

John Ray



In **Full Color**

Sams **Teach Yourself**

# iOS 8

Figures and  
code appear as they  
do in Xcode 6.x

Covers **iOS 8, Swift,  
Universal development  
and much more!**

Additional files and  
updates available  
online

# Application Development

in **24**  
**Hours**

**SAMS**

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

John Ray

Sams **Teach Yourself**

**iOS 8**

# **Application Development**

in **24**  
**Hours**

**SAMS**

800 East 96th Street, Indianapolis, Indiana, 46240 USA

## **Sams Teach Yourself iOS 8 Application Development in 24 Hours**

Copyright © 2015 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 9780672337239

ISBN-10: 0672337231

Library of Congress Cataloging-in-Publication Data: 2015900442

Printed in the United States of America

First Printing March 2015

### **Trademarks**

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

### **Warning and Disclaimer**

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

### **Special Sales**

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [international@pearsoned.com](mailto:international@pearsoned.com).

U.S. Corporate and Government Sales

1-800-382-3419

[corpsales@pearsontechgroup.com](mailto:corpsales@pearsontechgroup.com)

For sales outside of the U.S., please contact

International Sales

[international@pearsoned.com](mailto:international@pearsoned.com)

### **Editor-in-Chief**

Greg Wiegand

### **Senior Acquisitions Editor**

Laura Norman

### **Development Editor**

Mark Renfrow

### **Managing Editor**

Sandra Schroeder

### **Project Editor**

Seth Kerney

### **Indexer**

Lisa Stumpf

### **Proofreader**

Sarah Kearns

### **Technical Editor**

Anne Groves

### **Publishing Coordinator**

Cindy Teeters

### **Book Designer**

Mark Shirar

### **Compositor**

Bronkella

Publishing, LLC

# Contents at a Glance

Introduction .....	1
<b>HOUR 1</b> Preparing Your System and iDevice for Development .....	5
<b>2</b> Introduction to Xcode and the iOS Simulator .....	29
<b>3</b> Discovering Swift and the iOS Playground .....	77
<b>4</b> Inside Cocoa Touch .....	117
<b>5</b> Exploring Interface Builder .....	147
<b>6</b> Model-View-Controller Application Design .....	185
<b>7</b> Working with Text, Keyboards, and Buttons .....	215
<b>8</b> Handling Images, Animation, Sliders, and Steppers .....	251
<b>9</b> Using Advanced Interface Objects and Views .....	281
<b>10</b> Getting the User's Attention .....	317
<b>11</b> Implementing Multiple Scenes and Popovers .....	349
<b>12</b> Making Choices with Toolbars and Pickers .....	401
<b>13</b> Advanced Storyboards Using Navigation and Tab Bar Controllers .....	445
<b>14</b> Navigating Information Using Table Views and Split View Controllers .....	485
<b>15</b> Reading and Writing Application Data .....	527
<b>16</b> Building Responsive User Interfaces .....	573
<b>17</b> Using Advanced Touches and Gestures .....	611
<b>18</b> Sensing Orientation and Motion .....	639
<b>19</b> Working with Rich Media .....	669
<b>20</b> Interacting with Other iOS Services .....	713
<b>21</b> Implementing Location Services .....	751
<b>22</b> Building Background-Ready Applications .....	783
<b>23</b> Building Universal Applications .....	815
<b>24</b> Application Tracing, Monitoring, and Debugging .....	837
Index .....	863
Online Appendix: Introducing Xcode Source Control	

# Table of Contents

Introduction	1
<b>HOURL 1: Preparing Your System and iDevice for Development</b>	<b>5</b>
Welcome to the iOS Platform	5
Becoming an iOS Developer	9
Running an iOS App	16
Developer Technology Overview	23
Further Exploration	24
Summary	24
Q&A	25
Workshop	25
Activities	27
<b>HOURL 2: Introduction to Xcode and the iOS Simulator</b>	<b>29</b>
Using Xcode	29
Using the iOS Simulator	63
Further Exploration	71
Summary	71
Q&A	72
Workshop	72
Activities	75
<b>HOURL 3: Discovering Swift and the iOS Playground</b>	<b>77</b>
Object-Oriented Programming and Swift	77
The Terminology of Object-Oriented Development	79
Exploring the Swift File Structure	82
Swift Programming Basics	88
Memory Management and Automatic Reference Counting	107
Introducing the iOS Playground	108
Further Exploration	112
Summary	112
Q&A	113

Workshop .....	113
Activities .....	116
<b>HOURL 4: Inside Cocoa Touch</b>	<b>117</b>
What Is Cocoa Touch? .....	117
Exploring the iOS Technology Layers .....	119
Tracing the iOS Application Life Cycle .....	126
Cocoa Fundamentals .....	127
Exploring the iOS Frameworks with Xcode .....	135
Further Exploration .....	143
Summary .....	143
Q&A .....	143
Workshop .....	144
Activities .....	146
<b>HOURL 5: Exploring Interface Builder</b>	<b>147</b>
Understanding Interface Builder .....	147
Creating User Interfaces .....	154
Customizing the Interface Appearance .....	164
Connecting to Code .....	170
Further Exploration .....	179
Summary .....	180
Q&A .....	181
Workshop .....	181
Activities .....	184
<b>HOURL 6: Model-View-Controller Application Design</b>	<b>185</b>
Understanding the MVC Design Pattern .....	185
How Xcode Implements MVC .....	187
Using the Single View Application Template .....	191
Further Exploration .....	209
Summary .....	210
Q&A .....	210
Workshop .....	210
Activities .....	213

<b>HOURL 7: Working with Text, Keyboards, and Buttons</b>	<b>215</b>
Basic User Input and Output .....	215
Using Text Fields, Text Views, and Buttons .....	217
Further Exploration .....	246
Summary .....	246
Q&A .....	247
Workshop .....	247
Activities .....	249
<b>HOURL 8: Handling Images, Animation, Sliders, and Steppers</b>	<b>251</b>
User Input and Output .....	251
Creating and Managing Image Animations, Sliders, and Steppers .....	253
Further Exploration .....	275
Summary .....	276
Q&A .....	276
Workshop .....	277
Activities .....	279
<b>HOURL 9: Using Advanced Interface Objects and Views</b>	<b>281</b>
User Input and Output (Continued) .....	281
Using Switches, Segmented Controls, and Web Views .....	287
Using Scrolling Views .....	303
Further Exploration .....	312
Summary .....	313
Q&A .....	313
Workshop .....	313
Activities .....	316
<b>HOURL 10: Getting the User's Attention</b>	<b>317</b>
Alerting the User .....	317
Exploring User Alert Methods .....	328
Further Exploration .....	343
Summary .....	344
Q&A .....	344
Workshop .....	344
Activities .....	347

<b>HOURL 11: Implementing Multiple Scenes and Popovers</b>	<b>349</b>
Introducing Multiscene Storyboards .....	350
Using Segues .....	377
Popovers, Universal Applications, and iPhones .....	390
Further Exploration .....	395
Summary .....	396
Q&A .....	396
Workshop .....	396
Activities .....	399
 <b>HOURL 12: Making Choices with Toolbars and Pickers</b>	 <b>401</b>
Understanding the Role of Toolbars .....	401
Exploring Pickers .....	404
Using the Date Picker .....	412
Using a Custom Picker .....	425
Further Exploration .....	440
Summary .....	441
Q&A .....	441
Workshop .....	442
Activities .....	444
 <b>HOURL 13: Advanced Storyboards Using Navigation and Tab Bar Controllers</b>	 <b>445</b>
Advanced View Controllers .....	445
Exploring Navigation Controllers .....	447
Understanding Tab Bar Controllers .....	452
Using a Navigation Controller .....	458
Using a Tab Bar Controller .....	469
Further Exploration .....	479
Summary .....	480
Q&A .....	480
Workshop .....	481
Activities .....	483
 <b>HOURL 14: Navigating Information Using Table Views and Split View Controllers</b>	 <b>485</b>
Understanding Tables .....	486
Exploring the Split View Controller .....	495
A Simple Table View Application .....	498

Creating a Master-Detail Application .....	507
Further Exploration .....	521
Summary .....	522
Q&A .....	523
Workshop .....	523
Activities .....	525
<b>HOURL 15: Reading and Writing Application Data</b> .....	<b>527</b>
iOS Applications and Data Storage .....	527
Data Storage Approaches .....	530
Creating Implicit Preferences .....	539
Implementing System Settings .....	546
Implementing File System Storage .....	559
Further Exploration .....	567
Summary .....	568
Q&A .....	568
Workshop .....	569
Activities .....	571
<b>HOURL 16: Building Responsive User Interfaces</b> .....	<b>573</b>
Responsive Interfaces .....	573
Using Auto Layout .....	578
Programmatically Defined Interfaces .....	600
Further Exploration .....	607
Summary .....	607
Q&A .....	608
Workshop .....	608
Activities .....	610
<b>HOURL 17: Using Advanced Touches and Gestures</b> .....	<b>611</b>
Multitouch Gesture Recognition .....	611
Adding Gesture Recognizers .....	612
Using Gesture Recognizers .....	614
Further Exploration .....	635
Summary .....	635
Q&A .....	636
Workshop .....	636
Activities .....	638

<b>HOOR 18: Sensing Orientation and Motion</b>	<b>639</b>
Understanding Motion Hardware .....	639
Accessing Orientation and Motion Data .....	643
Sensing Orientation .....	647
Detecting Acceleration, Tilt, and Rotation .....	652
Further Exploration .....	663
Summary .....	664
Q&A .....	664
Workshop .....	664
Activities .....	666
 <b>HOOR 19: Working with Rich Media</b>	 <b>669</b>
Exploring Rich Media .....	669
The Media Playground Application .....	683
Further Exploration .....	708
Summary .....	709
Q&A .....	709
Workshop .....	710
Activities .....	712
 <b>HOOR 20: Interacting with Other iOS Services</b>	 <b>713</b>
Extending iOS Service Integration .....	713
Using the Address Book, Email, Social Networking, and Maps .....	730
Further Exploration .....	746
Summary .....	746
Q&A .....	746
Workshop .....	747
Activities .....	749
 <b>HOOR 21: Implementing Location Services</b>	 <b>751</b>
Understanding Core Location .....	751
Creating a Location-Aware Application .....	758
Further Exploration .....	777
Summary .....	778
Q&A .....	778
Workshop .....	778
Activities .....	781

<b>HOURL 22: Building Background-Ready Applications</b>	<b>783</b>
Understanding iOS Backgrounding .....	783
Disabling Backgrounding .....	789
Handling Background Suspension .....	790
Implementing Local Notifications .....	792
Using Task-Specific Background Processing .....	795
Completing a Long-Running Background Task .....	800
Performing a Background Fetch .....	806
Further Exploration .....	810
Summary .....	811
Q&A .....	811
Workshop .....	812
Activities .....	814
<b>HOURL 23: Building Universal Applications</b>	<b>815</b>
Universal Application Development .....	815
Size Classes .....	819
Further Exploration .....	833
Summary .....	834
Q&A .....	834
Workshop .....	834
Activities .....	836
<b>HOURL 24: Application Tracing, Monitoring, and Debugging</b>	<b>837</b>
Instant Feedback with NSLog .....	838
Using the Xcode Debugger .....	841
Further Exploration .....	858
Summary .....	859
Q&A .....	859
Workshop .....	859
Activities .....	862
<b>Index</b>	<b>863</b>
<b>ONLINE APPENDIX A: Introducing Xcode Source Control</b>	

# About the Author

**John Ray** currently serves as the Director of the Office of Research Information Systems at The Ohio State University. He has written numerous books for Macmillan/Sams/Que, including *Using TCP/IP: Special Edition*, *Teach Yourself Dreamweaver MX in 21 Days*, *Mac OS X Unleashed*, *My Yosemite MacBook*, and *Teach Yourself iOS 7 Development in 24 Hours*. As a Macintosh user since 1984, he strives to ensure that each project presents the Macintosh with the equality and depth it deserves. Even technical titles such as *Using TCP/IP* contain extensive information about the Macintosh and its applications and have garnered numerous positive reviews for their straightforward approach and accessibility to beginner and intermediate users.

You can visit his website at <http://teachyourselfios.com> or follow him on Twitter at @johnemeryray or #iOSIn24.

# Dedication

*This book is dedicated to taking a long nap. Shhhhhh...*

# Acknowledgments

Thank you to the group at Sams Publishing—Laura Norman, Keith Cline, Mark Renfrow—and my Tech Editor, Anne Groves, for helping me survive this tumultuous year of updates. From Yosemite, to Xcode 6.x, to Swift—Apple can't seem to sit still for more than a few minutes. Getting these changes into a book, and getting them right, has been quite the challenge for the entire team. Thank you all!

# We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

*Please note that we cannot help you with technical problems related to the topic of this book.*

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: consumer@sampublishing.com

Mail: Sams Publishing  
ATTN: Reader Feedback  
800 East 96th Street  
Indianapolis, IN 46240 USA

## Reader Services

Visit our website and register this book at [informit.com/register](http://informit.com/register) for convenient access to any updates, downloads, or errata that might be available for this book.

*This page intentionally left blank*

# Introduction

When you pick up an iOS device and use it, you feel connected. Whether it be an iPad, an iPhone, or an iPod, the interface acts as an extension to your fingers; it is smooth, comfortable, and invites exploration. Other competing devices offer similar features, and even sport gadgets such as styluses and trackpads, but they cannot match the user experience that is iOS.

iOS and its associated development tools have changed rapidly over the past few years. iOS 7 brought us a new user interface that used depth and translucency to keep users connected to their content and aware of the context in which they are accessing it. iOS 8 includes even more refinement, but, perhaps more importantly, supports a brand new language for developing apps – Swift.

Swift marks a dramatic change in the history of iOS and OS X development. With Swift, Apple has effectively retired the Objective-C language - used on Apple and NeXT platforms for over 20 years. Swift promises to be a friendlier development platform with more modern language features and tools. While in development for over four years at Apple, by the time this book reaches you, Swift will have existed as a public programming language for only a few months.

In writing the revision to this book, we had to make some tough choices – we could remain focused on Objective-C, or immediately shift to Swift. Swift is rapidly evolving, and changes with each release of Xcode. Code that was written in one version of Xcode, sometimes breaks in the next. Needless to say, Swift presented challenges - but that's the direction we took. Swift is the future of Apple development, and learning it now will give you a leg up on your Objective-C compatriots. Will there be things that make you scratch your head? Yup, but I also think you'll find Swift much more *fun* (yes, really) to use than Objective-C.

When creating Swift and the iOS development platform, Apple considered the entire application life-cycle – from the interface design tools, to the code that makes it function, to the presentation to the user – everything is integrated and works together seamlessly. As a developer, does this mean that there are rules to follow? Absolutely. But, by following these rules, you can create applications that are interactive works of art for your users to love—not software they will load and forget.

Through the App Store, Apple has created the ultimate digital distribution system for iOS applications. Programmers of any age or affiliation can submit their applications to the App Store for just the cost of a modest yearly Developer Membership fee. Games, utilities, and full-feature

applications have been built for everything from pre-K education to retirement living. No matter what the content, with a user base as large as the iPhone, iPod Touch, and iPad, an audience exists.

My hope is that this book brings iOS development to a new generation of developers. *Teach Yourself iOS 8 Development in 24 Hours* provides a clear and natural progression of skills development, from installing developer tools and registering your device with Apple, to debugging an application before submitting it to the App Store. It's everything you need to get started - in 24 one-hour lessons.

## Who Can Become an iOS Developer?

If you have an interest in learning, time to invest in exploring and practicing with Apple's developer tools, and an Intel Macintosh computer running Mavericks, Yosemite, or later, you have everything you need to begin creating software for iOS.

Developing an app won't happen overnight, but with dedication and practice, you can be writing your first applications in a matter of days. The more time you spend working with the Apple developer tools, the more opportunities you'll discover for creating new and exciting projects.

You should approach iOS application development as creating software that *you* want to use, not what you think others want. If you're solely interested in getting rich quick, you're likely to be disappointed. (The App Store is a crowded marketplace—albeit one with a lot of room—and competition for top sales is fierce.) However, if you focus on building useful and unique apps, you're much more likely to find an appreciative audience.

## Who Should Use This Book?

This book targets individuals who are new to development for iOS and have experience using the Macintosh platform. No previous experience with Swift, Cocoa, or the Apple developer tools is required. Of course, if you do have development experience, some of the tools and techniques may be easier to master, but the author does not assume that you've coded before.

That said, some things are expected of you, the reader. Specifically, you must be willing to invest in the learning process. If you just read each hour's lesson without working through the tutorials, you will likely miss some fundamental concepts. In addition, you need to spend time reading the Apple developer documentation and researching the topics presented in this book. A vast amount of information on iOS development is available, but only limited space in this book. Therefore, this book covers what you need to forge your own path forward.

## What Is (and Isn't) in This Book?

The material in this book specifically targets iOS release 8.1 and later on Xcode 6.1 and later. Much of what you'll learn is common to all the iOS releases, but this book also covers several important areas that have only come about in recent iOS releases, such as gesture recognizers, embedded video playback with AirPlay, Core Image, social networking, multitasking, universal (iPhone/iPad) applications, Auto Layout, Size Classes, and more!

Unfortunately, this is not a complete reference for the iOS application programming interfaces (APIs); some topics just require much more space than this book allows. Thankfully, the Apple developer documentation is available directly within the free tools you install in Hour 1, "Preparing Your System and iDevice for Development." In many hours, you'll find a section titled "Further Exploration." This identifies additional related topics of interest. Again, a willingness to explore is an important quality in becoming a successful developer.

Each coding lesson is accompanied by project files that include everything you need to compile and test an example or, preferably, follow along and build the application yourself. Be sure to download the project files from this book's website at <http://teachyourselfios.com>. If you have issues with any projects, view the posts on this site to see whether a solution has been identified.

In addition to the support website, you can follow along on Twitter! Search for #iOSIn24 on Twitter to receive official updates and tweets from other readers. Use the hashtag #iOSIn24 in your tweets to join the conversation. To send me messages via Twitter, begin each tweet with @johnemeryray.

*This page intentionally left blank*

# HOUR 3

## Discovering Swift and the iOS Playground

---

### What You'll Learn in This Hour:

- ▶ How Swift will be used in your projects
- ▶ The basics of object-oriented programming
- ▶ Simple Swift syntax
- ▶ Common Swift data types
- ▶ How to test your code in the iOS Playground

This hour's lesson marks the midpoint in our exploration of the Apple iOS development platform. It will give us a chance to sit back, catch our breath, and get a better idea of what it means to “code” for iOS. Both OS X and iOS share a common development environment and a common development language: Swift.

Swift provides the syntax and structure for creating applications on Apple platforms. Swift is a brand-new language developed internally in Apple that still has that “new language smell,” but also a few disconcerting knocks under the hood. It can seem a bit daunting at first, but after a few hours of practice, you'll begin to feel right at home. This hour takes you through the steps you need to know to be comfortable with Swift and starts you down the path to mastering this unique and powerful language.

## Object-Oriented Programming and Swift

To better understand the scope of this hour, take a few minutes to search for Swift or object-oriented programming in your favorite online bookstore. You will find quite a few books—lengthy books—on these topics. In this book, roughly 30 pages cover what other books teach in hundreds of pages. Although it's not possible to fully cover Swift and object-oriented development in this single hour, we can make sure that you understand enough to develop fairly complex apps.

To provide you with the information you need to be successful in iOS development, this hour concentrates on fundamentals—the core concepts that are used repeatedly throughout the examples and tutorials in this book. The approach in this hour is to introduce you to a

programming topic in general terms, and then look at how it will be performed when you sit down to write your application. Before we begin, let's look a bit closer at Swift and object-oriented programming.

## What Is Object-Oriented Programming?

Most people have an idea of what programming is and have even written a simple program. Everything from setting your DVR to record a show to configuring a cooking cycle for your microwave is a type of programming. You use data (such as times) and instructions (like “record”) to tell your devices to complete a specific task. This certainly is a long way from developing for iOS, but in a way the biggest difference is in the amount of data you can provide and manipulate and the number of different instructions available to you.

### Imperative Development

There are two primary development paradigms: imperative programming and object-oriented programming. First, imperative programming (a subset of which is called *procedural programming*) implements a sequence of commands that should be performed. The application follows the sequence and carries out activities as directed. Although there might be branches in the sequence or movement back and forth between some of the steps, the flow is from a starting condition to an ending condition, with all the logic to make things *work* sitting in the middle.

The problem with imperative programming is that it lends itself to growing, without structure, into an amorphous blob. Applications gain features when developers tack on bits of code here and there. Often, instructions that implement a piece of functionality are repeated over and over wherever something needs to take place. Procedural programming refers to an imperative programming structure that attempts to avoid repetition by creating functions (or procedures) that can be reused. This works to some extent, but long-term still often results in code bloat. The benefit of this approach, however, is that it is quite easy to pick up and learn: You create a series of instructions, and the computer follows them.

### The Object-Oriented Approach

The other development approach, and what we use in this book, is object-oriented programming (OOP). OOP uses the same types of instructions as imperative development but structures them in a way that makes your applications easy to maintain and promotes code reuse whenever possible. In OOP, you create objects that hold the data that describes something along with the instructions to manipulate that data. Perhaps an example is in order.

Consider a program that enables you to track reminders. With each reminder, you want to store information about the event that will be taking place—a name, a time to sound an alarm, a location, and any additional miscellaneous notes that you may want to store. In addition, you need to be able to reschedule a reminder's alarm time or completely cancel an alarm.

In the imperative approach, you have to write the steps necessary to track all the reminders, all the data in the reminders, check every reminder to see whether an alarm should sound, and so on. It's certainly possible, but just trying to wrap your mind around everything that the application needs to do could cause some serious headaches. An object-oriented approach brings some sanity to the situation.

In an object-oriented model, you could implement a reminder as a single object. The reminder object would know how to store the properties such as the name, location, and so on. It would implement just enough functionality to sound its own alarm and reschedule or cancel its alarm. Writing the code, in fact, would be very similar to writing an imperative program that only has to manage a single reminder. By encapsulating this functionality into an object, however, we can then create multiple copies of the object within an application and have them each fully capable of handling separate reminders. No fuss and no messy code!

