

nearest neighbor and, 58

overview, 57

pin, 59, 60

resolving unsatisfiable, 107

specifying, 13

**constraintWithItem:attribute:relatedBy...
...: toItem:attribute:multiplier:constant:**,
103, 104

content compression, 190

`contentMode (UIImageView)`, 222, 223

`contentView (UITableViewCell)`, 154

control events, 289

controller objects, 5

controls, programmatic, 104

Core Data

vs. archiving, 353

attributes, 354

fetch requests, 364

persistent store formats, 359

relationship management with, 369-384

role of, 354

subclassing **NSManagedObject**, 356-358

transforming values, 356

Core Graphics, 93

count (arrays), 34

currentLocale, 110

D

data source methods, 152, 174, 335, 364

data storage

(see also archiving, Core Data)

for application data, 246, 247

binary, 255, 261

with **NSData**, 254

`dataSource (UITableView)`, 145, 149-153

debugging

(see also debugging tools, exceptions)

debugging tools

issue navigator, 23

Xcode, 35

- declarations
 - protocol, 76
- default: (switch statement), 39
- delegation
 - design pattern, 174
 - for `UIImagePickerController`, 228
 - for `UITableView`, 145
 - overview, 76-78
 - `UICollectionViewController`, 345
- `deleteRowsAtIndexPaths(_:withRowAnimation:)`, 168
- dependency injection, 151
- dependency inversion principle, 151
- design patterns, 174
- `DetailViewController`, 236, 265-271
- developer documentation, 244
- devices
 - checking for camera, 227-229
 - display resolution, 47
 - Retina display, 24, 93, 94
- dictionaries
 - (see also JSON data)
 - about, 31, 32
 - accessing, 36
 - subscripting, 36
 - using, 233, 234
 - writing to filesystem, 261
- directories
 - application, 246, 247
 - Documents, 246
 - Library/Caches, 246
 - Library/Preferences, 246
 - proj, 113, 127
 - temporary, 246
- display resolution, 47
- document outline (Interface Builder), 7
- documentation
 - developer, 244
 - opening, 129
 - for Swift, 40
- Documents directory, 246
- Double**, 31
- drawing (see views)
- drill-down interface, 203
- Dynamic Type, 181-184
- E**
 - `editButtonItem`, 219
 - editing (`UITableView`, `UITableViewController`), 161, 165
 - editor area (Xcode), 7
 - `encodeInteger(_: forKey:)`, 242
 - `encodeObject(_: forKey:)`, 242
 - `encodeWithCoder(_:)`, 248
 - `encodeWithCoder:`, 242-244
 - `endEditing(_:)`, 214
 - entities
 - creating, 361
 - defined, 354
 - modeling, 354-356
 - relationships between, 370-372
 - saving changes to, 363
 - `enumerate()`, 37
 - enums (enumerations)
 - defined, 39
 - overview of, 320
 - and raw values, 40
 - and switch statements, 39
 - error handling, 257-259, 346
 - errors
 - dealing with, 35
 - in playgrounds, 30
 - traps, 32
 - event handling, 212
 - events
 - control, 104, 289
 - touch, 212, 274
 - (see also touch events)
 - exceptions
 - vs. error handling, 261
 - internal inconsistency, 166
 - Swift vs. other languages, 261
 - expressions, string interpolation and, 37
 - extensions, 347
- F**
 - fallthrough (switch statement), 39
 - fetch requests, 364
 - file inspector, 121
 - file URLs, retrieving, 247
 - filesystem, writing to, 254-256, 261
 - first responder
 - becoming, 72

- and nil-targeted actions, 289
- overview, 211-215
- resigning, 72, 213, 214
- and responder chain, 288
- and **UIMenuController**, 297

Flickr, 308

Float, 31

floating-point types, 31, 33

fonts

- changing preferred size, 183
- customizing size, 54

for-in, 37

forced unwrapping (of optionals), 35

frame (**UIView**), 45-47

frameworks

- Core Data (see Core Data)
- definition of, 46
- linking manually, 84

functions

- (see also *individual function names*)
- callback, 76

G

genstrings, 125

gesture recognizer (see **UIGestureRecognizer**)

gestures

- (see also **UIGestureRecognizer**,
UIScrollView)
- discrete vs. continuous, 299
- long press, 299, 300
- panning, 299-303
- taps, 292-298