Most of the tutorials in this book make use of one or two objects, so don't worry about being overwhelmed with OOP. You'll see enough to get accustomed to the idea, but we're not going to go overboard.

Another important facet of OOP is inheritance. Suppose that you want to create a special type of reminder for birthdays that includes a list of birthday presents that a person has requested. Instead of tacking this onto the reminder object, you could create an entirely new "birthday reminder" that inherits all the features and properties of a reminder and then adds in the list of presents and anything else specific to birthdays.

## The Terminology of Object-Oriented Development

OOP brings with it a whole range of terminology that you need to get accustomed to seeing in this book (and in Apple's documentation). The more familiar you are with these terms, the easier it will be to look for solutions to problems and interact with other developers. Let's establish some basic vocabulary now:

- ▶ **Class:** The code, usually consisting of a single Swift file, which defines an object and what it can do.
- ▶ **Subclass:** A class that builds upon another class, adding additional features. Almost everything you use in iOS development will be a subclass of something else, inheriting all the properties and capabilities of its parent class.
- ▶ **Superclass/parent class:** The class that another class inherits from.

- ▶ **Singleton:** A class that is instantiated only once during the lifetime of a program. For example, a class to read your device's orientation is implemented as a singleton because there is only one sensor that returns this information.
- ▶ **Object/instance:** A class that has been invoked and is active in your code. Classes are the code that makes an object work, whereas an object is the actual class "in action." This is also known as an *instance* of a class.
- ▶ **Instantiation:** The process of creating an active object from a class.
- ▶ **Instance method:** A basic piece of functionality, implemented in a class. For the reminder class, this might be something like `setAlarm` to set the alarm for a given reminder. Methods are, by default, available within the class they are defined and within other classes defined in the same project.
- ▶ **Extensions:** Provide a means of extending a class without modifying the class code itself.
- ▶ **Type method:** Similar to an instance method, but applicable to all the objects created from a class. The reminder class, for example, might implement a type method called `countReminders` that provides a count of all the reminder objects that have been created. If you're familiar with other OO languages, you may recognize this as a *static method* or a *class method*.
- ▶ **Variable property:** A storage place for a piece of information specific to a class. The name of a reminder, for example, might be stored in a variable property. All variables have a specific "type" that describes the contents of what they will be holding. Variable properties only differ from normal variables in where they are defined and where they can be accessed.
- ▶ **Variable:** A storage location for a piece of information. Unlike variable properties, a "normal" variable is accessible only in the method where it is defined.
- ▶ **Constant:** A Swift constant is another type of variable, but one that cannot be modified after it has been declared.
- ▶ **Parameter:** A piece of information that is provided to a method when it is use. If you were to use a `setAlarm` method, you would presumably need to include the time to set. The time, in this case, would be a parameter.
- ▶ **Protocol:** Protocols declare methods that can be implemented by a class—usually to provide functionality needed for an object. A class that implements a protocol is said to conform to that protocol. This is similar to a Java interface.
- ▶ **Self:** A way to refer to an object within its own methods. When an instance method or variable property is used in an application, it should be used with a specific object. If

you're writing code within a class and you want it to access one of its own methods or variable properties, you *can* `self` to refer to the object. In Swift, `self` is usually implied and only needs to be used explicitly in very specific circumstances.

It's important to know that when you develop for iOS you're going to be taking advantage of hundreds of classes that Apple has already written for you. Everything from creating onscreen buttons to manipulating dates and writing files is covered by prebuilt classes. You'll occasionally want to customize some of the functionality in those classes, but you'll be starting out with a toolbar already overflowing with functionality.

Confused? Don't worry! This book introduces these concepts slowly, and you'll quickly get a feel for how they apply to your projects as you work through several tutorials in the upcoming hours.

## What Is Swift?

For years, Apple development has centered on a decades-old language called Objective-C. Objective-C, while appealing to some, was about as far from a “modern” language as you could get. Languages like Python and Ruby have sprung up and attracted legions of followers with their simple syntax and focus on results, rather than esoteric concepts like memory management. Swift is Apple's answer to the call for a modern iOS and OS X development language.

Released in 2014, Swift carries with it many of the niceties of Objective-C, but loses much of the baggage. The biggest issue with Swift is that it is still evolving, and developers (including yours truly) are still trying to figure out the best way to use it. It will be several years before this churn settles down—but, in the meantime, the core of Swift is fast, flexible, and easy to learn.

Swift statements are easier to read than other programming languages and can often be deciphered just by looking at them. For example, code that checks to see if two dates are equal might be written like this:

```
if myBirthday.isEqualToDate(yourBirthday) {  
    // We're the same age!  
}
```

It doesn't take a very large mental leap to see what is going on in the code snippet. Throughout the book, I will try to explain what each line of code is doing—but chances are you can pick up on the intention just by reading the lines.

### CAUTION

---

#### Case Counts

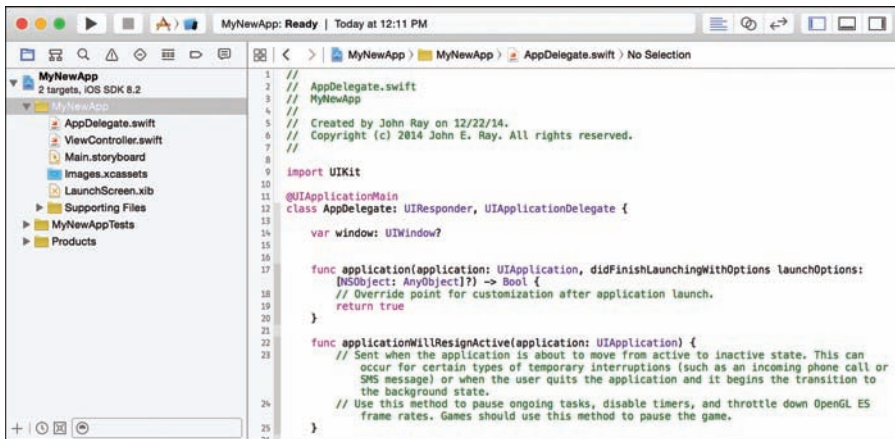
Swift is case sensitive. If a program is failing, make sure that you aren't mixing case somewhere in the code.

---

Now that you have an idea of what OOP and Swift are, let's take a look at how you'll be using them over the course of this book.

## Exploring the Swift File Structure

In the preceding hour, you learned how to use Xcode to create projects and navigate their files. As mentioned then, the vast majority of your time will be spent in the project group of Xcode, which is shown for the MyNewApp project in Figure 3.1. You'll be adding methods to class files that Xcode creates for you when you start a project or, occasionally, creating your own class files to implement entirely new functionality in your application.



**FIGURE 3.1**

Most of your coding will occur within the files in your project group.

Okay, sounds simple enough, but where will the coding take place? If you create a new project look, you'll see quite a few different files staring back at you.

## Class Files

In Swift, a class is implemented within a single file with the .swift extension. This file contains all of the variable/constant definitions, and all of the methods containing the application logic. Other classes in your project will automatically be able to access the methods in this file, if needed.

Let's review the structure of an entirely made-up class file in Listing 3.1.

---

**LISTING 3.1 A Sample Interface File**

---

```
1: import UIKit
2:
3: class myClass: myParent, myProtocol {
4:
5:     var myString: String = ""
6:     var myOtherString: String?
7:     var yetAnotherVariable: Float!
8:     let myAge: Int = 29
9:
10:    @IBOutlet weak var userOutput: UILabel!
11:    @IBOutlet var anotherUserOutput: UILabel!
12:
13:    class func myTypeMethod(aString: String) -> String {
14:        // Implement a Type method here
15:    }
16:
17:    func myInstanceMethod(myString: String, myURL: NSURL) -> NSDate? {
18:        // Implement the Instance method here
19:    }
20:
21:    override func myOverriddenInstanceMethod() {
22:        // Override the parent's method here
23:    }
24:
25:    @IBAction func myActionMethod(sender: AnyObject) {
26:        // React to an action in the user interface
27:    }
28:
29: }
```

---

---

**CAUTION**

---

**Line Numbers Are for Reference Only!**

In each hour, I present code samples like this one. Often, they include line numbers so that I can easily reference the code and explain how it works. Swift does not require line numbers, nor will the code work if you leave them in your application. If you see a line prefixed with a number and a colon (#:), don't type the line number prefix!

---

**The import Declaration**

```
1: import UIKit
```

First, in line 1, the interface file uses the `import` declaration to include any other files that our application will need to access. The string `UIKit` designates a system framework that gives us access to a vast majority of the classes.

Whenever we need to import something, I explain how and why in the text. The UIKit example is included by default when Xcode sets up your classes and covers most of what you need for this book's examples.

## NOTE

---

Wait a sec, what's a declaration? *Declarations* are commands that are added to your files that introduce a new object or feature within the application. They don't implement the logic that makes your app work, but they are necessary for providing information on how your applications are structured so that Xcode knows how to deal with them.

---

## The class Declaration

The class declaration, shown in line 3, tells Xcode what class the file is going to be implementing. In this case, the file should contain the code to implement `myClass`:

```
3: class myClass: myParent, myProtocol {
```

Notice that line 3 includes a few additional items as well: that is, `myParent`, `myProtocol`. The class name (`myClass`) is always followed by a colon (`:`) and a list of the classes that this class is inheriting from (that is, the *parent classes*) and any *protocols* it will be implementing. In this example, `myClass` is going to be inheriting from `myParent` and will be implementing the `myProtocol` protocol.

The `class` line ends with an opening curly brace `{`. All blocks of code are contained in curly braces. The rest of the class code will follow this brace and eventually be terminated with a closing brace `}` (line 29).

## NOTE

---

Protocol? What's a protocol? Protocols are a feature of Swift that sound complicated but really aren't. Sometimes you will come across features that require you to write methods to support their use, such as providing a list of items to be displayed in a table. The methods that you need to write are grouped together under a common name; this is known as a *protocol*.

Some protocol methods are required, and others are optional; it just depends on the features you need. A class that implements a protocol is said to *conform* to that protocol.

---

## Variable Properties Declarations

Lines 5–6 declare three different variable properties. A variable property is just a variable that can be accessed from any method in the class, or from code within other classes.

```
5:     var myString: String = ""
6:     var myOtherString: String?
7:     var yetAnotherVariable: Float!
```

In this example, a variable named `myString` that contains a `String` is declared and initialized with an empty string (`""`). A second `String` (`myOtherString`) is also declared, but designated as “optional” with the `?` modifier. A third variable property, `yetAnotherVariable`, is declared as a floating-point number and set to be “implicitly unwrapped” by including the `!` modifier. We’ll get to the point of these modifiers in a little bit. (They look confusing, but they have an important role to play.)

## NOTE

---

To retrieve a variable property from an object, you write `<objectname>.<variable property>` to access it. That means that if there is a property `myProperty` in an object `myAmazingObject`, you type `myAmazingObject.myProperty`. This is known as *dot notation*.

What if you want to access the variable property from inside the class where it is defined? Simple. You just refer to it by name (for example, `myProperty`). If you want to be overly pedantic, you can also use `self` to refer to the current object, as in `self.<variable property>`.

This will all be obvious once you start coding.

---

## Getters and Setters/Accessors and Mutators

Variables in Swift don’t necessarily just store and retrieve static data. Your variables can declare their own methods that define the information that is returned or committed to memory. This is done by getters and setters (also called accessors and mutators). A “time” variable, for example, might not store the time at all, but instead declare a custom “getter” that retrieves the time from the system clock when it is accessed. We’ll see this behavior a bit later in the book.

---

## A Constant Declaration

Just below the variable properties is a constant declaration:

```
8:      let myAge: Int = 29
```

This creates a constant (`myAge`) and sets it to the integer value 29. Constants declared alongside variable properties (that is, outside of a method) are nearly identical to variable properties in how they are used—but with one important difference—they can’t be changed or reassigned. In other words, I’m 29 forever.

## IBOutlet Declarations

Lines 9–10 are, yet again, variable property declarations, but they include the keyword `IBOutlet` at the start. This indicates that they are going to be connected to objects defined within an application’s user interface:

```
10:     @IBOutlet weak var userOutput: UILabel!  
11:     @IBOutlet var anotherUserOutput: UILabel!
```

You learn more about `IBOutlet` in Hour 5, “Exploring Interface Builder.”

## TIP

The attribute `weak` that is provided with the `variable` declaration tells Xcode how to treat the object when it isn't in use. The `weak` attribute informs the system that the object it is referring to can be cleaned up from memory when it isn't being used anymore. It also avoids what is called a *circular reference*, where an object can't be removed from memory, because it points to another object that, in turn, points back to it. In general, try to declare your variables with the `weak` attribute.

Unfortunately, sometimes the system may be a bit overzealous in its desire to keep things clean for us, and we need to leave `weak` out of the picture—thus creating a strong reference. (Lines 5–7, 8, and 11 all declare strong references.) A strong reference means that the object will be kept in memory unless we explicitly tell the system to remove it or the object that contains it is removed from memory. It's pretty rare that we need to worry about these, but I'll point it out when it's a concern.

## Declaring Methods

The final pieces of the class file are the method declarations. Lines 13, 17, 21, and 25 declare four methods that will be implemented in the class:

```
13:     class func myTypeMethod(aString: String) -> String {
14:         // Implement a Type method here
15:     }
16:
17:     func myInstanceMethod(myString: String, myURL: NSURL) -> NSDate? {
18:         // Implement the Instance method here
19:     }
20:
21:     override func myOverriddenInstanceMethod() {
22:         // Override the parent's method here
23:     }
24:
25:     @IBAction func myActionMethod(sender: AnyObject) {
26:         // React to an action in the user interface
27:     }
```

Method declarations follow a simple structure. They begin with the word `func`, but can include the prefix modifiers `class` and `override`. A method that begins with `class func` is a *Type* method (often also referred to as a *Class* method). A method starting with `override func` is one that is redefining a method already provided in a parent class. This indicates that rather than inheriting functionality from a higher class, we're going to write our own logic.

In the example file, line 13 defines a *Type* method named `myTypeMethod` that returns a `String` and accepts a `String` as a parameter. The input parameter is made available in a variable called `aString`.

Line 14 defines an instance method named `myInstanceMethod` that returns an *optional* `NSDate` object, taking a `String` and an `NSURL` as parameters. These are made available to the code in the method via the variables `myString` and `myURL`. I'll rant about what optional values

are and how to deal with them later in the hour. For the moment, just understand that by saying this method has an optional return type, it may return an `NSDate` object, or nothing (`nil`).

Line 21 declares an instance method, `myOverriddenInstanceMethod`, that takes no parameters and returns no results. What makes this interesting is that it uses the keyword `override` to indicate that it will be replacing a method provided by the parent class (`myParent`). When you start defining methods in your classes, Xcode knows what methods are inherited, so the moment you go to define a method provided by a parent class, it will automatically add the `override` keyword for you.

The fourth instance method, `myActionMethod`, declared in line 25 differs from the others because it defines an *action*, as indicated by the `@IBAction` keyword. Methods that begin with `@IBAction` are called when the user touches a button, presses a switch, or otherwise interacts with your application's user interface (UI). These methods take a single parameter, usually denoted as `sender` that is set to whatever object the user interacted with. Just as with the `@IBOutlet` mentioned earlier, you'll be learning much more about `IBAction` in Hour 5.

---

#### TIP

You will often see methods that accept or return objects of the type `AnyObject`. This is a special type in Swift that can reference *any* kind of object and proves useful if you don't know exactly what you'll be passing to a method or if you want to be able to return different types of objects from a single method.

---

---

#### TIP

You can add a text comment on any line within your class files by prefixing the line with two forward slash characters: `//`. If you want to create a comment that spans multiple lines, you can begin the comment with the characters `/*` and end with `*/`.

---

## Ending the Class File

To end the class file, you just need a closing brace: `}`. You can see this on line 29 of the example file:

```
29: }
```

Although this might seem like quite a bit to digest, it covers almost everything you'll see in a Swift class file.

## Public Versus Private

If you've worked in other languages, you might be familiar with the concepts of *public* versus *private* classes, methods, and variables. This lets you limit what can be accessed within a given class. Swift does not (currently) have any consistent way to provide this functionality, but Apple assures us it is coming. Thankfully, this is really only a concern when sharing code; so, it isn't something that will really impact most projects.

---

## Structure for Free

Even though we've just spent quite a bit of time going through the structure of a Swift class file, you're rarely (if ever) going to need to type it all out by hand. Whenever you add a new class to your Xcode project, the structure of the file will be set up for you. What's more, much of the work of declaring variable properties and methods can be done visually. Of course, you still need to know how to write code manually, but Xcode goes a long way toward making sure that you don't have to sweat the details.

## When Is a Class Not a Class?

Not every "object" that you use in Swift will actually *be* an object. Swift includes data structures called `structs` that you can think of as lightweight objects. A `CGRect`, for example, defines a rectangle on the screen and has a variety of variable properties used to describe it. You can use data structures just like objects, and rarely give it another thought.

The biggest difference, from the perspective of a developer using an object versus a struct, is that if you assign any one object to two variables, both variables will reference the same object. When you assign a structure to two variables, each variable gets a unique copy of that structure.

---

# Swift Programming Basics

We've explored the notion of classes, methods, and instance variables, but you probably still don't have a real idea of how to go about making a program do something. So, this section reviews several key programming tasks that you'll be using to implement your methods:

- ▶ Declaring variables and constants
- ▶ Understanding built-in swift data types
- ▶ Making sense of optional values
- ▶ Initializing objects
- ▶ Using an object's instance methods
- ▶ Making decisions with expressions
- ▶ Branching and looping

## Declaring Variables and Constants

Earlier we documented what variable properties will look like in your Swift files, but we didn't really get into the process of how you declare them (or use them). Nor did we talk about variables *within* methods!

Whatever the purpose, you declare your variables using this syntax:

```
var <Variable Name>[: Variable Type][Optional modifier] [ = Initialization]
```

Holy cow—that looks complicated! In practice, nearly everything but the `var` keyword and the variable name are optional. I make a point of always trying to provide the variable type, but if you don't, Swift will try to figure it out for you. The type is either a Swift data type or the name of a class that you want to instantiate and use.

Let's begin by taking a look at a few Swift data types and how they are declared and used.

### Swift Data Types

Swift includes a number of data types that enable you to work with common types of information. Most basic logic that we implement in this book will take advantage of one of these data types:

- ▶ **Int:** Integers (whole numbers such as 1, 0, and -99).
- ▶ **Float:** Floating-point numbers (numbers with decimal points in them).
- ▶ **Double:** Highly precise floating-point numbers that can handle a large number of digits.
- ▶ **String:** Collections of characters (numbers, letters, and symbols). Throughout this book, you'll often use strings to collect user input and to create and format user output.
- ▶ **Bool:** A Boolean value (that is, `true` or `false`), often used to evaluate conditions within your code.
- ▶ **Arrays:** A collection of ordered values that are accessed via a numeric index.
- ▶ **Dictionaries:** A collection of key/value pairs. A given value is accessed by providing its key.

### Integers and Floating-Point Numbers

Let's start with something easy: integers (`Int`) and floating-point numbers (`Float` or `Double`). To declare an integer variable that will hold a user's age, you might enter the following:

```
var userAge: Int
```

If you wanted, you could even initialize it with a value, all in the same line:

```
var userAge: Int = 30
```

After a variable is declared, it can be used for assignments and mathematical operations. The following code, for example, declares two variables, `userAge` and `userAgeInDays`, and uses the first (an age in years) to calculate the second (the age in days):

```
var userAge: Int = 30
var userAgeInDays: Int
userAgeInDays = userAge * 365
```

Notice that for the `userAgeInDays`, I declare it, then use it later. You're welcome to do this, or declare and initialize the variables on the exact same line.

Floating-point numbers work the same way—you declare them, then use them. A simple example to calculate the circumference of a circle (circumference = diameter \* 3.141), for example, could be written like this:

```
var diameter: Float = 2.5
var circumference: Float = diameter * 3.141
```

Pretty easy, don't you think? Swift data types have much more to offer as well. Let's see what else they can do!

## NOTE

---

As I said earlier, everything but the `var` keyword and the variable name is optional in a variable declaration. For example, the age calculation code could be written to leave out the variable type entirely:

```
var userAge = 30
var userAgeInDays = userAge * 365
```

Swift will automatically figure out what the variable is based on the initialization. Personally, I prefer including the variable type so that I can quickly see what each variable represents in my code.

---

## Strings

Strings are one of the most frequently used Swift types in this book. You'll be using strings for user input and output, data manipulation, and so on. As with every other variable, the life of a string begins with a declaration and an initialization:

```
var myName: String = "John"
```

Here, a string (`myName`) is initialized to the value `"John"`. Once initialized, the string can be manipulated using a number of techniques. String *concatenation* (adding two or more strings together) is performed with the addition (+) operator. To change `myName` to include my last name, I'd write the following:

```
myName = myName + " Ray"
```

You can even use a process called *string interpolation* to combine existing Strings, values returned by methods, and other Swift data types into a new String. Consider this line:

```
var sentence: String = "Your name is \(myName) and you are \(userAge) years old"
```

Here I've combined the `myName` string and `userAge` integer into a single string, assigned to a new variable named `sentence`. Any time Swift encounters the pattern `\(<variable or method name>)` in your code, it takes the result, turns it into a string, and substitutes it in place of the pattern. You can use this to quickly format strings based on other variables and methods.

In many languages, strings require special functions to check for equality. In Swift, the same comparison operators you'd use to compare two numbers also work for strings. We'll look at comparisons a bit later.

## Boolean Values

A Boolean value has only two states—represented by `true` or `false` in Swift. Booleans are most often used in comparisons, although some methods have Boolean parameters that you'll need to supply. As expected, Booleans are initialized using the same pattern you've seen for numbers and strings:

```
var myFlag: Bool = false
```

## Arrays

A useful category of data type is a collection. Collections enable your applications to store multiple pieces of information in a single object. An `Array` is an example of a collection data type that can hold multiple objects, accessed by a numeric index.

You might, for instance, want to create an array that contains all the user feedback strings you want to display in an application:

```
var userMessages: [String] = ["Good job!", "Bad Job", "Mediocre Job"]
```

Notice that the word `Array` doesn't even appear in the declaration and initialization? That's because all we need to do to declare an array is wrap the type we want to store (in this case, `String` values) in square brackets. If I wanted an array of integers, I'd use a type of `[Int]` and so on. The initialization values are provided as a comma-separated list enclosed in square brackets; if you use `[]` alone, the array is initialized as empty.

To access the strings in the `userMessages` array, you use an index value. This is the number that represents a position in the list, starting with 0. To return the "Bad job" message, we use the number 1 (the second item in the list):

```
userMessages[1]
```

You can also use the index to assign values to an array, replacing what is currently stored:

```
userMessages[1] = "Try again"
```

Swift lets you add new items to the end of the list using the array's `append` method. For example, to add a new message ("Meh") to the end of `userMessages`, I might write the following:

```
userMessages.append("Meh")
```

There are several other means of accessing and modifying arrays that we'll use over the course of the next 21 hours.

## Dictionaries

Like arrays, dictionaries are another collection data type, but with an important difference. Whereas the objects in an array are accessed by a numeric index, dictionaries store information as key/value pairs. The key is usually an arbitrary string, whereas the value can be anything you want, even objects. If the previous `userMessages` array were to be created as a `Dictionary` instead, it might look like this:

```
var userMessages: [String:String] =  
    ["positive": "Good job!", "negative": "Bad Job", "indifferent": "Mediocre Job"]
```

Similar to declaring the strings, I declare the dictionary without ever using the word *dictionary*. Instead, I provide the type data types that will form the keys and values within square brackets—for example, [`<key data type>`:`<value data type>`]. For the `userMessage` dictionary, I'm using keys that are strings, and values that are strings. The initialization is similar to an array, but consists of the key, a colon (:), and then the value. Each key/value pair is separated by a comma. Empty dictionaries can be created with the initializer `[:]`.

To access a value in the dictionary, I index into `userMessages` just like an array, but using the key rather than an integer. To access the "Bad Job" message (tied to the "negative" key), I could type the following:

```
userMessages["negative"]
```

Keys can also be used to modify or assign new values. Here the key "apathy" is assigned the value "Meh":

```
userMessages["apathy"] = "Meh"
```

Dictionaries are useful because they let you store and access data in abstract ways rather than in a strict numeric order.

---

**TIP****Counting the Contents**

Both dictionaries and arrays include a read-only variable property called `count` that returns the number of elements they've stored. The number of elements in the `userMessages` array (or dictionary), for example, can be accessed with the expression: `userMessages.count`.

---

**Object Data Types**

Just about everything that you'll be working with in your iOS applications will be an object. Onscreen text, for example, will be instances of the class `UILabel`. Buttons that you display are objects of the class `UIButton`. You'll learn about several of the common object classes in the next hour's lesson. Apple has literally provided thousands of different classes that you can use to store and manipulate data.

Objects are declared and initialized just like Swift data types. For example, to declare and create a new instance of the `UILabel` class, you could use the following code:

```
var myLabel: UILabel = UILabel()
```

Here, the initializer is `UILabel()`. This returns a new, ready-to-use instance of the `UILabel` class. You can initialize all classes using this same syntax `<class name>()`, but most will require additional setup after initialization. To speed things along, many will provide *convenience* methods. Convenience methods speed the process of creating a new object by taking the basic parameters needed to create and configure the object, all at once.

---

**NOTE**

When you read through the Xcode documentation (discussed in the next hour), you'll see initialization methods denoted with the function name `init` for Swift. This is the internal method name in the class. It is automatically invoked by using the `<class name>()` syntax.

---

**Convenience Methods**

When we initialized the `UILabel` instance, we did create an object, but it doesn't yet have any of the additional information that makes it useful. Attributes such as what the label should say, or where it should be shown on the screen, have yet to be set. We would need to use several of the object's other methods to really turn it into something ready to be displayed.

These configuration steps are sometimes a necessary evil, but Apple's classes often provide a special initialization method called a *convenience method*. These methods can be invoked to set up an object with a basic configuration so that it can be used almost immediately.

For example, the `NSURL` class, which you use later to work with web addresses, defines a convenience method called `initWithString`. We can use it to create a brand-new `NSURL` object, complete with the URL, just by typing the following:

```
var iosURL: NSURL = NSURL(string: "http://www.teachyourselfios.com/")!
```

This is where we (briefly) go off the tracks. Notice that nowhere in that line does `initWithString` appear. The `initWithString` method is the name of a convenience method in *Objective-C*. The method still goes by the same name when used in Swift, but it takes on a simplified form.

The general rule of thumb is that, in Swift, the `initWith` is removed from the name of convenience method. Whatever remains of the name becomes the first *named* parameter of the method. A named parameter, as you'll learn a bit later, is a parameter that requires you to spell out its name in the method call (in this case, `string`).

Because Xcode supports autocompletion, it is usually pretty easy to start typing in a method named and find it in the list that appears. Just keep in mind that what you see in the Xcode documentation doesn't necessarily apply to both Objective-C and Swift.

## Type Conversion and Type Casting

In your adventures in Swift, you will encounter code that doesn't quite work the way you want. You'll find legacy `CGFloat` floating-point numbers that must be used in place of Swift `Float`. You'll find places where you need to turn `Floats` into `Ints`, and vice versa. You'll even encounter objects that have no idea what they are. To get around these little snags, you'll likely employ type conversion, or type casting.

### Type Conversion

For most of the simple data types, you can convert between types by using the syntax: `<TypeName>(<Value to Convert>)`. For example, if a method calls for a `CGFloat` and you have a `Float` value (in the variable `myFloat`), you can convert it to the proper type with the following:

```
CGFloat(myFloat)
```

Swift does everything it can to silently bridge these older data types with the new built-in Swift types—and it is usually very successful. Unfortunately, sometimes this manual conversion will have to happen.

Another common circumstance is when a method returns an object of one type when it needs to be another. When this happens, you must type cast the result.

## Type Casting

Type casting takes an object of a higher-level class and tells Xcode which specific subclass it should be. Some methods will return an object of the type `AnyObject` rather than a specific type. Does this make any sense? Not really, but it happens often.

For example, the `NSDate` class includes several methods that return a date, but instead of being of the type `NSDate`, they are of the type `AnyObject`. The `NSDate` type method `distantPast` is one of these methods:

```
var myPastDate: NSDate = NSDate.distantPast() as NSDate
```

Because `distantPast()` results in an object of the type `AnyObject`, we must “tell” Xcode that it is really an `NSDate` by adding `as NSDate` to the end of the assignment. Using the syntax `as <class name>` after any object will attempt to type cast that object as being of whatever class you name.

After a variable is cast to an object of the correct type, we can interact with it directly as that type. This looks a bit unusual, I know, but it will come in handy later in the book. It’s easier to understand when you see it in an actual application; so for the moment, just be aware that it is an available development tool.

## Constants

Constants are declared and initialized just like variables, except they begin with the keyword `let`. For example, to create a constant named `lastName` that holds a `String` with my last name, I would write the following:

```
let lastName: String = "Ray"
```

The key difference between a constant and a variable is that constants, once assigned, cannot be changed or reassigned. This, however, isn’t as limiting as you might think. When you assign an object to a constant, you can access and modify all the variable properties in that object, execute all its methods, and so on. It can still be used just like any other variable—you just can’t reassign it later.

Constants are more efficient than variables and should be used in their place wherever possible. I think you’ll be surprised to find that we use more constants in our applications than actual variables.

## Optional Values

Possibly the most confusing, infuriating thing about Swift is the notion of optional values. In theory, it’s really quite simple. If you’ve developed in other languages, you’ve almost certainly written code that *thought* it was working with a value, only to find that the value had never been

set or that a method that was supposed to return a value didn't. Making the assumption that we *know* what is in a variable is dangerous, but it's something that developers do every day.

In Swift, Apple decided that developers should acknowledge when they're using a value that might not contain what they expect. The result requires interesting additions to the development process:

1. Method, variable, and constant declarations should state when they *may not have, or may not return*, a value. These are known as *optional* values.
2. Why would a method programmed to return a result ever make that result optional? If the method has bad input, or otherwise can't complete the operation it is tasked with performing, it makes perfect sense to return "nothing"—represented in Swift using the keyword `nil`.
3. When attempting to access methods or variables that are optional, developers must `unwrap` the values. This means that the developer acknowledges that he or she knows what is in a variable (or returned by a method), and wants to access and use the results.

Now, you might think to yourself, "Hey, I know what I'm doing, I'm not going to write any code where I name a variable or method return type as optional! That would just be extra work!" You're right, it is extra work—but it's utterly unavoidable.

All the code that makes up the Cocoa Touch classes is being updated by Apple to denote which variable properties and methods return optional values—and there are *many* (and the list is growing). I could tell you stories about the number of times I've opened a project while writing this book, only to find that Apple has changed a class somewhere that breaks the code I've written. That's one of the difficulties of being an early technology adopter.

Okay, enough ranting. What does all of this actually mean in terms of coding?

### Declaring Optionals

First, when declaring a variable, you can define it as optional by adding a `?` after the type name:

```
var myOptionalString: NSString? = "John"
```

This also means that if the string isn't immediately initialized, it automatically contains the value `nil`.

---

### NOTE

#### What the... It's an Optional Value, But It Has a Value (`nil`)?!

Yes, this is as weird as it sounds. In Swift, `nil` represents literally *nothing*. When we need some value to represent no value at all, `nil` is used. We can't use something like an empty string (`" "`) or `0` because those *are* values. Get used to the idea of `nil`, because even though it is something, it is also nothing. (Cue *Seinfeld* music.)

---

For method definitions, you denote an optional return value by adding `?` after the return type. In the sample class in Listing 3.1, I defined the method as having an optional return value of `NSDate` using this syntax in the declaration:

```
func myInstanceMethod(myString: String, myURL: NSURL) -> NSDate? {
```

## TIP

---

### Optional Downcasting

You've seen the syntax for downcasting, but consider what happens if the class you're downcasting *cannot* be cast to what you want. In this case, your app is likely going to crash. To deal with this scenario, you can create an *optional downcast*. With an optional downcast, if the downcast fails, the resulting variable will contain `nil`.

To define a downcast operation as optional, simply add a `?` to the end of the `as` keyword, as follows:

```
var myPastDate NSDate? = NSDate.distantPast() as? NSDate
```

---

## NOTE

---

Constants can also be assigned as optional using the same syntax as variables. Although this might seem counterintuitive (don't you assign a value when you create a constant?), it makes sense when you consider that you might be assigning a constant to the return value of a method with an optional return type.

---

### Unwrapping and Implicit Unwrapping

After you've either created (or encountered Swift variables and methods) that are optional, you need to know how to access them. Accessing optional values is called *unwrapping* in Swift. The easiest, most brute-force way is to use optional values is to *unwrap* them by adding an exclamation mark (!) to the end of their name.

In other words, each time I wanted to use the value in `myOptionalString`, I would reference it as follows:

```
myOptionalString!
```

The same goes for the `myInstanceMethod` method. To use it, I might write a line like this:

```
var myReturnedDate: NSDate = myInstanceMethod("A cool string", myURL: NSURL)!
```

The addition of the `!` tells Xcode that we want to access the return value and that we don't care if it is `nil`. We can take this a step further by defining what is called an *implicitly unwrapped*

optional. This is just an optional value that will *always* be unwrapped automatically when we use it.

To create an implicitly unwrapped variable, you add a `!` after the type name. For example, I could write the preceding line of code using an implicitly unwrapped variable, like this:

```
var myReturnedDate: NSDate! = myInstanceMethod("A cool string", myURL: iOSURL)
```

This declares `myReturnedDate` as an optional `NSDate` variable, but one that will be implicitly unwrapped. I can assign it the result of an optional method without unwrapping the return value of the method (because both are optional). However, when I go to use `myReturnedDate` elsewhere in my code, it will automatically be unwrapped for me—just as if I had put the `!` after it each time.

You really won't be doing this very often, but Xcode is going to do it *a lot* when it writes code for you. Why? Because every interface object that connects to your code will be referenced through an implicitly unwrapped variable. An interface object may be `nil` before it is loaded, so it has to be optional; but once your code is active, it should always have a value, and there's no point in hindering its use—thus, it is implicitly unwrapped for you.

### Optional Binding

Another (gentler) way to deal with optional values is called *optional binding*. This is the assignment of an optional value to a constant. If the assignment succeeds, the optional value is accessible through the constant. If it fails, the optional value is `nil`.

Applying optional binding to the `myOptionalString` variable, I might write this simple logic to test to see whether an optional value should be used:

```
if let stringValue:String = myOptionalString {  
    // myOptionalString has a non-nil value.  
}
```

This is a good approach for working with optionals in production-ready code. It gives you an opportunity to react to situations where optionals are set to `nil` and errors may have arisen. If you unwrap an optional and try to work with it even if it is `nil`, you may crash your code.

For most of the examples in the book, I manually unwrap values with `!` because the code is simple and we know how the different components are going to interact. In apps bound for the App Store, I recommend using optional binding to trap for error conditions that you may not have anticipated.

---

**TIP****Optionals: Don't Be Scared**

Optionals exist to help protect you from making bad assumptions in your code. At times, you'll feel like every single method or variable property you use has been declared as optional—and you'll likely start to think that your hair is optional as well. The good news is that Xcode recognizes optional values throughout Cocoa Touch and will prompt you if you're missing the required `?` or `!` characters. In most cases, it will even correct simple optional unwrapping errors for you.

Don't feel like you need to start memorizing the tens of thousands of optional values in Cocoa Touch. Xcode knows, and it will let you know.

---

## Using Methods

You've already seen how to declare and initialize objects, but this is only a tiny picture of the methods you'll be using in your apps. Let's start by reviewing the syntax of calling methods in Swift.

### Method Syntax

To use a method, provide the name of the variable that is referencing your object followed by the name of the method, followed by a period, the name of the method, and empty parentheses `()` (empty if there are no parameters). If you're using a type (class) method, just provide the name of the class rather than a variable name:

```
<object variable or class name>.<method name>()
```

Things start to look a little more complicated when the method has parameters. A single parameter method call looks like this:

```
<object variable or class name>.<method name>([parameter:]<parameter value>)
```

Earlier I noted that convenience initialization methods will usually include at least one named parameter, such as `string` when initializing an `NSURL` object:

```
var iOSURL: NSURL = NSURL(string: "http://www.teachyourselfios.com/")!
```

This is important to note because the style of using an initial named parameter is only really used in convenience initialization methods. In other (general use) methods, the first parameter is just provided as a value.

---

**TIP**

If you aren't sure whether the first parameter to a method is named or not, the Xcode documentation can help. If the first character after the parenthesis in a Swift method definition is an underscore (`_`), that parameter is *not* named. You'll learn all about the documentation system in the next hour.

---

For example, let's look at a method that takes multiple parameters:

```
var myFullName: String = "John Ray"
var myDescription: String =
myFullName.stringByReplacingOccurrencesOfString(myFullName, withString: "is awesome!")
```

This code fragment stores my name in the `myFullName` variable, then uses the `stringByReplacingOccurrencesOfString:withString` method to change my last name from “Ray” to “is awesome!”

In this example, the first parameter to the `stringByReplacingOccurrencesOfString:withString` method has no name; I just put in the value (`myFullName`). The second parameter *does* have a name (`withString:`), which must be provided along with the value.

The syntax for multiple parameter method calls looks like this:

```
<object variable or class name>.<method name>([parameter:]<parameter value>,
    <parameter>:<parameter value>, <parameter>:<parameter value> ...)
```

## NOTE

At the time of this writing, it was very difficult to break lines in Swift without literally breaking the code. I've found that you can break lines around assignment statements (`<blah> = <blah>`) as long as there are spaces around the `=`, as well as after a comma `,` in lists of parameters.

## NOTE

Throughout the lessons, methods are referred to by name. If the name includes a colon (`:`), this indicates a required named parameter. This is a convention that Apple has used in its documentation and that has been adopted for this book.

## Making Sense of Named Parameters

If you've gotten to this point and you aren't sure what a *named parameter* is, I am not surprised. To work, methods take parameter values as input. A named parameter is just a string, followed by a colon, that provides context for what a given value is or does. For example, a method that looked like this doesn't really tell you much:

```
myString.stringByReplacingOccurrencesOfString(String1, String2)
```

Is `String1` replacing `String2`? Vice versa?

By making the second parameter a *named* parameter, it becomes obvious:

```
myString.stringByReplacingOccurrencesOfString(String1, withString:String2)
```

The named parameter, `withString:`, shows us that `String2` will be used to replace `String1`. This also shows why the first parameter is *rarely* named: because the name of the method itself implies that the first parameter is a string that is going to be replaced.

## Chaining

Something that you'll see when looking at Swift code is that often the result of a method is used directly as a parameter within another method. In some cases, if the result of a method is an object, a developer may immediately use a method or variable property of that object without first assigning it to a variable. This is known as *chaining*.

Chaining results directly eliminates the need for temporary variables and can make code shorter and easier to maintain.

For example, consider this completely contrived code:

```
var myString: String = "JoHN ray"
myString = myString.lowercaseString
myString = myString.stringByReplacingOccurrencesOfString("john", withString: "will")
myString = myString.capitalizeString
```

Here I've created a string (`myString`) that holds my name with very very poor capitalization. I decide that I want to replace my first name (John) with my brother's name (Will). Because I cannot just search and replace on John because my capitalization is all messy and I don't want to try to remember how I wrote my name (this is *contrived* folks), I decide to first convert `myString` to lowercase by accessing the `lowercaseString` variable property. Once complete, I can just search for `john` and replace it with `will` without worrying about capitalization. Unfortunately, that means I still need a properly capitalized version of the string when I'm done. So, I access the `capitalizeString` variable property of my string when finished, and use its value for `myString`. (In case you're wondering, `capitalizeString` provides a copy of the string with all of the first letters capitalized.)

The code should make sense, even if my logic is a bit shaky. That said, each of the methods and variable properties I've used return a string. Instead of assigning things over and over, I can chain each of these actions together into a single line:

```
var myString: String = "JoHN ray".lowercaseString.stringByReplacingOccurrencesOf
↳String("john", withString: "will").capitalizeString
```

Chaining can be a powerful way to structure your code, but when overused it may lead to lines that can be difficult to parse. Do what makes you comfortable; both approaches are equally valid and have the same outcome.

## TIP

---

Although I tend to leave chained lines unbroken in my projects, you *can* break a chained line without causing an error if you break it immediately *before* one of the periods.

---

## Optional Chaining

Time for optionals to rear their head one more time this hour. As you've just seen, chaining can be a great way to use values without lots of temporary variables and multiple lines of code. What happens, however, if one of the values (the results of a method, or a variable property) in the middle of the chain is optional? You can unwrap the values using `!` and hope that they *do* exist, or you can take advantage of *optional chaining*. In optional chaining, you can write out your chain, placing a `?` after optional values. This allows the full chain to be evaluated, even if a value is missing somewhere in the middle. For example, assume I wrote a line of code like this:

```
myObject.optionalMethod()!.variableProperty.method()
```

If the `optionalMethod()!` in the middle didn't return what I was expecting, I wouldn't be able to access `variableProperty` or the subsequent `method()`. To get around this, I can write the chain as follows:

```
myObject.optionalMethod()?.variableProperty.method()
```

Doing this allows the line to be executed and fail gracefully. If `optionalMethod()` does not return a usable object, the entire line returns `nil`, which can be trapped and dealt with as you learned earlier.

---

## Closures

Although most of your coding will be within methods, you will also encounter *closures* when using the iOS frameworks. Sometimes referred to as *handler blocks* in the Xcode documentation, these are chunks of code that can be passed as values when calling a method. They provide instructions that the method should run when reacting to a certain event.

For example, imagine a `personInformation` object with a method called `setDisplayNames` that would define a format for showing a person's name. Instead of just showing the name, however, `setDisplayNames` might use a closure to let you define, programmatically, how the name should be shown:

```
personInformation.setDisplayName({(firstName: String, lastName: String) in
    // Implement code here to modify the first name and last name
    // and display it however you want.
})
```

Interesting, isn't it? Closures are relatively new to iOS development and are used throughout this book. You'll first encounter closures when writing alerts. The closure will provide the instructions that are executed when a person acts on an alert.

## Expressions and Decision Making

For an application to react to user input and process information, it must be capable of making decisions. Every decision in an app boils down to a true or false result based on evaluating a set of tests. These can be as simple as comparing two values, to something as complex as checking the results of a complicated mathematical calculation. The combination of tests used to make a decision is called an *expression*.

## Using Expressions

If you recall your high school algebra, you'll be right at home with expressions. An expression can combine arithmetic, comparison, and logical operations.

A simple numeric comparison checking to see whether a variable `userAge` is greater than 30 could be written as follows:

```
userAge > 30
```

When working with objects, we need to use variable properties within the object and values returned from methods to create expressions. If I have stored an `NSDate` object with my birthday in it (`myBirthday`), I could check to see whether the current day is my birthday with the expression:

```
myBirthday.isEqualToDate(NSDate())
```

Expressions are not limited to the evaluation of a single condition. We could easily combine the previous two expressions to find a person who is over 30 and is celebrating their birthday today:

```
userAge > 30 && myBirthday.isEqualToDate(NSDate())
```

## Common Expression Syntax

- ()** Groups expressions together, forcing evaluation of the innermost group first.
- ==** Tests to see whether two values are equal (for example, `userAge == 30`).
- !=** Tests to see whether two values are not equal (for example, `userAge != 30`).
- &&** Implements a logical AND condition (for example, `userAge > 30 && userAge < 40`).
- ||** Implements a logical OR condition (for example, `userAge > 30 || userAge < 10`).
- !** Negates the result of an expression, returning the opposite of the original result. (For example, `!(userAge == 30)` is the same as `userAge != 30`.)

It's good practice to put spaces on either side of the symbols you use for comparisons—especially when using `!=`. Recall that a `!` also indicates that a value should be unwrapped, so Xcode can be easily confused into thinking you want to unwrap something when really you're just testing for inequality.