GUIDs, 233

H

Hashable, 31

header view (**UITableView**), 161-164

hierarchies, view, 42-51

Homepwner application

- adding an image store, 231
- adding drill-down interface, 204-220
- adding item images, 221-237
- adjusting view properties per size class, 267
- application sandbox, 250
- creating nested stack views, 186-193
- enabling editing, 161-170
- object diagrams, 149, 206

- storing images, 254-259

horizontal ambiguity, 60

HTTP protocol, 331, 332

I

IBAction, 18, 226

@IBInspectable, 286

IBOutlet, 16, 196-198

icons

- (see also images)
- application, 24, 25
- asset catalogs for, 24
- camera, 224

if-let, 36

image picker (see **UIImagePickerController**)

imageName:, 94

**imagePickerController:didFinishPickingM...
...ediaWithInfo:**, 228

imagePickerControllerDidCancel:, 228

images

- (see also camera, icons, **UIImageView**)
- accessing from the cache, 234
- caching, 254-259
- displaying in **UIImageView**, 222, 223
- downloading image data, 327, 345-347
- fetching, 232
- modeling Photo class, 317
- navigating to photos, 350-352
- for Retina display, 93
- saving, 230
- saving to disk, 367
- storing, 231-233
- wrapping in **NSData**, 254

imageWithContentsOfFile(_:), 255

implementation files, navigating, 238

implicit constraints, 188

inactive state, 252

inequality constraints, 118

init(coder:), 245

init(frame:), 45

init?(contentsOfFile:encoding:error:), 261

initial view controller, 82

initializers

- about, 33
- convenience, 147
- custom, 147, 148
- designated, 147

- free, 148
- member-wise, 277
- for standard types, 33
- initWithCoder:**, 242
- inspectors (Xcode)
 - connections, 20
 - file, 121
- instance variables, 248
 - (see also pointers, properties)
- instances, 33
- Int**, 31
- integer types, 31
- Interface Builder
 - (see also Xcode)
 - adding constraints in, 59
 - attributes inspector, 51
 - canvas, 7
 - connecting objects, 16-20
 - connecting with source files, 178
 - document outline, 7
 - modifying view attributes, 286
 - and properties, 178
 - scene, 8
 - setting outlets in, 17, 196, 197
 - setting target-action in, 19
 - size inspector, 52
- interface files
 - bad connections in, 198
 - Base internationalization and, 113
 - connecting with source files, 226
 - making connections in, 226
- internal inconsistency exception, 166
- internationalization, 109-113, 127
 - (see also localization)
- intrinsic content size, 60
- inverse relationships, 371
- iOS simulator
 - running applications on, 12
 - sandbox location, 250
 - saving images to, 229
 - viewing application bundle in, 263
- iPad
 - (see also devices)
 - application icons for, 24
 - running iPhone applications on, 3
- isEmpty (strings), 34
- isSourceTypeAvailable:**, 227
- issue navigator, 23

J

- JSON data, 318, 319

K

- key-value pairs
 - in dictionaries, 31
 - in JSON data, 318
 - in web services, 310
- keyboard
 - attributes, 68-71
 - dismissing, 72, 211-215
 - number pad, 220
- keys
 - creating/using, 233
 - in dictionaries, 31

L

- labels
 - adding, 52
 - adding additional, 132
 - adding constraints to, 116
 - adding to tab bar, 88
 - customizing, 53
 - updating preferred text size, 184
- language settings, 109
 - (see also localization)
- launch images, 26
- layers (of views), 43
- layout attributes, 56, 57
- layout guides, 101, 141
- lazy loading, 82, 91
- let, 29
- libraries
 - (see also frameworks)
 - object, 9
- Library/Caches directory, 246
- Library/Preferences directory, 246
- literal values, 32
- loadView**, 82, 97
- local variables, 37
- localization
 - Base internationalization and, 113
 - internationalization, 109-113, 127
 - lproj directories, 113, 127
 - NSBundle**, 127
 - strings tables, 124-126
 - user settings for, 109