---

As mentioned repeatedly, you're going to be spending lots of time working with complex objects and using the methods within the objects. You cannot make direct comparisons between objects as you can with simple data types. To successfully create expressions for the myriad objects you'll be using, you must review each object's methods and variable properties.

## Making Decisions with `if-then-else` and `switch` Statements

Typically, depending on the outcome of the evaluated expression, different code statements are executed. The most common way of defining these different execution paths is with an `if-then-else` statement:

```
if <expression> {  
    // do this, the expression is true.
```

```

} else {
    // the expression isn't true, do this instead!
}

```

For example, consider the comparison we used earlier to check a `myBirthday NSDate` variable to see whether it was equal to the current date. If we want to react to that comparison, we might write the following:

```

if myBirthday.isEqualToDate(NSDate()) {
    let myMessage: String = "Happy Birthday!"
} else {
    let myMessage: String = "Sorry, it's not your birthday."
}

```

Another approach to implementing different code paths when there are potentially many different outcomes to an expression is to use a `switch` statement. A `switch` statement checks a variable for a value and then executes different blocks of code depending on the value that is found:

```

switch (<some value>) {
    case <value option 1>:
        // The value matches this option
    case <value option 2>:
        // The value matches this option
    default:
        // None of the options match the number.
}

```

Applying this to a situation where we might want to check a user's age (stored in `userAge`) for some key milestones and then set an appropriate `userMessage` string if they are found, the result might look like this:

```

switch userAge {
    case 18:
        let userMessage: String = "Congratulations, you're an adult!"
    case 21:
        let userMessage: String = "Congratulations, you can drink champagne!"
    case 50:
        let userMessage: String = "You're half a century old!"
    default:
        let userMessage: String = "Sorry, there's nothing special about your age."
}

```

## Repetition with Loops

In some situations, you will need to repeat several instructions over and over in your code. Instead of typing the lines repeatedly, you can loop over them. A loop defines the start and end of several lines of code. As long as the loop is running, the program executes the lines from top

to bottom and then restarts again from the top. The loops you'll use are of two types: `for` loops and condition-based `while/do-while` loops.

### for Loops

In a `for` loop, the statements are repeated a (mostly) predetermined number of times. You might want to count to 1000 and output each number, or create a dozen copies of an object. These are perfect uses for a `for` loop.

The `for` loop you'll encounter most often consists of this syntax:

```
for <initialization>;<test condition>;<count update> {
    // Do this, over and over!
}
```

The three “unknowns” in the `for` statement syntax are a statement to initialize a counter variable to track the number of times the loop has executed, a condition to check to see whether the loop should continue, and finally, an increment for the counter. A loop that uses the integer variable `count` to loop 50 times could be written as follows:

```
for var count=0;count<50;count=count+1 {
    // Do this, 50 times!
}
```

The `for` loop starts by setting the `count` variable to 0. The loop then starts and continues as long as the condition of `count<50` remains `true`. When the loop hits the bottom curly brace (`}`) and starts over, the increment operation is carried out and `count` is increased by 1.

### NOTE

---

Integers are usually incremented by using `++` at the end of the variable name. In other words, rather than using `count=count+1`, most often you'll encounter `count++`, which does the same thing. Decrementing works the same way, but with `--`.

---

`for` loops can also iterate over collections using the following syntax:

```
for <variable> in <collection> {
    // Do this for each value in the collection, where <variable> contains the value
}
```

Consider the array of messages we created earlier in the hour:

```
var userMessages: [String] = ["Good job!", "Bad Job", "Mediocre Job"]
```

To loop over this array of messages, we can write a “`for in`” loop as follows:

```
for message in userMessages {
    // The message variable now holds an individual message
}
```

The same applies to dictionaries as well, but the syntax changes just a little bit. If `userMessages` is defined as a dictionary:

```
var userMessages: [String:String] =
    ["positive": "Good job!", "negative": "Bad Job", "indifferent": "Mediocre Job"]
```

We can loop over each key/value pair like this:

```
for (key, value) in userMessages {
    // The key and value variables hold an individual dictionary entry
}
```

### **while and do-while Loops**

In a condition-based loop, the loop continues while an expression remains true. You'll encounter two variables of this loop type, `while` and `do-while`:

```
while <expression> {
    // Do this, over and over, while the expression is true!
}
```

and

```
do {
    // Do this, over and over, while the expression is true!
} while <expression>
```

The only difference between these two loops is when the expression is evaluated. In a standard `while` loop, the check is done at the beginning of the loop. In the `do-while` loop, however, the expression is evaluated at the end of every loop. In practice, this difference ensures that in a `do-while` loop, the code block is executed at least once; a `while` loop may not execute the block at all.

For example, suppose that you are asking users to input their names and you want to keep prompting them until they type John. You might format a `do-while` loop like this:

```
do {
    // Get the user's input in this part of the loop
} while userName != "John"
```

The assumption is that the name is stored in a string called `userName`. Because you wouldn't have requested the user's input when the loop first starts, you would use a `do-while` loop to put the test condition at the end.

Loops are a very useful part of programming and, along with the decision statements, will form the basis for structuring the code within your object methods. They allow code to branch and extend beyond a linear flow.

Although an all-encompassing picture of programming is beyond the scope of this book, this should give you some sense of what to expect in the rest of the book. Let's now close out the hour with a topic that causes quite a bit of confusion for beginning developers: memory management.

## Memory Management and Automatic Reference Counting

In the first hour of this book, you learned a bit about the limitations of iOS devices as a platform. One of the biggies, unfortunately, is the amount of memory that your programs have available to them. Because of this, you must be extremely judicious in how you manage memory. If you're writing an app that browses an online recipe database, for example, you shouldn't allocate memory for every single recipe as soon as your application starts.

In the latest Xcode releases, Apple has implemented a new compiler called LLVM, along with a feature known as Automatic Reference Counting (ARC). ARC uses a powerful code analyzer to look at how your objects are allocated and used, and then it automatically retains and releases them as needed. When nothing is referencing an object, ARC ensures it is automatically removed from memory. No more `retain` or `release` messages to be sent, no more phantom crashes and memory leaks; you just code and it works.

For most objects you declare and use in a method, you do not need to do anything; when the method is finished, there are no more references to the object, and it is automatically freed. The same goes for variable properties you've declared with the `weak` attribute. Of course, it's hyperbole to say that errors won't happen with ARC; we have to use strong references in a few places in this book to keep iOS from deciding that we have finished with an object before we actually do.

When using a variable property that has a strong reference, you should tell Xcode that you're finished using an object if you want it removed from memory. How do you do that? Easy: by setting its reference to `nil`.

For example, assume we've created a giant object called `myMemoryHog`:

```
var myMemoryHog: SomeHugeObject? = SomeHugeObject()
```

To tell Xcode when we're done using the object and let it free up the memory, we would type the following:

```
myMemoryHog = nil
```

Once the huge object isn't directly reference by any variables, it can be removed from memory, and all will be well with the world.

You've learned quite a bit in this hour's lesson, and there are plenty of places for even the most experienced developer to make mistakes. As with everything, practice makes perfect, which is why our final topic focuses on a tool that makes practicing Swift *fun*.

## Introducing the iOS Playground

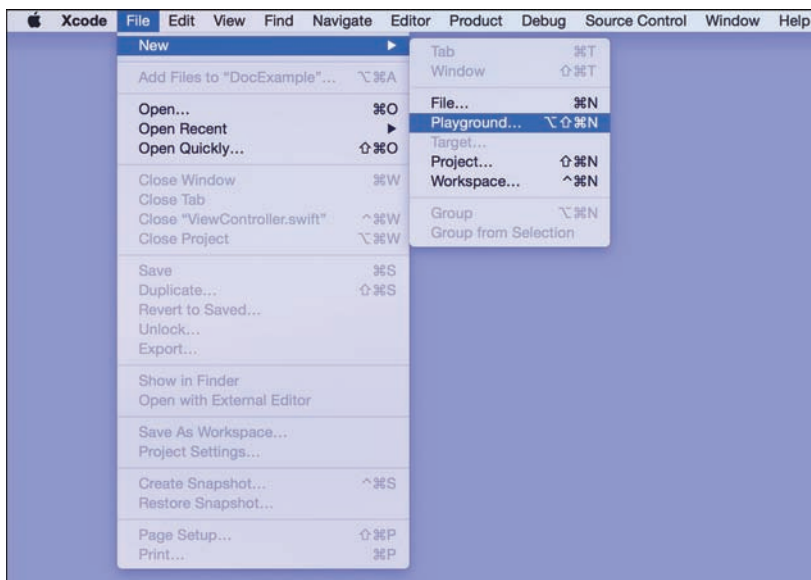
For a new developer, getting started with a language can be a pain. You’ve got to figure out how to create new projects, understand how a bunch of development tools work, and if you’re lucky, after a few hours you might get “Hello World” to display on your screen.

When Apple introduced Swift in Xcode 6, they realized that developers would need a way to get their feet wet (or hands dirty, if you prefer) without all the steps of creating new iOS applications. Heck, why would you want to try building an application if you aren’t sure you’re even going to be writing code that works? Therefore, the iOS Playground was born. The Playground gives you an area to type in experimental code and see the result—*immediately*—without even pressing a Run button.

### Creating a New Playground

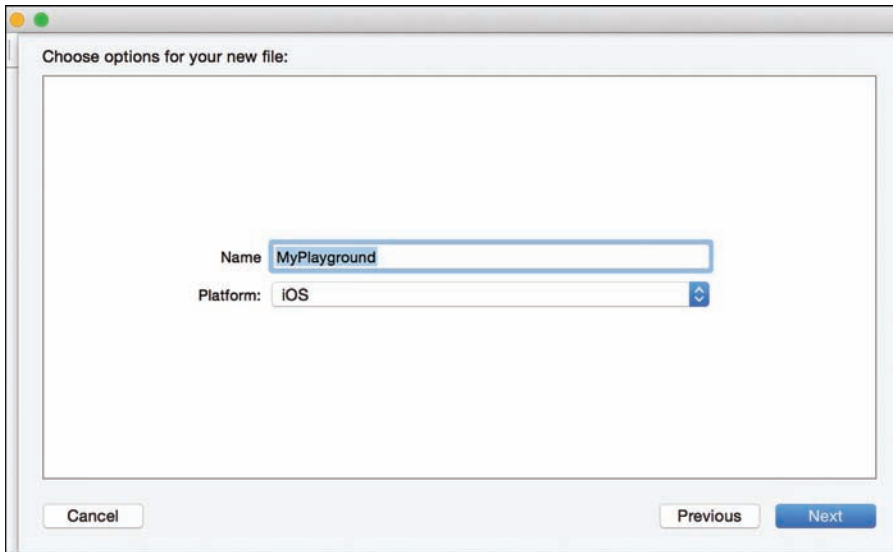
As a first step, you’ll need to create a new Playground. We’ll be using the Playground throughout the book, so understanding this process is a must.

To create a new Playground, choose File, New, Playground from the Xcode menu, as shown in Figure 3.2.



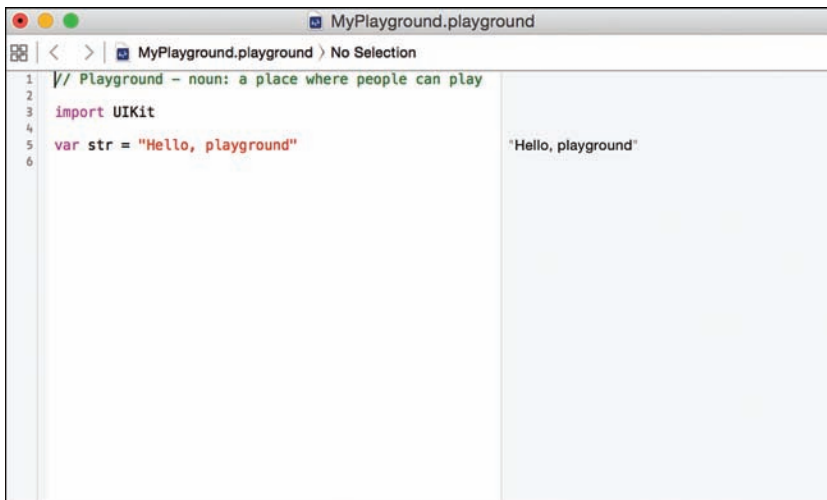
**FIGURE 3.2**  
Create a new Playground from the Xcode File, New menu.

When prompted, provide a name for the playground and make sure that the platform is set to iOS, as demonstrated in Figure 3.3. The name can be anything you’d like. Unlike a project, a Playground creates a single file, so it’s easy to rename later. Click Next to continue.

**FIGURE 3.3**

Name the playground and set the Platform to iOS.

Finally, choose where the Playground will be saved, and then click Create. After a few seconds, the Playground window opens, as shown in Figure 3.4.

**FIGURE 3.4**

The new playground opens, already populated with some sample code.

## Using the Playground

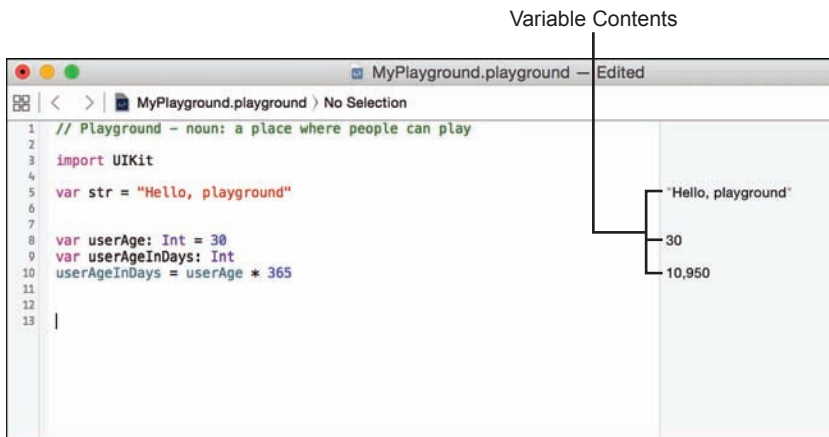
When the Playground first opens, it already contains some sample code that imports the `UIKit` framework and defines a variable named `str`. You can safely delete the `str` line if you'd like, but I recommend that you leave the `import` statement at the top. This adds access to most of the objects and methods you'll need to actually do anything useful in the playground.

So, what do you do now? You *play*. Any code that you enter is immediately evaluated and the results appear in the margin on the right.

For example, remember the calculation of a person's age in days? Try typing this code into the playground:

```
var userAge: Int = 30
var userAgeInDays: Int
userAgeInDays = userAge * 365
```

As you type the lines, watch what happens in the margin on the right. For the first statement, declaring `userAge`, you'll see 30 appear in the margin because the variable contains the value 30. The second line won't generate any output because it doesn't contain anything yet. The third line, however, calculates and stores the age in days, which is displayed in the margin (10,950, if you're interested). You can see this in Figure 3.5.



**FIGURE 3.5**

The contents of your variables are shown on the right.

## Generating and Inspecting Output

The iOS Playground can be used to generate and display output as well as inspect variables. Later in the book, we'll use it to examine the contents of web pages retrieved by our code (and in a variety of other exercises). A simple way to generate output in Swift, however, is to use the

`print(<String>)` or `println(<String>)` functions. These take a string as an argument and print it—with `println` adding return at the end.

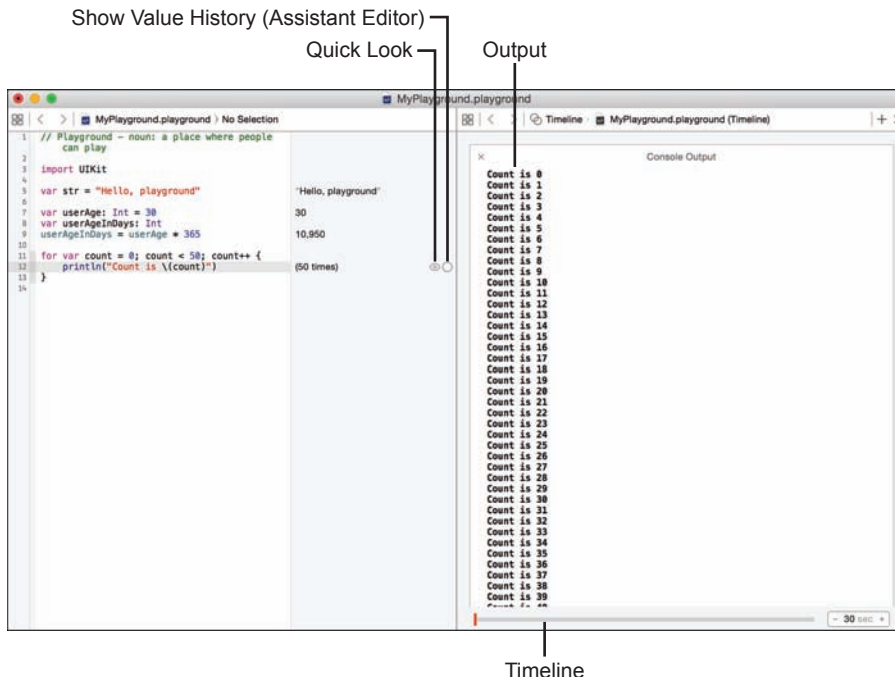
Add these lines to the end of the code in the Playground:

```
for var count = 0; count < 50; count++ {  
    println("Count is \(count)")  
}
```

Shortly after typing the lines, you'll see a value in the margin quickly count to 50. This is a count of the number of times the `println` statement was executed. What you don't see, however, is the output that is generated.

To view the output, position your cursor over the line that reads “(50 times).” You'll notice two icons appear; the first (an outline of an eye) is the Quick Look icon and lets you inspect some variable contents more closely. (We'll do this in the next hour.) The second icon, a circle, opens the assistant editor and displays a value history for that item, as well as any output that was generated.

Go ahead and click the circle by the `println` statement. The assistant editor displays the output from the `println` statement, as well as a draggable timeline at the bottom. If your code has values that change over time, you can drag the timeline to inspect their value at an arbitrary point in time. These controls are visible in Figure 3.6.



**FIGURE 3.6**

Additional output and controls are available in the assistant editor.

The beauty of the Playground is that you can do anything you want. Go back and try some of the Swift syntax discussed in this hour. Try comparing dates, test a few loops and `switch` statements... whatever you want.

We'll be using the Playground often to explore new code concepts, even making it do a few things that might surprise you. I recommend using it now to gain experience writing Swift.

## Further Exploration

Although you can be successful in learning iOS programming without spending hours and hours learning more Swift, you will find it easier to create complex applications if you become more comfortable with the language. Swift, as mentioned before, is not something that can be described in a single hour. It is a new language that is evolving to meet the specific needs of Apple's computing platform.

To learn more about Swift, check out *Programming in Objective-C 2.0, Third Edition* (Addison-Wesley Professional, 2011), *Objective-C Phrasebook* (Addison-Wesley Professional, 2011), *Sams Teach Yourself Swift in 24 Hours* (Sams, 2014), and *Xcode 4 Unleashed* (Sams, 2012).

Apple has also published a book (*The Swift Programming Language*) that covers the entirety of the Swift language, including Playground exercises. It is available directly with the iBook store on your Mac or iOS device. This isn't just recommended reading; it's a required download for any serious developer-to-be.

## Summary

In this hour, you learned about object-oriented programming and the Swift language. Swift will form the structure of your applications and give you tools to collect and react to user input and other changes. After reading this hour, you should understand how to make classes, declare objects, call methods, and use decision and looping statements to create code that implements more complex logic than a simple top-to-bottom workflow. You should also have an understanding of the iOS Playground and how it can be used to test code without needing to create a full iOS project.

Keep in mind that a typical book would spend multiple chapters on these topics, so our goal has been to give you a starting point that future hours will build on, not to define everything you'll ever need to know about Swift and OOP.

## Q&A

**Q. Is Swift on iOS the same as on OS X?**

**A.** For the most part, yes. OS X includes thousands of additional application programming interfaces (APIs), however, and provides access to the underlying UNIX subsystem.

**Q. Can an if-then-else statement be extended beyond evaluating and acting on a single expression?**

**A.** Yes. The if-then-else statement can be extended by adding another if statement after the else:

```
if <expression> {  
    // do this, the expression is true.  
} else if <expression> {  
    // the expression isn't true, do this instead.  
} else {  
    // Neither of the expressions are true, do this anyway!  
}
```

You can continue expanding the statement with as many else-if statements as you need.

**Q. Why is the Playground better than coding up a new project? Seems the same to me.**

**A.** The biggest advantage of the iOS Playground is that it lets you see instant feedback from the code you enter. In addition, you don't even have to add output statements to inspect the contents of variables you declare; they appear automatically in the Playground margin.

## Workshop

### Quiz

1. ARC stands for what?

- a. Automatic Reference Counting
- b. Aggregated Recall Counts
- c. Automated Reference Cycling
- d. Apple Really Cares

2. The class files that you create for your applications will have which file extension?
  - a. .swift
  - b. .m
  - c. .c
  - d. .swf
3. Variables and methods that may return `nil` are known as what?
  - a. Uncontrolled
  - b. Controlled
  - c. Optional
  - d. Implicit
4. Declaring a variable with a `!` after the type definition makes it what?
  - a. Unwrapped
  - b. Optional
  - c. Explicitly unwrapped
  - d. Implicitly unwrapped
5. Stringing together method calls and variable properties is known as which of the following?
  - a. Spanning
  - b. Bridging
  - c. Chaining
  - d. Declaring
6. A variable that is defined outside of a method and that can be accessed from other classes is called what?
  - a. Constant
  - b. Instance variable
  - c. Implicitly unwrapped variable
  - d. Variable property
7. To declare a constant versus a variable, you replace the `var` keyword with which of the following?
  - a. `set`
  - b. `get`
  - c. `let`
  - d. `constant`

8. Variable types that can store multiple different values are known (in general) as what?
- a. Collections
  - b. Sets
  - c. Structs
  - d. Aggregates
9. At the time iOS 8 was released, Swift had been available to the public for how many years?
- a. 0
  - b. 1
  - c. 2
  - d. 3
10. Swift classes are defined using how many files?
- a. 1
  - b. 2
  - c. 3
  - d. 4

## Answers

- 1. A. ARC stands for Automatic Reference Counting and is the process Apple's development tools use to determine whether an object can be freed from memory.
- 2. A. Class files developed in swift should include the .swift file extension.
- 3. C. A variable or method that returns `nil` (no value) is said to be optional.
- 4. D. Adding an exclamation point (!) after a variable's type definition sets that variable to be implicitly unwrapped.
- 5. C. Swift methods and variable properties can be strung together in a process called chaining.
- 6. D. Variable properties are declared outside of methods and can be accessed and used by other classes.
- 7. C. The `let` keyword is used to declare a constant.
- 8. A. Collections, including `Arrays` and `Dictionaries`, are used to store multiple pieces of data.
- 9. A. Swift and iOS 8 were released to the public at exactly the same time.
- 10. B. A Swift class requires exactly one file for its implementation.