XLIFF data type, 128

locationInView:, 296

loops

 examining in Value History, 38

 for, 37

 for-in, 37

 in Swift, 37

low-memory warnings, 230

lproj directories, 113, 127

M

main bundle, 113, 127

 (see also application bundle)

main interface, 85

main thread, 330

mainBundle, 264

margins, 102, 103

// MARK:, 239, 240

member-wise initializers, 277

memory management

 memory warnings, 230

UITableViewCell, 156

menus (**UIMenuController**), 297, 298, 306

messages

 (see also methods)

 action, 289, 292, 298

 log, 285

UIResponder, 275

methods

 (see also *individual method names*)

 action, 18, 289

 data source, 152

 defined, 27, 34

 protocol, 79

 static, 27

minimumPressDuration, 299

modal view controller, 172, 228

model layer, 5

Model-View-Controller (MVC), 5-7, 145, 174

multi-threading, 330

multipleTouchEnabled (**UIView**), 281

multitouch, enabling, 281

mutableSetValueForKey(_:), 377

MVC (Model-View-Controller), 5-7, 145, 174

N

naming conventions

 cell reuse identifiers, 156

 delegate protocols, 76

 navigation controllers (see

UINavigationController)

 navigationItem (**UIViewController**), 216

 navigators (Xcode)

 defined, 4

 issue, 23

 project, 4

 nearest neighbor, 58

 nextResponder, 288

 nil-targeted actions, 289

NSBundle, 127

NSCoder, 242, 245

NSCoding protocol, 242-245

NSData, 254

NSDate, 261

NSDateFormatter, 110

NSFetchRequest, 364

NSIndexPath, 155, 168

NSJSONSerialization, 319

NSKeyedArchiver, 248-251

NSKeyedUnarchiver, 251

NSLocale, 110

NSLocalizedString(), 124, 126

NSManagedObject, 356-358, 372-382

NSNumber, 261

NSString

 conversion to, 243

 property list serializable, 261

NSTemporaryDirectory, 246

NSURL, 312-315

NSURLRequest, 314, 315, 331, 332

NSURLSession, 314-316

NSURLSessionDataTask, 315, 316, 326, 330

NSURLSessionTask, 331

NSUserDefaults, 246

NSUUID, 233, 234

NSValueTransformer, 356

 number formatters, 75, 110-113

 number pad, 220

O

object graphs, 353

object library, 9

objects

 (see also memory management)

property list serializable, 261

optional, 79

optional binding, 36

optional methods (protocols), 79

optionals

about, 34

and dictionary subscribing, 36

forced unwrapping of, 35

if-let, 36

and optional binding, 36

syntax for, 34

unwrapping, 35

outlets

autogenerating/connecting, 196

connecting constraints to, 136

connecting with source files, 178

defined, 16

setting, 16-18

setting in Interface Builder, 195

P

padding, 158

parallel computing, 330

parent-child contexts, 382-384

Photorama application

adding persistence to, 353-368

adding tags to photos, 369-384

collection view, 333-344

downloading image data, 327-329

web service requests, 308-326

photos (see camera, images)

pixels, 47

playgrounds (Xcode), 28-30

errors in, 30

Value History, 38

viewing console in, 35

pointers

in Interface Builder (see outlets)

points (vs. pixels), 47

predicates, 364

preferences, 246

(see also Dynamic Type, localization)

prepareForSegue:sender:, 200

presentViewController:animated:completion:, 228

preview assistant, 114

project navigator, 4

projects

cleaning and building, 123

creating, 2-4

target settings in, 263

properties

creating in Interface Builder, 178

defined, 34

property list serializable objects, 261

property observer, 74

protocol, 76

protocols

conforming to, 76

declaring, 76

delegate, 76-78

NSCoding, 242-245

optional methods in, 79

required methods in, 79

structure of, 76

UIApplicationDelegate, 254

UICollectionViewDataSource protocol, 335

UICollectionViewDelegate, 345

UIGestureRecognizerDelegate, 301, 302

UIImagePickerControllerDelegate, 228, 230

UINavigationControllerDelegate, 230

UIResponderStandardEditActions, 306

UITableViewDataSource, 145, 152, 153, 155, 168, 169

UITableViewDelegate, 145

UITextFieldDelegate, 76, 211

pseudolanguage, 115

Q

Quartz, 93 (see Core Graphics)