## Activities

1. Start Xcode and create a new project using the iPhone or iPad Single View Application template. Review the contents of the classes in the project folder. With the information you've read in this hour, you should now be able to read and navigate the structure of these files.
2. Use iBooks on your iOS Device or Mac to download Apple's free book *The Swift Programming Language*. This is an entirely free guide to the complete Swift language, straight from the source: Apple!
3. Use the iOS Playground to test your knowledge of Swift syntax and build simple procedural programs.

# Index

## Symbols

! (exclamation mark), 103, 197, 268

!=, 103

&&, 103

||

+/- icons, 553

==, 103

(), 103

@2x, 220

@3x, 220

## A

A chips, 8

About, 554

About.plist file, 555

Accelerate, 125

acceleration, reading, with Core Motion, 645-647

acceleration data, 660

accelerometers, 639-642

accessibility attributes, 165-167

Accessibility Inspector, enabling, 167

Accessibility Programming Guide, 180

accessing

media items, 675-676

motion data, 643

orientation data, 643

System Sound Services, 326-327

tutorials, 880-884

Variable List, debuggers, 852-853

accessors, 85

Accounts, 123

action sheets, 322-324

actions

adding, 239

Gestures project, 624-625

GettingAttention project, 332-333

- ImageHop project, 266-267
- LetsTab project, 476
- Single View Application template, 206-207
- BackgroundColor project, 541-543
- BestFriend project, 732-733
- ColorTilt project, 654-655
- connecting to, 174-176
- CustomPicker project, 431-432
- DateCalc project, 418-419
- FloraPhotographs project, 295-298
  - adding, 296-298
- Gestures project, 623-625
- GettingAttention project, 331-333
- ImageHop project, 264-265
- implementing, 338-341
- Interface Builder (IB), 172-178
- LetsNavigate project, 466-467
- LetsTab project, 475-476
- MediaPlayground project, 687-688
- Modal Editor project, 386-387
- responding to, alert controllers, 321-322
- Scroller project, 310
- setting up, 197-199
- Single View Application template, 203-207
- Survey project, 562-563

**active size classes, setting, 821-823**

**adaptive segues, 358**

- disabling, 394

**Add Missing Constraint menu option, 579**

**adding**

- actions, 239
  - FloraPhotographs project, 296-298
  - GettingAttention project, 332-333
  - to ImageHop project, 266-267
  - LetsTab project, 476
  - Single View Application template, 206-207
- animation resources, ImageHop project, 255

- assets catalogs, Xcode projects, 38
- audio directions, Cupertino project, 796-799
- audio files, 796
- AudioToolbox framework, 795-796
- background modes, 799-800
- blur effect, 549
  - BestFriend project, 744
  - Cupertino project, 768
- constants, 540
  - ReturnMe project, 547
- constraints, Auto Layout, 579-581
- DateChooserViewController, 413
- EditorViewController class, 378
- frameworks, 684
  - BestFriend project, 730
- generic view controller classes, 459-460, 471
- gesture recognizers, 612-613
  - to views, 619-622
- Hop button, 262
- image resources, 547-548
  - CustomPicker project, 427
  - FlowerColorTable project, 499
  - FlowerDetail project, 508
  - Gestures project, 616
- image views, ImageHop project, 256
- images
  - to asset catalogs, 39-56
  - button templates, 220
- media files, 684
- navigation controller classes, 459-460
- navigation controllers, LetsNavigate project, 460
- navigation scenes, with show segues, 451-452
- new scenes and associating the view controller, 379-380
- objects
  - to interfaces, 201-203
  - to scrolling view, 308

- objects to views, 156-157
- outlets, 238
  - FloraPhotographs project, 296
  - GettingAttention project, 331
  - to ImageHop project, 265
  - LetsNavigate project, 466-467
  - LetsTab project, 476
  - Single View Application template, 205-206
- push count variable property, 468
  - LetsTab project, 476-477
- resources, Xcode projects, 37
- scenes, 352-353
  - LetsNavigate project, 461-462
  - LetsTab project, 472
- scenes and associating view controllers, 472
- scrolling behavior, 310-311
- scrolling views, Scroller project, 305-306
- segmented controls, FloraPhotographs project, 289
- segments, 289-290
- simulated devices, iOS Simulator, 69-71
- sliders, ImageHop project, 259
- speed output labels, 262
- steppers, 261
- styled buttons, 234
- switches, FloraPhotographs project, 291
- tab bar controller classes, 471
- tab bar controllers, 471-472
- tab bar item images, 471
- tab bar scenes, 456-457
- table views, 488
  - data source protocols, 491-494
  - delegate protocols, 494-495
  - prototype cell attributes, 489-491
  - setting table attributes, 488-489
- text fields, FieldButtonFun project, 225
  - text views, FieldButtonFun project, 230-231
  - variable property, for image view size, 616-617
  - variables, SystemSoundIDs, 796
  - web views, FloraPhotographs project, 292
- Address Book, 123, 713-714**
  - Address Book framework, 716-717
  - BestFriend project. See BestFriend project
  - people picker navigation controller delegates, 715-716
- Address Book framework, 716-717**
- Address Book people picker, displaying, 734**
- Address Book selection, tying to map displays, 740-741**
- Address Book UI framework, 121, 714-715**
- advanced delegate methods, picker views, 410-412**
- AirPlay, 671**
- alert controllers, 318**
  - action sheets, 322-324
  - alerts, 318-321
  - responding to actions, 321-322
- alert sounds, 327-328, 341-343**
  - playing, 342
- alertBody, 793**
- alerting users, 317**
  - alert controllers, 318
    - action sheets, 322-324
    - alerts, 318-321
    - responding to actions, 321-322
  - GettingAttention project. See GettingAttention project
  - System Sound Services, 325-326
    - accessing, 326-327
    - alert sounds and vibrations, 327-328
    - vibrations, 341-343
- alerts, 318-321, 333-338**
  - multibutton alerts, 334-336
  - using in fields, 337-338

**Align, 159-160****AllInCode project**

- programming interfaces
  - button touches, 606
  - defining variables and methods, 603
  - drawing interfaces when the application launches, 606
  - implementing interface update method, 604-605
  - initializing interface objects, 603-604
  - updating the interface when orientation changes, 606
- setting up, 602

**altitude property, 753****angles, 659****AnimalChooserViewController, 427**

- viewDidLoad, 439

**animated image views, ImageHop project, 267-269****animation, starting/stopping, ImageHop project, 269-270****animation resources, adding, to ImageHop project, 255****animation speed**

- ImageHop project, 270-272
- incrementing, 273-274

**animations, ImageHop project. See ImageHop project****annotations, mapping, 723-724**

- map view delegate protocol, 724-725

**Any, size classes, 833****AnyObject, 95****API Reference, 137****app icons**

- universal applications, 817
- Xcode, 60-62

**AppDelegate.swift, 194****Apple, blurs, 303****Apple Developer Program, 10-11**

- joining paid Developer Program, 12-14
- registering as a developer, 11

**Apple Developer tools, 23**

- Cocoa Touch, 24
- Model-View-Controller (MVC), 24
- Swift, 23

**Apple iOS HIG document, 180****Apple Maps, 721****application data source, FlowerDetail project, 512-515**

- application data structures, 512-515
- populating data structures, 515

**application data structures, FlowerDetail project, 512-515****application designs, MVC (Model-View-Controller). See MVC (Model-View-Controller)****application logic**

- BackgroundColor project, 543
  - reading preferences, 545
  - storing preferences, 544-545
- BackgroundDownload project, 808-809
- BestFriend project
  - conforming to people picker delegate protocol, 734
  - contact information, 734-737
  - displaying Address Book people picker, 734

**ColorTilt project, 656**

- acceleration data, 660
- displaying attitude data, 659-660
- initializing Core Motion Motion Manager, 656-657
- managing motion updates, 657-658
- preventing interface-orientation changes, 661-662
- reacting to rotation, 661

**Cupertino project**

- configuring location manager instance, 763
- implementing location manager delegate, 763-766
- location manager, 762-763
- setting status bar to white, 766

- updating, 771-776
- updating plist files, 766
- FieldButtonFun project, 243-245
- FloraPhotographs project, 298
  - fixing up interface when app loads, 302
  - hiding/showing detail web views, 298-300
  - loading and displaying images and details, 300-302
- FlowerColorTable project, 502
  - populating flower arrays, 502
  - table view data source protocols, 503-505
  - table view delegate protocols, 505-507
- Gestures project, 625-626
  - pinch recognizer, 627-630
  - replacing image views, 626
  - responding to tap gesture recognizers, 627
  - rotation recognizer, 630-632
  - shake gestures, 634-635
  - swipe recognizer, 627
- ImageHop project, 267
  - animated image views, 267-269
  - animation speed, 270-272
  - incrementing animation speed, 273-274
  - starting/stopping animation, 269-270
  - unreadable status bar, 274
- implementing, 208
- LetsNavigate project, 467
  - adding push count variable property, 468
  - incrementing/displaying counters, 468-469
- LetsTab project, 476
  - adding push count variable property, 476-477
  - counter displays, 477
  - incrementing tab bar item badge, 477-478
  - triggering counter updates, 478-479
- Modal Editor project, 388-389
  - hiding keyboards, 389
- Orientation project
  - determining orientation, 650-651
  - registering orientation updates, 649-650
- ReturnMe project, 557-559
- Scroller project, 310
  - adding scrolling behavior, 310-311
- SlowCount project, 802-804
- Survey project
  - hiding keyboards, 564
  - showing survey results, 565-567
  - storing survey results, 564-565
- application object (UIApplication), 128**
- application preferences, 527-529**
  - pseudo preferences, 529-530
- application resource constraints, iOS devices, 8**
- applicationDidEnterBackground, 788**
- applicationDidEnterBackground, 127, 567, 787-788**
- application:didFinishLaunchingWithOptions, 788**
- applicationIconBadgeNumber, 793**
- application:performFetchWithCompletionHandler, 808-809**
- applications, entering background, 127**
- applicationWillEnterForeground, 790-791**
- applicationWillResignActive, 788**
- apps**
  - iOS apps. See iOS apps
  - launching with iOS Simulator, 65-66
  - lifecycle of iOS apps, 126-127
  - quitting, 209
  - running Xcode, 53-54
  - testing, FloraPhotographs project, 303
- ARC (Automatic Reference Counting), 107**
- Arrange, 159-160**
- Arrays, 89**

**arrays, 91-92**

- populating, FlowerColorTable project, 502

**arrows, compass, 776****asset catalogs, 684**

- adding to Xcode projects, 38
- images, adding, 39-56
- retina image assets, 41-42

**assistant editor, Xcode, 48-49****associating view controllers, 461-462**

- LetsTab project, 472

**attitude, 646**

- reading with Core Motion, 645-647

**attitude data, displaying, 659-660****attributed text versus plain text, 227****attributes**

- bar button items, 404
- date pickers, 406-407
- navigation bar item attributes, 450-451
- prototype cell attributes, 489-491
- tab bar item attributes, 455-456
- table attributes, 488-489

**Attributes Inspector, 164-165, 584****audio, adding task-specific background processing, 795-796****audio directions, adding, to Cupertino project, 796-799****audio files, adding, 796****audio formats, 671****audio playback, MediaPlayerGround project, 692-693, 695-696**

- controlling, 696
- loading recorded sound, 696-697

**audio recording, MediaPlayerGround project, 692-693**

- controlling, 694-695
- implementing, 693-694

**AudioToolbox framework, adding, 795-796****authorization**

- requesting for Core Location, 752
- requesting for notifications, backgrounding, 792

**Auto Layout**

- constraint errors, 586-590
- constraints
  - adding, 579-581
  - centering, 590-592
- content compression resistance, 585-586
- content hugging, 585-586
- designing rotatable and resizable interfaces, 576-577
- gesture recognizers, 615
- scrolling views, 311

**Auto Layout Guide, 180****Auto Layout system, 161**

- constraints, 161-162, 164
- Content Compression Resistance, 162-163
- Content Hugging, 162-163

**Automatic Reference Counting (ARC), 107****autosizing features, reverting to old layout approach, 163****AV Audio Player, 677**

- completion, 677

**AV audio recorder, 678-679****AV Foundation framework, 121, 676-677**

- AV Audio Player, 677
- completion, 677
- AV audio recorder, 678-679

**availability, 142****AVAudioPlayer, 676****AVAudioRecorder, 676****AVEncoderAudioQualityKey, 678****AVFormatIDKey, 678****AVNumberOfChannelsKey, 678****AVSampleRateKey, 678**

## B

### Back button, 451

- navigation controllers, 448

### background color, ImageHop project, 262-264

### Background Fetch mode, adding, 809

### background fetches, 786-787, 806

- BackgroundDownload project

- adding Background Fetch mode, 809

- application logic, 808-809

- designing interfaces, 807

- implementation overview, 806

- outlets, 807-808

- setting up, 807

### background graphics, ImageHop project, 262-264

### background image resources, Cupertino project, 759

### background modes, 805

- adding, 799-800

### background processing, 785

### background suspension, 790-791

### background task processing, SlowCount project, 804-805

### background touch, keyboard hiding, 242

### background-aware application life cycle methods, 787-789

### BackgroundColor project

- application logic, 543

- reading preferences, 545

- storing preferences, 544-545

- building apps, 545-546

- designing interfaces, 540-541

- implementation overview, 539-540

- outlets and actions, 541-543

- setting up, 540

### BackgroundDownload project

- adding Background Fetch mode, 809

- application logic, 808-809

- designing interfaces, 807

- implementation overview, 806

- outlets, 807-808

- setting up, 807

### backgrounding, 783-784

- background fetches, 786-787

- background-aware application life cycle methods, 787-789

- disabling, 789-790

- local notifications, 784-785, 792

- creating/scheduling, 793-794

- properties, 793

- requesting authorization for notifications, 792

- long-running background tasks. *See* long-running background tasks

- suspension, 784, 790-791

- task completion for long-running tasks, 785-786

- task-specific background processing. *See* task-specific background processing

### badges, source control projects, 871-872

### bar button items, 403-404

- attributes, 404

- navigation controllers, 448

### BarItem, 475

### batteries, locations, 756

### BestFriend project

- application logic

- conforming to people picker delegate protocol, 734

- contact information, 734-737

- displaying Address Book people picker, 734

- blur effect, adding, 549

- designing interfaces, 731-732

- configuring map view, 732

- email logic

- conforming to the mail compose delegate protocol, 741

- displaying mail compose view, 741-742
- mail completion, 742
- implementation overview, 730
- map logic, 737
  - controlling map display, 738-740
  - customizing pin annotation view, 740
  - requesting permission to use user's location, 737-738
  - tying map display to Address Book selection, 740-741
- outlets and actions, 732-733
- setting status bar to white, 744-745
- setting up, 730-731
- social networking logic, 742-743
  - displaying compose view, 743-744

**Blame mode, 876**

**blueButton.png, 220**

**Bluetooth, 9**

**blur effect, adding, 549**

- to BestFriend project, 744
- to Cupertino project, 768

**blurs, Apple, 303**

**Bool, 89**

**Boolean values, 91**

**Bottom Layout Guide, 151**

**branching, source control, 865, 877-880**

**breakpoint navigators, 853**

**breakpoints**

- removing, 854
- setting, 845-847

**bridged data types, 129-130**

**browsing documentation, Xcode, 138**

**build schemes, choosing, 52-53**

**built-in actions, connections, 176**

**button attributes, editing, 235**

**button templates, slicing, 219**

- adding images, 220

**button touches, AllInCode project, 606**

**buttons, 132, 215-216**

- Back button, 451
- bar button items, 403-404
  - attributes, 404
- custom button images, setting, 235-237
- styled buttons, adding, 234

## C

**calculateDateDifference, 421**

**calculating heading to Cupertino, 773-774**

**cameras, MediaPlayground project, 697-700**

**canBecomeFirstResponder, 632**

**cells**

- configuring to display in table view, 504-505
- custom cells, 490
- tables, 487

**centering constraints, 590-592**

**CFNetwork, 123**

**CGFloat(), 646**

**CGRect, 88**

**chaining, methods, 101**

**changing state, 235**

**check boxes, 282**

**checkout, 864**

**child properties, 553**

**chooseImage method, 698**

**choosing build schemes, 52-53**

**chosen images, showing, 698-699**

**chosenColor, 173**

**CIFilter, 682-683**

**class declaration, 84**

**class fields Swift variable properties declarations, 84-85**

**class files**

- Single View Application template, projects, 194-195
- Swift, 82-83
  - class declaration, 84
  - constant declaration, 85
  - declaring methods, 86-87
  - ending, 87
  - IBOutlet declarations, 85-86
  - import declaration, 83-84

**classes**

- Cocoa Touch, Playground feature, 131-132
- Core Application classes, 128
  - application object (UIApplication), 128
  - onscreen controls (UIControl), 129
  - responders (UIResponder), 128-129
  - view controllers (UIViewController), 129
  - views (UIView), 128
  - windows object (UIWindow), 128
- data type classes, 129
  - bridged data types, 129-130
  - nonbridged data types, 130-131
- interface classes, 132
  - buttons (UIButton), 132
  - labels (UILabel), 132
  - pickers (UIDatePicker/UIPicker), 134
  - popovers UIPopoverPresentation Controller), 134-135
  - sliders (UISlider), 133
  - steppers (UISteppers), 133-134
  - switches (UISwitch), 133
  - text fields (UITextField/UITextView), 134
- OOP (object-oriented programming), 79
- classes interface classes segmented control (UISegmentedControl), 133**

**cleanup**

- image picker, 699-700
- movie player, MediaPlayer project, 691

**CLLocationCoder class, 738****CLHeading, 758****CLLocation, 754****CLLocationDistance, 753****closures, 318**

- methods, 102

**Cocoa, 119****Cocoa Touch, 24, 117-119, 317****classes**

- data type classes, 129-131
- Playground feature, 131-132
- Core Application classes, 128-129
- interface classes, 132-135
- layers, 120
  - Address Book UI framework, 121
  - Event Kit UI, 121
  - Game Kit, 120
  - iAd, 121
  - Map Kit, 120
  - Message UI framework, 121
  - Notification Center framework, 121
  - PhotosUI, 121
  - UIKit, 120

**code**

- debuggers, 848-850
- Interface Builder (IB), 170
  - implementing, 171-172
  - object identity, 178-179
  - opening projects, 170
  - outlets and actions, 172-178
  - writing, 178
- keyboard hiding, 242-243
- low-level code, 446

**Xcode**

- activating tabbed editing, 50
- adding marks, to do's and fix me's, 47-48
- assistant editor, 48-49
- code completion, 44-46
- editing tools, 42
- navigating, 42
- searching with search navigator, 46-47
- snapshots, 50-51
- symbol navigator, 43

**code completion, Xcode, 44-46****color, background color, ImageHop project, 262-264****colorChoice, 172****ColorTilt project**

- application logic, 656
  - acceleration data, 660
  - displaying attitude data, 659-660
  - initializing Core Motion Motion Manager, 656-657
  - managing motion updates, 657-658
  - preventing interface-orientation changes, 661-662
  - reacting to rotation, 661
- designing interfaces, 653
- implementation overview, 652
- outlets and actions, 654-655
- setting up, 652-653

**comments, adding marks, to do's and fix me's, 47-48****commits, source control projects, 873-874****committing changes, source control, 864****Comparison mode, 876****compass, 768**

- arrows, 776
- Cupertino project. See Cupertino project, updating user interfaces, 769-770
- setting up, 768-769

**compiling, 52****component constants, 428****compose view, displaying in BestFriend project, 743-744****configureView, 521****configuring, 289-290**

- cells to display in table view, 504-505
- devices for development, 16-17
- installed size classes, 823
- location manager instance, 763
- map views, 732
- navigation controllers, LetsNavigate project, 460
- popover segue, 362-365
- popovers, 391-392
- projects, as universal, 816-817
- segue style, 370-371

**connecting**

- to actions, 174-176
- to exits, 367-368

**connections**

- AllInCode project, 602
- BestFriend project, 730-731
- built-in actions, 176
- ColorTilt project, 652-653
- creating to outlets, 173-174
- Cupertino project, 759
- CustomPicker project, 427-428
- DateCalc project, 414-415
- editing with Quick Inspector, 177
- FloraPhotographs project, 288
- FlowerColorTable project, 499
- FlowerDetail project, 509-510
- Gestures project, 616
- ImageHop project, 255
- LetsNavigate project, 463
- LetsTab project, 472
- MediaPlayground project, 685

- Modal Editor project, 381
- Orientation project, 647
- planning, 197-199
- SlowCount project, 801
- verifying, Connections Inspector, 625

**Connections Inspector, 174-175, 446**

**connectivity, iOS devices, 9**

**constant declaration, 85**

**constants, 428, 537**

- adding, 540
  - ReturnMe project, 547
- component constants, 428
- declaring, 95
- location constants, 759-760
- OOP (object-oriented programming), 80
- table section constants, 499

**constants radian conversion constants ColorTilt project, 653**

**constraint errors, 586-590**

**constraint objects, top/bottom layout guides, 581**

**constraint tools, 589**

**constraints**

- adding with Auto Layout, 579-581
- Auto Layout system, 161-162, 164
- centering, 590-592
- editing via Size Inspector, 582-585
- horizontal constraints, 581
- iOS devices, 8
- matching sizes, 598-600
- Modal Editor project, 382
- setting, 595-597
- storyboards, size classes, 830
- vertical constraints, 581
- viewing via Size Inspector, 582-585
- Xcode, 579

**constraints objects, navigating, 581-590**

**contact information, BestFriend project, 734-737**

**contacts applications, 447**

**Content Compression Resistance, 162-163, 585-586**

**Content Hugging, 162-163, 585-586**

**controlHardware method, 657**

**controllers**

- navigation controllers. See navigation controllers
- tab bar controllers. See tab bar controllers
- view controllers, multiscene development, 446

**controlling**

- audio playback, MediaPlayer project, 696
- audio recording, MediaPlayer project, 694-695

**controls, expanding, 592-597**

**convenience initialization method, 268, 285**

**convenience methods, 93-94**

**copy and paste, 229**

**Core Application classes, 128**

- application object (UIApplication), 128
- onscreen controls (UIControl), 129
- responders (UIResponder), 128-129
- view controllers (UINavigationController), 129
- views (UIView), 128
- windows object (UIWindow), 128

**Core Audio, 121**

**Core Bluetooth, 125**

**Core Data, 32, 123, 190, 568**

**Core Foundation, 123, 536**

**Core Graphics, 122**

**Core Image, 121, 682**

- filters, 682-683, 700-702

**Core Location, 123, 751**

- Cupertino project
  - application logic, 762-767
  - designing views, 760-761

- implementation overview, 759
- outlets, 762
- setting up, 759-760
- getting headings, 757-758
- getting locations, 751-752
  - location accuracy and update filter, 756
  - location errors, 754-756
  - location manager delegate protocol, 752-754
  - requesting authorization and plist files, 752

**Core Motion, 123, 643, 645-647**

- Motion Manager, 656-657
- radians, 653
- reading acceleration, 645-647

**Core OS layer, 125**

- Accelerate, 125
- Core Bluetooth, 125
- External Accessory, 125
- Local Authentication, 125
- Security framework, 125
- System framework, 125

**Core Services layer, 123**

- Accounts, 123
- Address Book, 123
- CFNetwork, 123
- Core Data, 123
- Core Foundation, 123
- Core Location, 123
- Core Motion, 123
- Event Kit, 124
- Foundation, 124
- HealthKit, 124
- HomeKit, 124
- Newsstand, 124
- Pass Kit, 124
- Quick Look, 124
- Social, 124
- Store Kit, 125
- System Configuration, 125

**Core Text, 122**

correcting errors with issue navigator, 54-57

Count Down Timer, date picker attributes, 406

counter displays, LetsTab project, 477

counter updates, triggering, 478-479

**counters**

- incrementing/displaying, 468-469
- initializing, 803
- updating, 803-804

**Counting Navigation Controller, 462**

countLabel, 463

CPU usage, monitoring, 855-856

createStory method, 244

**Cupertino project****adding**

- audio, 795
- audio directions, 796-799
- background modes, 799
- blur effect, 768

**application logic**

- configuring location manager instance, 763
- implementing location manager delegate, 763-766
- location manager, 762-763
- setting status bar to white, 766
- updating, 771-776
- updating plist files, 766

**compass**

- implementation overview, 768
- outlets, 771
- setting up, 768-769
- updating user interfaces, 769-770

designing views, 760-761

implementation overview, 759

outlets, 762

setting up, 759-760

curly braces, 318

**custom button images, setting, 235-237**

**custom cells, 490**

**custom picker views, 405, 407-408, 432-438**

changing component and row sizes, 436

CustomPicker project, scene segue logic, 438-439

data source protocols, 433-434

delegate protocols, 434-435

implicit selection, 438

loading picker data, 432-433

reacting to selections, 436-438

**custom pickers, 425-436**

CustomPicker project

creating segues, 431

custom picker views, 432-438

implementation overview, 426

outlets and actions, 431-432

setting up, 426-428

implementation overview, 426

**customizing**

interfaces, 164

accessibility attributes, 165-167

Attributes Inspector, 164-165

keyboard displays with text input traits, 228-229

pin annotation view, 740

**CustomPicker project, 427-428**

connections, 427-428

creating segues, 431

custom picker views, 432-438

changing component and row sizes, 436

data source protocols, 433-434

delegate protocols, 434-435

implicit selection, 438

loading picker data, 432-433

reacting to selections, 436-438

designing interfaces, 428-430

implementation overview, 426

outlets and actions, 431-432

scene segue logic, 438-439

setting up, 426-428

## D

**data**

acceleration data, 660

attitude data, displaying, 659-660

sharing between tab bar scenes, 457-458

sharing between navigation scenes, 452

**data detectors, 232-233**

**data models, MVC (Model-View-Controller), 190**

**data source protocols**

custom picker views, 433-434

FlowerColorTable project, 501

picker view data source protocol, 408-409

table view data source protocols,  
FlowerColorTable project, 503-505

table views, 491-494

**data storage, 530**

direct file system access, 534-535

file paths, 536-537

reading/writing data, 537-538

storage locations for application data,  
535-536

settings bundles, 532-534

user defaults, 530-531

reading/writing, 531-532

**data structures, populating, 515**

**data type classes, 129**

bridged data types, 129-130

nonbridged data types, 130-131

URLs (NSURL), 131

**data types**

- declaring, 89
- object data types, 93

**Date, date picker attributes, 406****Date and Time, date picker attributes, 406****date calculation logic, 419**

- determining the differences between dates, 420
- displaying date and time, 419-420
- getting dates, 419
- implementing date calculation and display, 421-422
- updating date output, 422-423

**date formats, 420****date output, updating, 422-423****date pickers, 405-406**

- attributes, 406-407
- DateCalc project, 412
  - creating segues, 417
- date calculation logic. *See* date calculation logic
- designing interfaces, 415-417
- implementation overview, 413
- implementing scene segue logic, 424-425
- outlets and actions, 418-419
- setting up, 413-415

**DateCalc project, 412**

- building apps, 425
- connections, 414-415
- creating segues, 417
- date calculation logic, 419
  - determining the differences between dates, 420
  - displaying date and time, 419-420
  - getting dates, 419
  - implementing date calculation and display, 421-422
  - updating date output, 422-423

- designing interfaces, 415-417
- implementation overview, 413
- implementing scene segue logic, 424-425
- outlets and actions, 418-419
- setting up, 413-415
- variables, 414-415

**DateChooserViewController, 414**

- adding, 413

**dates, determining the differences between dates, 420****dates (NSDate), 130****Debug, 842****debug, Xcode, 33****debug navigators, 853-855****DebuggerPractice project, 843-845****debuggers, 841-842**

- accessing Variable List, 852-853
- breakpoint navigators, 853
- breakpoints, setting, 845-847
- code, 848-850
- Debug, 842
- debug navigators, 853-855
- DebuggerPractice project, 843-845
- lldb, 845
- monitoring CPU and memory usage, 855-856
- Release, 842
- variable states, 847-848
- view hierarchy, checking, 856-858
- watchpoints, 851-852

**decision making, 102****declaration, 142****declarations**

- class declaration, 84
- constant declaration, 85
- IBOutlet declarations, 85-86
- import declaration, 83-84
- variable properties declarations, 84-85

**declared in, 142****declaring**

- constants, 95
- methods, 86-87
- variables, 89
  - arrays, 91-92
  - Boolean values, 91
  - convenience methods, 93-94
  - data types, 89
  - dictionaries, 92-93
  - integers and floating-point numbers, 89-90
  - object data types, 93
  - optional values, 95-97
  - strings, 90-91

**default images, ImageHop project, 257****default selections, 438****default simulated devices, 154****default states, FloraPhotographs project, 292****default transitions, 361****delegate protocols**

- custom picker views, 434-435
- FlowerColorTable project, 501, 505-507
- location manager delegate protocol, 752-754, 763-766
- mail compose delegate protocol, 741
- map view delegate protocol, 724-725
- people picker delegate protocol, 734
- picker views, 409-410
- table views, 494-495

**describeInteger, 843, 854****description, 142****designing interfaces**

- adding objects, 201-203
- BackgroundColor project, 540-541
- BackgroundDownload project, 807
- BestFriend project, 731-732
  - configuring map view, 732

## ColorTilt project, 653

## CustomPicker project, 428-430

## DateCalc project, 415-417

## FieldButtonFun project, 224

- adding styled buttons, 234
- adding text fields, 225
- adding text views, 230-231
- customizing keyboard displays with text input traits, 228-229
- editing button attributes, 235
- editing text field attributes, 225-227
- editing text view attributes, 231-232
- scrolling options, 233-234
- setting custom button images, 235-237
- setting simulated interface attributes, 224

## FloraPhotographs project, 288, 295

- adding segmented controls, 289
- adding switches, 291
- adding web views, 292
- segments, adding/configuring, 289-290
- setting default state, 292
- setting web view attributes, 293-294
- sizing controls, 291

## FlowerColorTable project, 500-501

## Gestures project, 617-618

## GettingAttention project, 329-330

## ImageHop project, 256

- adding Hop button, 262
- adding image views, 256
- adding sliders, 259
- adding speed output labels, 262
- adding steppers, 261
- background graphics and color, 262-264
- making copies, 258
- setting default images, 257
- setting slider range attributes, 259-261
- setting stepper range attributes, 261-262

LetsNavigate project, 465-466

LetsTab project, 473-474

MediaPlayground project, 685-686

Modal Editor project, 381-384

Orientation project, 648

ReturnMe project, 548

Scroller project, 305

    adding objects, 308

    adding scrolling views, 305-306

    resetting View Controller Simulated Size, 309

    setting freeform size, 306-307

SlowCount project, 802

setting simulated interface attributes, 199-201

Survey project, 560-561

**designing views, Cupertino project, 760-761**

**desiredAccuracy, 756**

**detail scenes, updating, 511-512**

**detail view controllers, 497**

    FlowerDetail project, 519

    displaying detail view, 520-521

**detail views, displaying, 520-521**

**detail web views, hiding/showing, FloraPhotographs project, 298-300**

**detailItem, 519, 521**

**details, loading/displaying, FloraPhotographs project, 300-302**

**developers, 9-10**

    Apple Developer Program, 10-11

        joining paid Developer Program, 12-14

        registering, 11

    installing Xcode, 14-16

    who can become iOS developer, 9-10

**development provisioning profiles, iOS apps, 16**

**device models**

    universal applications, 819

**device orientations, Xcode, 59**

**devices**

    configuring for development, 16-17

    default simulated devices, 154

    iOS devices, 5-6

**Dictionaries, 89**

**dictionaries, 92-93**

**didRotateFromInterfaceOrientation, 577**

**different screen sizes, accommodating, 7**

**direct file system access, 530, 534-535**

    file paths, 536-537

    reading/writing data, 537-538

    storage locations for application data, 535-536

**direction image resources, Cupertino project, 769**

**directionArrow, 775**

**disabling**

    adaptive segues, 394

    backgrounding, 789-790

    editing, Master-Detail Application template, 518

**dismissDateChooser, 418**

**dismissing modal scenes, programmatically, 366**

**displaying**

    Address Book people picker, 734

    attitude data, 659-660

    compose view, BestFriend project, 743-744

    counters, 468-469

    date and time, 419-420

    detail views, 520-521

    images and details, FloraPhotographs project, 300-302

    mail compose view, 741-742

    media picker, 704-705

- displays, 6-7**
  - updating, 803-804
- distanceFromLocation method, 764**
- distanceView outlet, 762**
- distantPast(), 95**
- doAcceleration, 660**
- doActionSheet method, 339-340**
- doAlert method, 333, 793-794**
- doAlertInput, 337**
- doAttitude, 659-660**
- document outline, storyboards, 149-152**
- document outline objects, storyboards, 153-154**
- documentation, Xcode, 135-136**
  - browsing, 138
  - navigating, 139-140
  - searching, 137-138
  - setting up documentation downloads, 136-137
- Documents directory, 536**
- doMultipleButtonAlert, 335**
- Done button, keyboard hiding, 240-242**
- doSound method, 341-342**
- Double, 89**
- doVibration method, 343**
- do-while loops, 106-107**
- downcasting, 97**
- downloading changes, source control, 865**
- downloads, documentation, setting up, 136-137**

## E

- editing**
  - button attributes, 235
  - code, Xcode, 42
  - connections, with Quick Inspector, 177
  - constraints, via Size Inspector, 582-585

- disabling, Master-Detail Application template, 518
  - tabbed editing, Xcode, 50
  - text field attributes, 225-227
  - text view attributes, 231-232
- editing tools, Interface Builder (IB), 157**
  - guides, 157-158
  - selection handles, 158-159
  - Size Inspector, 159-161
- editor, Xcode, 33**
- Editor menus, 589**
- EditorViewController, 376**
  - adding, 378
- email, 717-719**
  - BestFriend project. See BestFriend project
  - mail compose view controller delegate, 719
- email logic, BestFriend project**
  - conforming to the mail compose delegate protocol, 741
  - displaying mail compose view, 741-742
  - mail completion, 742
- empty selections, 706**
- endBackgroundTask, 801**
- ending class files, Swift, 87**
- errors**
  - constraint errors, 586-590
  - correcting with issue navigator, 54-57
  - location errors, 754-756
  - placement errors, 588
- Event Kit, 124**
- Event Kit UI, 121**
- exclamation mark (!), 197, 268**
- Exit icon, 151**
- exits, 351, 389**
  - connecting to, 367-368
  - multiscene projects, 366-368
  - view controllers, 381

**expanding controls, 592-597**

**expressions, 102-103**

if-then-else, 103-104

loops

do-while loops, 106-107

for loops, 105-106

while loops, 106-107

switch statements, 103-104

syntax, 103

**extensions, OOP (object-oriented programming), 80**

**External Accessory, 125**

## F

**feedback, iOS devices, 9**

**FieldButtonFun project, 217**

application logic, 243-245

building apps, 245

designing interfaces, 224

adding styled buttons, 234

adding text fields, 225

adding text views, 230-231

customizing keyboard displays with text input traits, 228-229

editing button attributes, 235

editing text field attributes, 225-227

editing text view attributes, 231-232

scrolling options, 233-234

setting custom button images, 235-237

setting simulated interface attributes, 224

keyboard hiding, 240-243

outlets and actions, 237-239

preparing button templates with slicing, 219-224

setting up, 218-219

**fields, alert, 337-338**

**file formats, web views, 284**

**file paths, direct file system access, 536-537**

**file storage**

implementation overview, 559-560

Playground feature, 538-539

Survey project

application logic, 564-567

designing interfaces, 560-561

outlets and actions, 562-563

setting up, 560

**fileExistsAtPath, 537**

**files, removing, from Xcode projects, 37-38**

**filtering, 35**

**filters**

Core Image, 682-683, 700-702

media picker, 673

**fireDate, 793**

**First Responder icon, 151**

**first responders, 632-634**

**FIXME, 47-48**

**Flash Professional, 10**

**Float, 89**

**floating-point numbers, 89-90**

**FloraPhotographs project, 287**

application logic, 298

fixing up interface when app loads, 302

hiding/showing detail web views, 298-300

loading and displaying images and details, 300-302

designing interfaces, 288, 295

adding segmented controls, 289

adding switches, 291

adding web views, 292

segments, adding/configuring, 289-290

setting default state, 292

setting web view attributes, 293-294

sizing controls, 291

outlets and actions, 295-298

setting up, 288

testing apps, 303

### **FlowerColorTable project**

application logic, 502

populating flower arrays, 502

table view data source protocols, 503-505

table view delegate protocols, 505-507

connections, 499

data source protocols, 501

delegate protocols, 501

designing interfaces, 500-501

implementation overview, 499

setting up, 499

variables, 499

### **FlowerDetail project**

application data source, 512-515

application data structures, 512-515

populating data structures, 515

connections, 509-510

detail view controllers, 519

displaying detail view, 520-521

master view controllers, 515

creating table cells, 516-517

creating table view data methods, 515-516

disabling editing, 518

handling navigation events from a segue,  
518-519

setting up, 508-510

tweaking interfaces, 510-512

updating detail scenes, 511-512

updating master scenes, 510

web view outlets, 512

variables, 509-510

### **flowerView, 173**

**fonts, size classes, 826**

**for loops, 105-106**

**Foundation, 124**

**foundPinch method, 628**

**foundRotation method, 630-631**

**foundSwipe method, 627**

**foundTap method, 627**

**frames, adding, in BestFriend project, 730**

**frameworks, 117, 119**

Accelerate, 125

Accounts, 123

adding, 684

Address Book, 123

Address Book UI framework, 121

AudioToolbox, adding, 795-796

AV Foundation framework, 121

CFNetwork, 123

Core Audio, 121

Core Bluetooth, 125

Core Data, 123

Core Foundation, 123

Core Graphics, 122

Core Image, 121

Core Location, 123

Core Motion, 123

Core Text, 122

Event Kit, 124

Event Kit UI, 121

External Accessory, 125

Foundation, 124

Game Kit, 120

HealthKit, 124

HomeKit, 124

iAd, 121

Image I/O, 122

Local Authentication, 125

Map Kit, 120

Media Player framework, 122

Message UI framework, 121

Metal, 122

- Newsstand, 124
- Notification Center framework, 121
- OpenGL ES, 122
- Pass Kit, 124
- Photos framework, 122
- PhotosUI, 121
- Quartz Core, 122
- Quick Look, 124
- Security framework, 125
- Social, 124
- Store Kit, 125
- System Configuration, 125
- System framework, 125
- UIKit, 120

**freeform size, setting up in Scroller project, 306-307**

**fullscreen view, transitioning to in media player, 671**

**functions versus methods, 190**

## G

**Game Kit, 120**

**generating**

- multitouch events, 66-67
- output, from Playground, 110-112

**generic view controller classes, adding, 459-460, 471**

**geocoding, 725-728**

- Playground feature, 727-728

**geographic north, 757**

**gesture recognizers, 614**

- adding, 612-613
- to views, 619-622

- Auto Layout, 615

- projects, Gestures project. *See Gestures project*

**gesture-recognizer classes, 612**

**gestures, multitouch gesture recognition, 611-612**

**Gestures project**

- adding gesture recognizers to views, 619-622
- application logic, 625-626
  - pinch recognizer, 627-630
  - replacing image views, 626
  - responding to tap gesture recognizers, 627
  - rotation recognizer, 630-632
  - shake gestures, 634-635
  - swipe recognizer, 627
- building apps, 635
- designing interfaces, 617-618
- implementation overview, 614-615
- outlets and actions, 623-625
- setting up, 616-617
- shake recognizer, 632

**getFlower, 172**

**getters, 85**

**GettingAttention project**

- action sheets, 338-341
- alert sounds and vibrations, 341-343
- alerts, 333-338
  - creating multibutton alerts, 334-336
- fields, 337-338
- designing interfaces, 329-330
- outlets and actions, 331-333
- setting up, 328-329

**Git, 863**

- branching/merging, 865
- committing changes, 864
- downloading changes, 865
- repositories, 865-866
  - connecting to remote repositories, 868-869
  - creating local, 866-868
- working copies, 870-871

**Google Maps, 721**

**GPS, 751**

**graphics, 6-7**

grouped tables, 486

guides, 142

Interface Builder (IB), 157-158

gutters, 844

gyroscope, 639, 642

## H

heading updates, Cupertino project, 771-772, 774-776

headingAvailable, 757, 771

headingFilter, 775

headings

calculating, Cupertino project, 773-774

Core Location, 757-758

HealthKit, 124

hideKeyboard method, 242, 564

hiding

detail web views, FloraPhotographs project, 298-300

keyboards, 564

Modal Editor project, 389

Hint attributes, 166

hi-res images, loading for retina display, 258

HomeKit, 124

Hop button, adding, 262

horizontal constraints, 581

hueSlider, 540

## I

iAd, 121

IB editor, 589

@IBAction, 189-190

@IBOutlet, 188-189, 197

IBOutlet declarations, 85-86

icons, app icons

universal applications, 817

Xcode, 60-62

IDE (integrated development environment), 29

identifier attribute, 534

identifiers, 359

if-then-else, 103-104

Image I/O, 122

image picker, 679-680

cleanup, 699-700

MediaPlayground project, photo library and camera, 697-698

image resources

adding, 547-548

CustomPicker project, 427

FlowerColorTable project, 499

FlowerDetail project, 508

Gestures project, 616

Cupertino project, 759

image views, 253

adding to ImageHop project, 256

animated image views, ImageHop project, 267-269

replacing, Gestures project, 626

ImageHop project, 253-254

application logic, 267

animated image views, 267-269

animation speed, 270-272

incrementing animation speed, 273-274

starting/stopping animation, 269-270

unreadable status bar, 274

building apps, 274-275

designing interfaces, 256

adding Hop button, 262

adding image views, 256

adding sliders, 259

- adding speed output labels, 262
- adding steppers, 261
- background graphics and color, 262-264
- making copies, 258
- setting default images, 257
- setting slider range attributes, 259-261
- setting stepper range attributes, 261-262
- outlets and actions, 264-265
  - adding outlets, 265
- setting up, 254-255
  - adding animation resources, 255

## images

- adding
  - to asset catalogs, 39-56
  - to button templates, 220
- chosen images, 698-699
- default images, ImageHop project, 257
- direction image resources, Cupertino project, 769
- hi-res images, loading for retina display, 258
- JPEG images, 40
- loading/displaying, FloraPhotographs project, 300-302
- PNG images, 40
- retina image assets, 41-42
- size classes, 826-827
- tab bar item images, adding, 471
- UI image picker controller delegate, 680-682

## imperative programming, 78

### implementing

- application logic, Single View Application template, 208
- audio recording, MediaPlayground project, 693-694
- code, Interface Builder (IB), 171-172
- interface update method, 604-605
- Single View Application template, 191-192
- split view controllers, 496-497

### implicit preferences, 539

- BackgroundColor project
  - application logic, 543-545
  - designing interfaces, 540-541
  - implementation overview, 539-540
  - outlets and actions, 541-543
  - setting up, 540