query items, 310

Quick Help, 244

Quick Help (Xcode), 30

Quiz application, 2-26

R

Range, 37

rawValue (enums), 40

reference pages, 244

reference types, 277

region settings, 109

reordering controls, 170

required methods (protocols), 79

requireGestureRecognizerToFail(_:), 304
resignFirstResponder, 72, 213
resources
 asset catalogs for, 24
 defined, 24, 263
responder chain, 288
responders (see first responder, **UIResponder**)
Retina display, 24, 93, 94
reuse identifiers, 335
reuseIdentifier (**UITableViewCell**), 156
reverse(), 34
root view controller (**UINavigationController**), 205-207
rows (**UITableView**)
 adding, 166, 167
 deleting, 168
 moving, 169, 170

S

sandbox, application, 245-247, 263
schemes, 12
sections (**UITableView**), 153, 161
segues, 194
sendAction(_:to:from:forEvent:), 289
sendActionsForControlEvents(_:), 289
setEditing:animated:, 165, 219
sets, 32, 33
settings (see preferences)
Settings application, 246
simulator
 running applications on, 12
 sandbox location, 250
 saving images to, 229
 viewing application bundle in, 263
size classes, 265-271
sort descriptors (**NSFetchRequest**), 364
sourceType (**UIImagePickerController**), 226, 227
stack views, 185-193
states, application, 252-254
static methods, 27
String
 internationalizing, 124
 writing to filesystem, 255-261
string interpolation, 37
strings
 (see also **NSString**)

 initializers for, 33
 interpolation, 37
 isEmpty, 34
 literal, 32
strings tables, 124-126
structs, 277
subscripting
 arrays, 32
 dictionaries, 36
subviews, 42, 92
superview, 48
suspended state, 252, 254
Swift
 about, 27
 documentation for, 40
 enumerations and switch statement, 39
 extensions in, 347
 loops and string interpolation, 37
 optional types in, 34, 257-259
 types in, 27
 using standard types, 28-34
 value types, 277
switch, 39
switch statements, 39

T

tab bar controllers (see **UITabBarController**)
tab bar items, 88-90
table view cells (see **UITableViewCell**)
table view controllers (see **UITableViewController**)
table views (see **UITableView**)
tables (database), 353
tableView, 159
tableView(_:commitEditingStyle:forRowAt...IndexPath:), 168
tableView(_:moveRowAtIndexPath:toIndexPath...ath:), 169
tableView:cellForRowAtIndexPath:, 152, 155-157
tableView:numberOfRowsInSection:, 152, 153
tags
 adding to photos, 377-381
 adding to the interface, 372-377
 creating relationships between, 370-372
target-action pairs
 defined, 18, 19, 174

setting programmatically, 218
 and **UIControl**, 289
 and **UIGestureRecognizer**, 292
 targets, build settings for, 263
 templates (Xcode), xv
 text
 (see also Auto Layout)
 aligning, 67
 compression of, 190
 customizing appearance, 53, 67
 dynamic styling of, 181
 input, 65-73
textFieldShouldReturn:, 211
 threads, 330
 timing functions, 139, 140
 tmp directory, 246
 to-many relationships, 370
 to-one relationships, 370
toggleEditingMode:, 165
 toolbars
 adding, 224
 adding buttons to, 225
 adding constraints to, 224
 anchoring, 372
topViewController (UINavigationController),
 205
 touch events
 basics of, 274, 275
 defined, 212
 enabling multitouch, 281-285
 and responder chain, 288
 and target-action pairs, 289
 and **UIControl**, 289
touchesBegan(_:withEvent:), 274
touchesCancelled(_:withEvent:), 274
touchesEnded(_:withEvent:), 274
touchesMoved(_:withEvent:), 274
 TouchTracker application
 creating, 275
 drawing lines, 276-285
 recognizing gestures, 291-306
 transformable attributes (Core Data), 356
translationInView(_:), 302
 traps, 32
 tuples, 37
 type inference, 30
 types
 boolean, 31

floating-point, 31, 33
 hashable, 31
 inference of, 30
 instances of, 33
 integer, 31
 sets, 32, 33
 specifying, 30
 tuples, 37