### implicit selection, custom picker views, 438

### implicit unwrapping, 97-98

### import declaration, class files, 83-84

### incrementCount, 468

### incrementCountFirst, 475

### incrementCountSecond, 475

### incrementCountThird, 475

### incrementing

- animation speed, 273-274
- counters, 468-469
- tab bar item badge, 477-478

### indexed tables, 486

### initializing, interface objects, 603-604

### initializing Core Motion Motion Manager, 656-657

### initWithContentURL, 689

### input. *See also* output

- FieldButtonFun project, 217
  - application logic, 243-245
  - building the app, 245
  - designing interfaces. *See* designing interfaces
  - keyboard hiding, 240-243
  - preparing button templates with slicing, 219-224
  - setting up, 218-219
- iOS devices, 9

### inspecting output from Playground, 110-112

### installed size classes, configuring, 823

### installing, Xcode, 14-16

### instance methods, OOP (object-oriented programming), 80

**instances, OOP (object-oriented programming), 80**

**instantiation, 149**

OOP (object-oriented programming), 80

**Int, 89**

**integers, 89-90**

**interactions**

buttons, 215-216

labels, 216

text fields, 216

text views, 216

**Interface Builder (IB), 147-148**

Auto Layout system, 161

constraints, 161-162, 164

Content Compression Resistance, 162-163

Content Hugging, 162-163

connecting to code, 170

implementing, 171-172

object identity, 178-179

opening projects, 170

outlets and actions, 172-178

writing code, 178

customizing interfaces, 164

accessibility attributes, 165-167

Attributes Inspector, 164-165

editing tools, 157

guides, 157-158

selection handles, 158-159

Size Inspector, 159-161

overview, 148

previewing interfaces, 168-169

resources, 179

storyboards, 149

document outline, 149-152

document outline objects, 153-154

user interfaces

adding object to views, 156-157

Object Library, 154-155

**interface classes, 132**

buttons (UIButton), 132

labels (UILabel), 132

pickers (UIDatePicker/UIPicker), 134

popovers

UIPopoverPresentation

Controller), 134-135

sliders (UISlider), 133

steppers, 133-134

switches (UISwitch), 133

text fields (UITextField/UITextView), 134

**interface classes segmented control**

(UISegmentedControl), 133

**interface objects, initializing, 603-604**

**interface rotation events, 644**

**interface update method, implementing, 604-605**

**interface-orientation changes, preventing, 661-662**

**interfaces**

customizing, 164

accessibility attributes, 165-167

Attributes Inspector, 164-165

designing, 199-203

adding objects, 201-203

setting simulated interface attributes,  
199-201

expanding controls, 593-595

iOS 7, 7

previewing, 168-169

programmatically defined interfaces. *See*  
programmatically defined interfaces

responsive interfaces

designing rotatable and resizable interfaces,  
576-578

rotation, 573-574

rotation, enabling, 574-575

tweaking. *See* tweaking interfaces

interpreting results, of Quick Help, 142-143

Intrinsic Size setting, Auto Layout system, 163

iOS 6, segmented controls, 290

iOS 7, interfaces, 7

iOS Accessibility Inspector, enabling, 167

iOS applications, data storage, 527-529

iOS apps

configuring devices for development, 16-17

development provisioning profiles, 16

launching, 19-22

lifecycle of iOS apps, 126-127

running, 16-19

iOS Dev Center, 14

iOS developers, 9-10

Apple Developer Program, 10-11

joining paid Developer Program, 12-14

registering as a developer, 11

installing Xcode, 14-16

iOS devices, 5-6

application resource constraints, 8

connectivity, 9

display and graphics, 6-7

feedback, 9

input, 9

registering multiple devices, 17

iOS Human Interface Guidelines, 180

iOS Simulator, 63-64

adding additional simulated devices, 69-71

data storage, 535

launching apps, 65-66

multitouch events, generating, 66-67

rotating, simulated devices, 67

running, first time, 57

testing other conditions, 68-69

iPads, screens, 6

iPhone, 6

iPhone 5, screens, 7

iPhone 6, screens, 6

iPhone 6+, screens, 6

iPhones

popovers, 393-395

screens, 6

iPod touch, 6

isAnimating, 269

issue navigator, correcting, errors, 54-57

items, navigation controllers, 448

## J

JPEG images, 40

## K

key constants, 540

ReturnMe project, 547

keyboard displays, customizing with text input traits, 228-229

keyboard hiding

adding code, 242-243

FieldButtonFun project, 240-243

background touch, 242

Done button, 240-242

Modal Editor project, 389

keyboard types, 229

keyboards

hiding, 564

virtual keyboards, 217

keychains, 17

## L

**Label attributes, 166**

**labels, 216**

getting in scrolling view, 309

UILabel, 132

**lastSound, 797**

**launch images, Xcode, 60, 62-63**

**launch screens**

universal applications, 818

Xcode, 60, 62-63

**launching**

apps, iOS Simulator, 65-66

iOS apps, 19-22

**layers, 119**

Cocoa Touch, 120

Address Book UI framework, 121

Event Kit UI, 121

Game Kit, 120

iAd, 121

Map Kit, 120

Message UI framework, 121

Notification Center framework, 121

PhotosUI, 121

UIKit, 120

Core OS layer, 125

Accelerate, 125

Core Bluetooth, 125

External Accessory, 125

Local Authentication, 125

Security framework, 125

System framework, 125

Core Services layer, 123

Accounts, 123

Address Book, 123

CFNetwork, 123

Core Data, 123

Core Foundation, 123

Core Location, 123

Core Motion, 123

Event Kit, 124

Foundation, 124

HealthKit, 124

HomeKit, 124

Newsstand, 124

Pass Kit, 124

Quick Look, 124

Social, 124

Store Kit, 125

System Configuration, 125

Media layer, 121

AV Foundation framework, 121

Core Audio, 121

Core Graphics, 122

Core Image, 121

Core Text, 122

Image I/O, 122

Media Player framework, 122

Metal, 122

OpenGL ES, 122

Photos framework, 122

Quartz Core, 122

**layout guides, 581**

**leading, 579**

**leading space, 581**

**LetsNavigate project**

application logic, 467

adding push count variable property, 468

incrementing/displaying counters, 468-469

building apps, 469

designing interfaces, 465-466

implementation overview, 458-459

outlets and actions, 466-467

setting up, 459

adding scenes and associating view controllers, 461-462

adding/configuring navigation controllers, 460

connections, 463

variables, 463

show segues, creating, 464-465

**LetsNavigate project setting up adding navigation controllers and generic view controller classes, 459-460**

**LetsTab project**

adding scenes and associating view controllers, 472

application logic, 476

adding push count variable property, 476-477

counter displays, 477

incrementing tab bar item badge, 477-478

triggering counter updates, 478-479

building apps, 479

designing interfaces, 473-474

implementation overview, 470

outlets and actions, 475-476

setting up, 470

adding tab bar controller and generic view controller classes, 471

adding tab bar controllers, 471-472

connections, 472

variables, 472

tab bar relationships, creating, 472-473

**LetsTab project setting up adding tab bar item images, 471**

**Library/Caches directory, 536**

**lifecycle of iOS apps, 126-127**

**lifecycles, background-aware application life cycle methods, 787-789**

**limitations, MVC (Model-View-Controller), 186**

**linking, 52**

## listings

Activating Interface Rotation, 575

Add a Method as a Placeholder for the Unwind Segue, 381

Adding a Method in GenericViewController.swift to Update Each Scene's Counter, 478

Adding Audio Feedback When the Heading Updates, 798-799

Adding the foundRotation Method, 630-631

Adding the getFlower Implementation, 300-301

Applying a Filter to the Image in the UIImageView, 700-701

Asking to Become a First Responder, 634

Calculating a Heading to a Destination, 774

Calculating the Date Difference, 422

Calculating the Difference Between Two Dates, 421

Calculating the Distance When the Location Updates, 764-765

Calling the NSLog Function, 839

Centering the Map and Adding an Annotation, 739

Changing the Label as the Orientation Changes, 650

Cleaning Up After the Movie Player, 691

Completed setOutput Method, 208

The Completed setSpeed Method, 270-271

Completing the recordAudio Method, 696

Configuring a Cell to Display in Table View, 504-505

Configuring and Displaying the Mail Compose View, 741

Configuring the Detail View Using the detailItem, 521

Configuring the Sections and Row Count for the Table View, 492

Creating a Method to Display the User's Selection, 437

Creating and Initializing the Audio Recorder, 693

- Creating the Location Manager Instance, 763
- Customizing the Annotation View, 724, 740
- Defining Handlers Within Alert Actions, 322
- Defining the Minimum Background Fetch Interval, 808
- Disable the Adaptive Segue, 394
- Disabling Editing of Table Cells, 518
- Disabling Editing of the UI, 518
- Disabling Interface Rotation, 662
- Dismissing the Mail Compose View, 742
- Dismissing the Modal Scene, 424, 439
- Displaying the Media Picker, 704
- The doAlertInput Implementation, 337
- Editing the viewDidLoad Method, 842-844
- Enabling Scrolling in Your Scroll View, 311
- Enabling the Ability to Be a First Responder, 632
- Example of the Tap Gesture Recognizer, 613
- The Final viewDidLoad Implementation, 703-704
- Finishing the Background Fetch by Implementing application:performFetchWithCompletionHandler, 808-809
- Forward Geocoding, 725
- Handling a Cancel Action in the People Picker, 715
- Handling a Popover Dismissal, 374
- Handling a Row Selection Event, 506
- Handling a User's Music Selection, 705
- Handling Button Touches, 606
- Handling Drilling Down to Individual Properties, 716
- Handling Empty Selections in the Media Picker, 706
- Handling Heading Updates, 758
- Handling Location Manager Errors, 763-764
- Handling Playback Completion, 677
- Handling Rotation in didRotateFromInterfaceOrientation, 606
- Handling the Cancellation of a Media Selection, 674
- Handling the Cancellation of an Image Selection, 681, 699
- Handling the Composition Completion, 719
- Handling the Heading Updates, 775-776
- Handling the Notification of Playback Completion, 672
- Handling the Selection of a Contact, 735-736
- Handling the Selection of a Person in the Address Book, 716
- Handling the Selection of an Image, 681
- Handling the Selection of Media Items, 674
- Handling the User's Selection of an Image, 699
- Hiding the Keyboard, 243
- Hiding the Keyboard When Its Done Key Is Pressed, 389
- Hiding the Keyboard When It Isn't Needed, 564
- Implementing a Custom Picker Data Source Protocol, 408
- Implementing a Custom Picker Delegate Protocol, 409-410
- Implementing a UISheet Class, 323-324
- Implementing an Alert-styled UIAlertController, 319-320
- Implementing playAudio Method, 696
- Implementing the chooseImage Method, 698
- Implementing the controlHardware Method, 657-658
- Implementing the createStory Method, 244
- Implementing the describeInteger Method, 843
- Implementing the doAcceleration Method, 660
- Implementing the doActionSheet Method, 339-340
- Implementing the doAlert Method, 333
- Implementing the doAttitude Method, 659
- Implementing the doMultipleButtonAlert Method, 335
- Implementing the doRotation Method, 661

- Implementing the doVibration Method, 343
- Implementing the Final setBackgroundHueValue Method, 544
- Implementing the foundPinch Method, 628
- Implementing the foundSwipe Method, 627
- Implementing the foundTap Method, 627
- Implementing the incrementCount Method, 468
- Implementing the Initial setBackgroundHueValue method, 543
- Implementing the newBFF method, 734
- Implementing the playMusic Method, 706
- Implementing the setValuesFromPreferences Method, 558
- Implementing the showResults Method, 566
- Implementing the Simple Tweet Compose View, 743-744
- Implementing the storeSurvey Method, 564-565
- Implementing the toggleFlowerDetail Method, 300
- Implementing the viewDidLoad Method, 626
- Implementing updateInterface, 604-605
- Initializing the Interface When the Application Loads, 606
- Initializing the Motion Manager, 657
- Initializing the Movie Player, 689-690
- Initializing the Sound File References in viewDidLoad, 796-797
- Initiating Movie Playback, 690-691
- Loading and Playing a Sound, 327
- Loading the Animation, 267-268
- Loading the Data Required for the Picker View, 432-433
- Loading the Settings When the Initial View loads, 559
- Performing a Default Calculation When the Date Chooser Is First Displayed, 423
- Placing an Annotation, 723
- Populating the Field with the Current Email Address, 388
- Populating the Flower Data Structures, 513
- Prepare the Interface (But Don't Display It Yet), 603
- Preparing and Showing the Compose Dialog, 719
- Preparing the Audio Player with a Default Sound, 695-696
- Preparing to Post to Facebook, 721
- Presenting the Picker with Custom Views, 411
- Processing a CImage with a CFilter, 682
- Providing a Custom View for Each Possible Picker Element, 434-435
- Reacting to a User's Selection, 437
- Reacting to a User's Touch, 494
- Reacting to Core Location Errors, 755
- Requesting Heading Updates, 771-772
- Requesting Notification Authorization, 792
- Responding to a Shake Gesture, 634-635
- Returning a Count of the Rows (Array Elements) in Each Section, 503
- Returning a Heading for Each Section, 504
- Returning the Number of Components, 433
- Returning the Number of Elements per Component, 434
- Returning the Number of Sections in the Table, 503
- Reverse Geocoding, 726
- A Sample Interface File, 83
- Scheduling a Timer When the Application Starts, 803
- Set a Preferred Size for the Popover, 425
- Set a Size for the Editor Popover, 391
- Set the Popover Presentation Controller Delegate, 394
- Setting a Custom Height and Width for the Picker Components and Rows, 436
- Setting a Default Selection, 438
- Setting an Exit Point, 367

- Setting the Detail View Controller's detailItem, 519
- Setting the End of Background Processing, 805
- Setting the Initial Scene's Label to the Editor Scene's Field, 389
- Setting the Start on Background Processing, 805
- Setting the Status Bar Appearance in preferredStatusBarStyle, 274, 745, 766
- Setting Up and Displaying the Image Picker, 680
- A Silly Implementation of tableView:cellForRowAt IndexPath, 493
- Starting and Stopping the Animation in toggleAnimation, 269-270
- Storing the Recently Received Location for Later Use, 772-773
- Supporting All Interface Orientations, 602
- Typical Setup and Display of a Media Picker, 673
- Update the viewDidLoad Method to Ask for Location Authorization, 738
- ateAnimalChooserViewController's viewDidLoad Method, 439
- Updating showAlert to Register a Local Notification, 793-794
- Updating the Counter, 804
- Updating the Display in viewWillAppear:animated, 469
- Updating the Display Using the Counter Values, 477
- Updating the Initial recordAudio Method, 694
- Updating the Settings in viewDidLoad, 545
- Updating the Tab Bar Item's Badge, 477-478
- Updating the viewDidLoad Method to Set the Initial Display, 302
- Updating viewDidLoad to Loop 2,000 Times, 851
- Using prepareForSegue:sender to Grab the View Controllers, 375
- Using the Motion Manager, 646
- The ViewController.swift Outlets and Actions, 266-267
- ViewController.swift with Connections Defined, 198
- Watching for Orientation Changes, 649
- Your First Code Exercise, 44
- lldb, 845**
- loading**
  - hi-res images for Retina display, 258
  - images and details, FloraPhotographs project, 300-302
  - picker data, 432-433
  - recorded sound, 696-697
  - remote content, web views, 284-285
  - sound, 326-327
- loadRequest, loading remote content, 284-285**
- Local Authentication, 125**
- local notifications, backgrounding, 784-785, 792**
  - creating/scheduling, 793-794
  - properties, 793
  - requesting authorization for notifications, 792
- local repositories, Git, 866-868**
- location accuracy, Core Location, 756**
- location constants, Cupertino project, 759-760**
- location errors, 754-756**
- location manager, Cupertino project, 762-763**
- location manager delegate implementing, 763-766**
- location manager delegate protocol, 752-754**
- location manager instance, configuring, 763**
- location managers, Core Location, 751-752**
- locations**
  - batteries, 756
  - Core Location. See Core Location
  - mapping, 728-729
  - north, 757
  - permissions, requesting to use user's location, 737-738
  - storing recent, Cupertino project, 772-773

**locMan, 764**

**Log mode, 876**

**long pressing, 612**

**long-running background tasks, 800**

SlowCount project

application logic, 802-804

background task processing, 804-805

designing interfaces, 802

implementation overview, 800

outlets, 802

setting up, 801

**long-running tasks, task completion for long-running tasks, 785-786**

**loops**

do-while loops, 106-107

for loops, 105-106

while loops, 106-107

**low-level code, 446**

## M

**Mac Developer Program, 19**

**magnetic compass, 768**

**magnetic north, 757**

**mail completion, BestFriend project, 742**

**mail compose delegate protocol, 741**

**mail compose view, displaying, 741-742**

**mail compose view controller delegate, 719**

**Main.storyboard file, 196**

**managing, project properties, Xcode, 58**

**map displays**

controlling map display, 738-740

tying to Address Book selection, 740-741

**Map Kit, 120, 721-723**

**map logic, BestFriend project, 737**

controlling map display, 738-740

customizing pin annotation view, 740

requesting permission to use user's location, 737-738

tying map display to Address Book selection, 740-741

**map view delegate protocol, 724-725**

**map views, configuring, 732**

**mapping, 721-723**

annotations, 723-724

map view delegate protocol, 724-725

geocoding, 725-728

permissions, 728-729

**maps, BestFriend project. See BestFriend project**

**MARK, 47-48**

**master scenes, updating, 510**

**master view controllers, 497**

FlowerDetail project, 515

creating table cells, 516-517

creating table view data methods, 515-516

disabling editing, 518

handling navigation events from a segue, 518-519

**Master-Detail Application template, 497-498, 507**

FlowerDetail project

application data source, 512-515

master view controllers, 515-519

setting up, 508-510

tweaking interfaces, 510-512

implementation overview, 507-508

**matching sizes, constraints, 598-600**

**media files, adding, 684**

**media items, accessing, 675-676**

**Media layer, 121**

AV Foundation framework, 121

Core Audio, 121

- Core Graphics, 122
- Core Image, 121
- Core Text, 122
- Image I/O, 122
- Media Player framework, 122
- Metal, 122
- OpenGL ES, 122
- Photos framework, 122
- Quartz Core, 122
- media picker, 673-674**
  - displaying, 704-705
  - filters, 673
  - music library, 702-703
- media picker controller delegate, 674**
- media player**
  - audio formats, 671
  - transitioning to fullscreen view, 671
- Media Player framework, 122, 670-671**
  - accessing media items, 675-676
  - media picker, 673-674
  - media picker controller delegate, 674
  - movie player, 671
  - movie player completion, 672
  - music player, 675
- media selections, 674**
- MediaPlayground project**
  - audio playback, 692-693, 695-696
    - controlling, 696
    - loading recorded sound, 696-697
  - audio recording, 692-693
    - controlling, 694-695
    - implementing, 693-694
  - Core Image filter, 700-702
  - designing interfaces, 685-686
  - implementation overview, 683-684
  - movie player, 689
    - cleanup, 691
    - implementing playback, 690-691
    - initializing movie player instance, 689-690
  - music library, 702
    - displaying media picker, 704-705
    - empty selections, 706
    - media picker, 702-703
    - playing music, 706-708
    - preparing music player, 703-704
    - user's selection, 705
  - outlets and actions, 687-688
  - photo library and camera, 697-700
  - setting up, 684-685
  - variables, 685
- memory management, 107**
- memory usage, monitoring, 855-856**
- merging, source control, 865, 877-880**
- Message UI framework, 121, 741**
- Metal, 122**
- method stubs, 178**
- methods**
  - chaining, 101
  - closures, 102
  - declaring, 86-87
  - versus functions, 190
  - syntax, 99-100
- MFMailComposeViewController, 742**
- Misplaced Views error, 587**
- MKMapView, 728**
- MKMapViewDelegate protocol, 740**
- Modal Editor project, 377**
  - application logic, 388-389
    - hiding keyboards, 389
  - connections, 381
  - designing interfaces, 381-384

implementation overview, 377

modal segues, creating, 384-385

outlets and actions, 386-387

popovers

    configuring, 391-392

    iPhones, 393-395

setting up, 377-381

toggling to universal applications, 392-393

unwinding back to the initial scene, 386

variables, 381

**modal scenes, dismissing, programmatically, 366**

**modal segues**

    creating, 359-361, 384-385

    presenting manually, 366

**modal user interface, 318**

**modal views, 351**

**Model-View-Controller (MVC), 24**

**monitoring CPU and memory usage, 855-856**

**motion data, accessing, 643**

**motion hardware**

    accelerometers, 640-642

    gyroscope, 642

**Motion Manager, Core Motion, 656-657**

**motion sensing, 639**

**motion updates, managing, 657-658**

**motion-input mechanisms, 639**

**movie playback, implementing, 690-691**

**movie player, 671**

    MediaPlayground project, 689

        cleanup, 691

        implementing playback, 690-691

        initializing movie player instance, 689-690

**movie player instance, initializing, 689-690**

**MPMediaItem, 670, 675-676**

**MPMediaItemCollection, 670, 674**

**MPMediaPickerController, 670, 673-674**

**MPMoviePlayerController, 670**

**MPMusicPlayerController, 670**

**multibutton alerts, creating, 334-336**

**multiscene development, view controllers, 446**

**multiscene projects, 352**

    exits, 366-368

    modal segues

        creating, 359-361

        presenting manually, 366

    naming scenes, 354-355

    passing data between scenes, 375-377

    popover segue, configuring, 362-365

    popovers, 371-374

        popover arrow, 372-373

        PopoverPresentation

            ControllerDelegate Protocol, 374

    segues

        creating, 356-359

        programming from scratch, 369-371

    view controller subclasses, 355-356

**multiscene projects adding additional scenes, 352-353**

**multiscene storyboards, 350**

    multiscene projects. See multiscene projects

**multitouch events, generating, 66-67**

**multitouch gesture recognition, 611-612**

**multivalue picker, 552, 556**

**music library**

    displaying media picker, 704-705

    empty selections, 706

    MediaPlayground project, 702

        media picker, 702-703

    playing music, 706-708

    preparing music player, 703-704

    user's selection, 705

**music player, 675**

preparing, 703-704

**mutators, 85**

**MVC (Model-View-Controller), 24, 185**

data models, 190

limitations, 186

Single View Application template. See Single View Application template

structured application design, 186-187

view controllers, 188

@IBAction, 189-190

@IBOutlet, 188-189

views, 187-188

**myInstanceMethod, 97**

**myOptionalString, 97**

## N

**named parameters, 100**

**naming, scenes, 354-355**

**navigating**

code, Xcode, 42

constraints objects, 581-590

documentation, Xcode, 139-140

projects, Xcode, 34-35

**navigation bar item attributes, setting up, 450-451**

**navigation bars, 448**

**navigation controller classes, adding, 459-460**

**navigation controllers, 445, 447**

bar button items, 448

items, 448

navigation bars, 448

people picker navigation controller delegates, 715-716

projects, 458

LetsNavigate project. See LetsNavigate project

sharing data between navigation scenes, 452

storyboards, 449-450

adding navigation scenes with show segues, 451-452

setting navigation bar item attributes, 450-451

**navigation events, segues, 518-519**

**navigation scenes, adding with show segues, 451-452**

**navigator, Xcode, 33**

**Newsstand, 124**

**nonbridged data types, 130-131**

URLs (NSURL), 131

**non-Retina display, 61**

**north, locations, 757**

**Notification Center framework, 121**

**NSDate, 130, 423-424**

**NSDateFormatter, 423-424**

**NSFileHandle, 538**

**NSLog, 838-839**

viewing output, 839-841

**NSNotificationCenter, 647, 672**

**NSOperationQueue, 647**

**NSsearchPathForDirectories**

InDomains, 536

**NSTimeInterval, 420**

**NSURL, 131**

loading remote content, 284-285

**NSURLRequest, loading remote content, 284-285**

**number formatters, 765**

**numberOfComponentsInPickerView, 408**

**numberOfSectionsInTableView, 491**

**numberOfTapsRequired, 613**

**numberOfTouchesRequired, 613**

## O

object data types, 93

object identity, 178-179

Object Library, user interfaces, 154-155

Objective-C, 81, 129

object-oriented programming. *See* OOP (object-oriented programming)