U

UI thread, 330
UIAlertController, 170-173
UIApplication
 and events, 274
 and responder chain, 288, 289
UIApplicationDelegate, 254
UIBarButtonItem, 217-219, 224-226
UICollectionViewCell, 341-344
UICollectionViewDataSource protocol, 335
UICollectionViewDelegate protocol, 345
UICollectionViewFlowLayout, 333-340
UIColor, 46
UIControl, 289
UIControlEvent.TouchUpInside, 289
UIControlEvents, 104
UIGestureRecognizer
 action messages of, 292, 299
 cancelsTouchesInView, 300
 chaining recognizers, 304
 delaying touches, 304
 described, 291
 detecting taps, 292-298
 enabling simultaneous recognizers, 301
 implementing multiple, 294-296, 301-303
 intercepting touches from view, 292, 300, 301
 locationInView:, 296
 long press, 299, 300
 panning, 299-303
 state (property), 299, 302, 304
 subclasses, 72, 292, 304
 subclassing, 305
 translationInView(_:), 302
 and **UIResponder** methods, 300
UIGestureRecognizerDelegate, 301, 302
UIImage, 255
 (see also images, **UIImageView**)
UIImageJPEGRepresentation, 255

- UIImagePickerController**
 - instantiating, 226-228
 - presenting, 228-230
- UIImagePickerControllerDelegate**, 228, 230
- UIImageView**, 222, 223
- UIInterpolatingMotionEffect**, 130
- UIKit, 46
- UILongPressGestureRecognizer**, 299, 300
- UIMenuController**, 297, 298, 306
- UINavigationController**, 206-220
- UINavigationController**
 - (see also **view controllers**)
 - adding view controllers to, 210
 - described, 205-208
 - instantiating, 207
 - managing view controller stack, 205
 - root view controller, 205, 206
 - in storyboards, 194
 - topViewController, 205, 206
 - and **UINavigationController**, 216-219
 - view, 206
 - viewControllers, 205
 - viewWillAppear:**, 210
 - viewWillDisappear:**, 210
- UINavigationControllerDelegate**, 230
- UINavigationControllerItem**, 216-219
- UIPanGestureRecognizer**, 299-303
- UIResponder**
 - menu actions, 306
 - and responder chain, 288
 - and touch events, 274
- UIResponderStandardEditActions** (protocol), 306
- UIScrollView**, 158
- UIStackView**, 187-193
- UINavigationControllerSegue**, 194-200
- UITabBarController**
 - implementing, 85-90
 - vs. **UINavigationController**, 203
 - view, 87
- UITabBarItem**, 88-90
- UITableView**, 143-145
 - (see also **UITableViewCell**, **UITableViewController**)
 - adding rows to, 166, 167
 - deleting rows from, 168
 - editing mode of, 161, 165, 176, 219
 - editing property, 161, 165
 - footer view, 161
 - header view, 161-164
 - moving rows in, 169, 170
 - populating, 149-156
 - sections, 153, 161
 - view, 146
- UITableViewCell**
 - cell styles, 154
 - contentView, 154
 - editing styles, 168
 - retrieving instances of, 155, 156
 - reusing instances of, 156-158
 - subclassing, 175-184
 - UITableViewCellStyle**, 154
- UITableViewCellEditingStyleDelete**, 168
- UITableViewController**
 - (see also **UITableView**)
 - adding rows, 166, 167
 - data source methods, 152
 - dataSource, 149-153
 - deleting rows, 168
 - described, 145
 - editing property, 165
 - moving rows, 169, 170
 - returning cells, 155, 156
 - subclassing, 146
 - tableView**, 159
- UITableViewDataSource** (protocol), 145, 152, 153, 155, 168, 169
- UITableViewDelegate**, 145
- UITapGestureRecognizer**, 72, 292-298
- UITextField**
 - as first responder, 211, 289
 - and keyboard, 211
 - setting attributes of, 220
 - text editing, 65
- UITextFieldDelegate**, 76, 211
- UIToolbar**, 217, 225
- UITouch**, 274, 275, 280-285
- UIView**
 - (see also **UIViewController**, **views**)
 - animation documentation, 129
 - defined, 41
 - frame, 45-47
 - instantiating, 45
 - superview, 48
- UIViewController**
 - (see also **UIView**, **view controllers**)

loadView, 82, 97
 navigationItem, 216
 tabBarItem, 88
 view, 82, 288
viewDidLoad, 92
viewWillAppear:, 92, 230
UIWindow
 purpose of, 42
 and responder chain, 288
unarchiveObjectWithFile(_:), 251
URLForResource(_:withExtension:), 127
 URLs, 309
 (see also **NSURL**)
 user alerts, displaying, 170-173
 user interface
 (see also Auto Layout, views)
 drill-down, 203
 keyboard, 211
 user settings (see preferences)
 UUIDs, 233, 356
V
 value types, 277
 var, 29
 variables, 29
 (see also instance variables, local variables,
 pointers, properties)
 view (**UIViewController**), 82
 view controllers
 (see also **UIViewController**, views)
 allowing access to image store, 232
 interacting with, 92
 lazy loading of views, 82, 91
 modal, 172, 228
 navigating between, 194
 presenting, 85
 reloading subviews, 230
 root, 205
 setting initial, 82
 and view hierarchy, 82, 97
 view hierarchy, 42-51, 97
viewControllers (**UINavigationController**),
 205
viewDidLoad, 46, 92
 views
 (see also Auto Layout, touch events, **UIView**,
 view controllers)

 adding to window, 42, 97
 animating, 129-139
 appearing/disappearing, 210
 content compression resistance priorities, 190
 content hugging priorities, 189
 creating programmatically, 97
 defined, 41
 drawing to screen, 43
 in hierarchy, 42, 43
 layers and, 43
 lazy loading of, 82, 91
 misplaced, 62
 modal presentation of, 228
 in Model-View-Controller, 5
 removing from storyboard, 95
 rendering, 43
 resizing, 222, 223
 scroll, 334
 size and position of, 45-47
 stack view, 185-193, 267
 and subviews, 42-51
viewWillAppear:, 92, 210, 230
viewWillDisappear:, 210

W

web services
 and HTTP protocol, 331, 332
 with JSON data, 318, 319
 and **NSURLSession**, 314-316
 overview, 307, 308
 requesting data from, 309-316
 wildcard Any Width/Height layout, 266
 workspaces (Xcode), 4
 WorldTrotter application
 adding tab bar controller, 85-90
 configuring, 44
 implementing temperature conversion, 73
 interface layout, 51
 localizing, 110-127
 multiple view controllers for, 81
 programmatic views in, 95
 text input, 65-73
writeToURL(_:atomically:), 255
writeToURL(_:atomically:encoding:error:),
 261

X

- `.xcassets` (asset catalog), 24
- `.xcdatamodeld` (data model file), 354

Xcode

- (see also debugging tools, Interface Builder, projects, iOS simulator)
 - API Reference, 244
 - asset catalogs, 24
 - assistant editor, 226
 - creating projects in, 2-4
 - documentation, 129
 - editor area, 7
 - file inspector, 121
 - issue navigator, 23
 - navigator area, 4
 - navigators, 4
 - object library, 9
 - organizing methods with `// MARK:`, 239
 - playgrounds, 28-30
 - Quick Help, 30, 244
 - schemes, 12
 - source editor jump bar, 238
 - versions, 2
 - workspaces, 4
- XLIFF data type, 128
- XML property lists, 261