imperative programming, 78

object-oriented approach, 78-79

objects, 513

adding

to interfaces, 201-203

to scrolling view, 308

adding to views, 156-157

OOP (object-oriented programming), 80

onscreen controls (UIControl), 129

OOP (object-oriented programming), 78-79

terminology, 79-81

OpenGL ES, 122

opening projects, Interface Builder (IB), 170

opinionated software, 527-528

optional binding, 98-99

optional chaining, 101

optional values, 95-96, 99

declaring, 96-97

optional binding, 98-99

unwrapping, 97-98

orientation

constraints. *See* constraints

determining, 650-651

screen orientations, 575

sensing. *See* sensing orientation

orientation changes

AllInCode project, 602

orientation data, accessing, 643

orientation notifications, requesting, through  
UIDevice, 644

Orientation project

application logic, determining orientation,  
650-651

designing interfaces, 648

implementation overview, 647

outlets, 649

sensing orientation, application logic, 649-651

setting up, 647

orientation updates, registering, 649-650

orientationChanged method, 650

originalRect, 617

outlets

BackgroundColor project, 541-543

BackgroundDownload project, 807-808

BestFriend project, 732-733

ColorTilt project, 654-655

Cupertino project, 762, 769, 771

CustomPicker project, 431-432

DateCalc project, 418-419

FieldButtonFun project, 237-239

FloraPhotographs project, 295-298

Gestures project, 623-625

GettingAttention project, 331-333

ImageHop project, 264-265

Interface Builder (IB), 172-178

LetsNavigate project, 466-467

LetsTab project, 475-476

MediaPlayground project, 687-688

Modal Editor project, 386-387

Orientation project, 649

ReturnMe project, 549

Scroller project, 310

setting up, 197-199

Single View Application template, 203-207

SlowCount project, 802

Survey project, 562-563

**output.** *See also* **input**

from Playground, generating and inspecting  
output, 110-112

**outputLabel**, 418, 475

## P

**pagination**, scrolling views, 308

**paid Developer Program**, joining, 12-14

**panning**, 612

**parameters**, 142

named parameters, 100

OOP (object-oriented programming), 80

**parent classes**, OOP (object-oriented programming),  
79

**parent controllers**, 446

**parentViewController**, 468

**Pass Kit**, 124

**passing data between scenes**, 375-377

**passthrough views**, 364-365

**pausing**, playback, media player, 671

**people picker delegate protocol**, conforming to, 734

**people picker navigation controller delegates**,  
715-716

**peoplePickerNavigationControllerDidCancel**, 715

**peoplePickerNavigationController:didSelectPerson**,  
715

**permissions**

mapping, 728-729

requesting to use user's location, 737-738

**photo library**, MediaPlayground project, 697-700

**Photos framework**, 122

**PhotosUI**, 121

**picker data**, loading, 432-433

**picker view data source protocol**, 408-409

**picker views**, 407-408

advanced delegate methods, 410-412

picker view data source protocol, 408-409

picker view delegate protocol, 409-410

**pickers**, 134, 401, 404-405

custom pickers. *See* custom pickers

date pickers, 406

attributes, 406-407

projects. *See* DateCalc project

picker views, 407-408

advanced delegate methods, 410-412

picker view data source protocol, 408-409

picker view delegate protocol, 409-410

**pickerView:didSelectRow:inComponent**, 409

**pickerView:numberOfRowsInComponent**, 408

**pickerView:rowHeightForComponent**, 410

**pickerView:titleForRow:forComponent**, 409

**pickerView:viewForRow:viewForComponent:Reusing  
View**, 410

**pickerView:widthForComponent**, 410

**pin annotation view**, customizing, 740

**pinch recognizer**, 621-622

responding to, 627-630

**pinching**, 612

**pinning**, 581

**placement errors**, 588

**plain tables**, 486

**plain text versus attributed text**, 227

**play**, 689

**play method**, media player, 671

**playAudio**, 696

**autoplay**, implementing, 690-691

**playbackState, 675****Playground feature, 108**

- Cocoa Touch classes, 131-132
- creating new, 108-109
- dates and times, 423-424
- FieldButtonFun project, 244
- file storage, 538-539
- generating and inspecting output, 110-112
- geocoding, 727-728
- ImageHop project, 272-273
- testing, user notifications, 324-325
- testing web views, 285-286
- transformations, 632-633
- using, 110

**playing**

- alert sounds with vibrations, 342
- music, music library, 706-708
- sound, 326-327
- System Sound Services, 341-342

**playMovieFinished, 672****playMusic method, 706****plist files**

- requesting, 752
- updating, 766

**PNG images, 40****pop, 447****popover arrow, 372-373****popover segue, configuring, 362-365****popoverPresentationController, 394****popovers, 134-135, 340, 371-374, 439**

- configuring, 391-392
- iPhones, 393-395
- popover arrow, 372-373
- preparing, 362-363
- sizing, 363
- UIPrPresentationController
  - Delegate Protocol, 374

**populating**

- arrays, FlowerColorTable project, 502
- data structures, 515

**posting to social networking sites, 720-721****preference types, 533****preferences**

- BackgroundColor preferences, reading, 545
- storing in BackgroundColor project, 544-545

**preferredStatusBarStyle, 745****prepareForSegue:sender method, 375-377****presentation directions, 364-365****presentation styles, segues, 370****presentingViewController, 422****preventing, interface-orientation changes, 661-662****previewing, interfaces, 168-169****print, 838****println, 838****private, 87****products, 35****programmatically defined interfaces, 600-601**

- AllInCode project
  - programming interfaces, 602-606
  - setting up, 602
- implementation overview, 601-602

**programming interfaces**

- AllInCode project
  - button touches, 606
  - defining variables and methods, 603
  - drawing interfaces when the application launches, 606
  - implementing interface update method, 604-605
  - initializing interface objects, 603-604
  - updating the interface when orientation changes, 606
- rotation, 577

**project code, 35**

**project groups, 35**

**project properties, managing, 58**

**project types, Xcode, 30-32**

## projects

AllInCode project. See AllInCode project

BackgroundColor project. See BackgroundColor project

BackgroundDownload project. See BackgroundDownload project

BestFriend project. See BestFriend project

ColorTilt project. See ColorTilt project  
configuring as universal, 816-817

Cupertino project. See Cupertino project

CustomPicker project. See CustomPicker project

DateCalc project. See DateCalc project

DebuggerPractice project, 843-845

FieldButtonFun project. See FieldButtonFun project

FloraPhotographs project. See FloraPhotographs project

FlowerColorTable project. See FlowerColorTable project

FlowerDetail project. See FlowerDetail project

Gestures project. See Gestures project

GettingAttention project. See GettingAttention project

ImageHop project. See ImageHop project

MediaPlayground project. See MediaPlayground project

Modal Editor project. See Modal Editor project

multiscene projects. See multiscene projects  
navigation controllers, 458

LetsNavigate project. See LetsNavigate project

Orientation project. See Orientation project

ReturnMe project. See ReturnMe project

Scroller project. See Scroller project

Single View Application template, 192-193

class files, 194-195

planning variables and connections, 197-199

storyboard files, 195-197

SlowCount project. See SlowCount project

source control, 871

branching/merging, 877-880

commits, 873-874

pulls, 874

pushes, 873-874

status codes, 871-872

updates, 874

viewing revisions, 874-877

Survey project. See Survey project

Swapper project. See Swapper project

tab bar controllers, 469

LetsTab project. See LetsTab project

Xcode, 30

adding new assets catalogs, 38

adding resources, 37

choosing project types, 30-32

getting your bearings, 33-34

navigating, 34-35

removing files and resources, 37-38

Xcode Asset Catalog, 38

## ProjectTests, 35

### properties, 551

child properties, 553

local notifications, backgrounding, 793

Values property, 552

versus variables, 189

### protocols, 84

OOP (object-oriented programming), 80

### prototype cell attributes, 489-491

- provisioning profiles, viewing, 22
- pseudo preferences, 529-530
- public, 87
- pulls, source control projects, 874
- push, 447
- push count variable property, adding, 468
  - LetsTab project, 476-477
- pushCount, 463
- pushes, source control projects, 873-874

## Q

- Quartz Core, 122
- Quick Help, Xcode, 140-141
  - activating Quick Help Inspector, 141-142
  - interpreting results, 142-143
- Quick Help Inspector, activating, 141-142
- Quick Inspector, editing, connections, 177
- Quick Look, 124
- quitting apps, 209

## R

- radian conversion constants ColorTilt project, 653
- radian/degree conversion constants, Cupertino project, 769
- radio buttons, 282
- reading
  - acceleration with Core Motion, 645-647
  - attitude with Core Motion, 645-647
  - BackgroundColor preferences, 545
  - data, direct file system access, 537-538
  - rotation, with Core Motion, 645-647
  - user defaults, 531-532
- recordAudio method, 693-694, 696

- recorded sound, loading, 696-697
- registering
  - as a developer, Apple Developer Program, 11
  - multiple devices, 17
  - orientation updates, 649-650
- related, 142
- relationships, 351
  - tab bar relationships, creating, 472-473
- Release, 842
- remote content, loading, with web views, 284-285
- remote repositories, Git, 868-869
- removeConstraints, 615
- removing
  - annotations from map view, 724
  - breakpoints, 854
  - files and resources, Xcode projects, 37-38
  - objects from views, 156
- repeatInterval, 793
- replacing image views, Gestures project, 626
- repositories
  - Git, 865-866
    - connecting to remote repositories, 868-869
    - creating local, 866-868
    - source control, 864
- requestAlwaysAuthorization, 752
- requesting
  - authorization, Core Location, 752
  - authorization for notifications, backgrounding, 792
  - orientation notifications, through UIDevice, 644
  - permissions, to use user's location, 737-738
  - plist files, 752
- requestWhenInUseAuthorization, 737, 752
- resetting View Controller Simulated Size, 309
- resizable interfaces, Auto Layout, 576-577
- resources
  - adding to Xcode projects, 37
  - removing from Xcode projects, 37-38

**responders (UIResponder), 128-129**

**responding**

- to pinch recognizer, 627-630
- to rotation recognizer, 630-632
- to swipe recognizer, 627
- to tap gesture recognizers, 627

**responding to actions, alert controllers, 321-322**

**responsive interfaces**

- designing rotatable and resizable interfaces
  - Auto Layout, 576-577
  - programming interfaces, 577
  - size classes, 578
  - swapping views, 577-578
- rotation, 573-574
  - enabling, 574-575

**restoration, pseudo preferences, 529-530**

**retina display, hi-res images, loading, 258**

**retina image assets, 41-42**

**Retina image files, 61**

**Retina-naming convention, 42**

**return statement, 575**

**ReturnMe project**

- application logic, 557-559
- building apps, 559
- designing interfaces, 548
- outlets, 549
- setting up, 547-548
- settings bundles, creating, 549-556

**returns, 142**

**revisions, viewing, source control, 874-877**

**rich media**

- AV Foundation framework, 676-677
  - AV Audio Player, 677
  - AV Audio Player completion, 677
  - AV audio recorder, 678-679

Core Image, 682

filters, 682-683

image picker, 679-680

UI image picker controller delegate, 680-682

Media Player framework, 670-671

accessing media items, 675-676

media picker, 673-674

media picker controller delegate, 674

movie player, 671

movie player completion, 672

music player, 675

MediaPlayground project. See MediaPlayer project

**Root.plist file, 554**

**rootViewController, 809**

**rotatable interfaces, Auto Layout, 576-577**

**rotating, 612**

simulated devices, 67

**rotation, 573-574**

enabling, 574-575

programming interfaces, 577

reacting to, 661

reading, with Core Motion, 645-647

screen-locking function, 578

size classes, 578

swapping views, Swapper project. See Swapper project

**rotation recognizer, 622**

responding to, 630-632

**rotationRate, 646**

**running**

apps, Xcode, 53-54

iOS apps, 16-19

## S

**sample code, 137, 142**

**sandbox, 535**

**scaling, 294**

**scene segue logic**

CustomPicker project, 438-439

DateCalc project, 424-425

**scenes, 188, 351**

adding, 352-353

LetsNavigate project, 461-462

LetsTab project, 472

naming, 354-355

passing data between, 375-377

**scheduling notifications, backgrounding, 793-794**

**scrolling views, Auto Layout, 311**

**screen orientations, 575**

**screen-edge panning, 612**

**screen-locking function, 578**

**screens**

accommodating different screens, 7

iPads, 6

iPhone 5, 7

iPhone 6, 6

iPhone 6+, 6

iPhones, 6

**Scroller project**

application logic, 310

adding scrolling behavior, 310-311

designing interfaces, 305

adding objects, 308

adding scrolling views, 305-306

resetting View Controller Simulated Size, 309

setting freeform size, 306-307

outlets and actions, 310

**scrolling behavior, adding, 310-311**

**scrolling options, FieldButtonFun project, 233-234**

**scrolling views, 286, 303-304**

adding to Scroller project, 305-306

building apps, 312

implementation overview, 304-305

pagination, 308

Scroller project. See Scroller project

**SDK (software development kit), 534**

**SDK Guides, 137**

**search navigator, searching code, Xcode, 46-47**

**searching**

code, Xcode, 46-47

documentation, Xcode, 137-138

**Security framework, 125**

**segmented controls, 133, 282-283**

adding, FloraPhotographs project, 289

iOS 6, 290

sizing in FloraPhotographs project, 291

**segments, adding/configuring, 289-290**

**Segue drop-down, 359**

**segue style, configuring, 370-371**

**segues, 351, 377, 446**

adaptive segues, 358

disabling, 394

creating, 356-359

CustomPicker project, 431

DateCalc project, 417

Modal Editor project, 377

implementation overview, 377

setting up, 377-381

modal segues

creating, 359-361

presenting manually, 366

navigation events, 518-519

popover segue, configuring, 362-365

programming from scratch, 369-371

- show segues, adding navigation scenes, 451-452
- unwind segues, 366-368, 389
- selection handles, Interface Builder (IB), 158-159**
- selections**
  - empty selections, 706
  - music library, 705
- self, OOP (object-oriented programming), 80-81**
- sender variable, 630**
- sendMail method, 741**
- sendTweet method, 742**
- sensing orientation, 647**
  - implementation overview, 647
  - Orientation project
    - application logic, 649-651
    - designing interfaces, 648
    - outlets, 649
    - setting up, 647
- setBackgroundHueValue method, 542-544**
- setDateTime, 418, 422-423**
- setFullscreen:animated, 689**
- setMessageBody:isHTML, 742**
- setMinimumBackgroundFetchInterval, 808**
- setObject:forKey, 531**
- setOutput method, 208**
- setters, 85**
- Settings application, 528, 546-547**
  - implementation overview, 547
  - ReturnMe project, 547-548
    - application logic, 557-559
    - designing interfaces, 548
    - outlets, 549
    - setting up, 547-548
    - settings bundles, 549-556
- settings bundles, 528, 530, 532-534, 547**
  - creating, 549-556

- Settings icon, 61**
- setToRecipients method, 719**
- setValuesFromPreferences, 557-558**
- shake gestures, 634-635**
- shake recognizer, 632**
  - first responders, 632-634
- shaking, 612**
- sharing data**
  - between navigation scenes, 452
  - between tab bar scenes, 457-458
- show segues**
  - adding, navigation scenes, 451-452
  - creating, 464-465
- showing**
  - chosen images, 698-699
  - detail web views, FloraPhotographs project, 298-300
  - survey results, Survey project, 565-567
- showResults method, 566**
- signing identity, 17**
- simulated devices**
  - adding, iOS Simulator, 69-71
  - rotating, 67
- simulated interface attributes**
  - FieldButtonFun project, 224
  - setting, 199-201
- Single View Application template, 191**
  - application logic, implementing, 208
  - applications, building, 208-209
  - designing interfaces, 199-203
  - implementing, 191-192
  - outlets and actions, 203-207
  - projects, 192-193
    - class files, 194-195
    - planning variables and connections, 197-199
    - storyboard files, 195-197

**singleton, OOP (object-oriented programming), 80**

**size classes**

- Any, 833
- configuring installed, 823
- designing rotatable and resizable interfaces, 578
- fonts, 826
- images, 826-827
- setting active size classes, 821-823
- setting manually, 824-825
- storyboards, 827-833
- tools, 821
- universal applications, 818-820
  - tools, 821-827

**Size Inspector**

- editing, constraints, 582-585
- Interface Builder (IB), 159-161
- viewing, constraints, 582-585

**sizes, matching, constraints, 598-600**

**sizing, 490**

- controls, FloraPhotographs project, 291
- popovers, 363

**SLComposeViewController, 720**

**slices, creating, 220-224**

**slicing, 40, 220-224**

- button templates, 219
- adding images, 220

**slider range attributes, setting, 259-261**

**sliders, 133, 251-252**

- adding, ImageHop project, 259
- ImageHop project. See ImageHop project

**SlowCount project**

- application logic, 802-804
- background task processing, 804-805
- designing interfaces, 802
- implementation overview, 800
- outlets, 802
- setting up, 801

**snapshots, Xcode, 50-51**

**Social, 124**

**social networking, BestFriend project. See BestFriend project**

**social networking logic, BestFriend project, 742-743**

- displaying compose view, 743-744

**social networking sites, posting to, 720-721**

**sound**

- loading, 326-327
- user notifications, 326

**sound resources, 328**

**soundName, 793**

**sounds, alerting users, 327-328**

**soundSetting, 679**

**source control, 863**

- branching/merging, 865
- committing changes, 864
- downloading changes, 865
- projects, 871
  - branching/merging, 877-880
  - commits, 873-874
  - pulls, 874
  - pushes, 873-874
  - status codes, 871-872
  - updates, 874
  - viewing revisions, 874-877
- repositories, 864
- working copies, 864

**space, data storage, 536**

**speed, animation speed**

- ImageHop project, 270-272
- incrementing, 273-274

**speed output labels, adding, 262**

**split view controllers, 495**

- hierarchies, FlowerDetail project, 509
- implementing, 496-497
- Master-Detail Application template, 497-498

**splitViewController**, 497

**startAnimating**, 269

**starting**

animation, ImageHop project, 269-270

segues, 366

**startUpdatingLocation** method, 763

**state preservation**, 529-530

**status bars**

setting to white, 744-745

Cupertino project, 766

unreadable status bar, 274

**status codes**, source control, projects, 871-872

**stepper range attributes**, setting up, 261-262

**steppers**, 133-134, 252-253

adding, 261

ImageHop project. See ImageHop project

**stopAnimating**, 269

**stopping animation**, ImageHop project, 269-270

**storage locations for application data**, 535-536

**Store Kit**, 125

**storeSurvey** method, 564-565

**storing**

BackgroundColor preferences, 544-545

recent locations, 772-773

survey results, 564-565

**storyboard feature**, 446

**storyboard files**, Single View Application template, projects, 195-197

**storyboard identifiers**, setting, 369

**storyboard segues**, 446

**storyboards**, 188, 351

Interface Builder (IB), 149

document outline, 149-152

document outline objects, 153-154

multiscene storyboards. See multiscene storyboards

navigation controllers, 449-450

adding navigation scenes with show segues, 451-452

setting navigation bar item attributes, 450-451

size classes, 827-833

tab bar controllers, 454-455

adding tab bar scenes, 456-457

setting tab bar item attributes, 455-456

universal applications, 818

**String** objects, 89, 536

**stringForKey**, 532

**strings**, 90-91

**structs**, 88

**stub methods**, 178

**styled buttons**, adding, 234

**subclasses**, OOP (object-oriented programming), 79

**subversion**. See SVN

**subviews**, 152

**superclasses**, OOP (object-oriented programming), 79

**superviews**, 152, 579

**Supported Device Orientations**, 59

**supportedInterfaceOrientations** method, 574, 602

**Supporting Files** group, 35

**Survey** project

application logic

hiding keyboards, 564

showing survey results, 565-567

storing survey results, 564-565

designing interfaces, 560-561

outlets and actions, 562-563

setting up, 560

**survey results**

showing, 565-567

storing, 564-565

**suspension, backgrounding, 784, 790-791****SVN, 863**

- branching/merging, 865
- committing changes, 864
- downloading changes, 865

**swapping views**

- designing rotatable and resizable interfaces, 577-578
- rotation, Swapper project. *See* Swapper project

**Swift, 23, 77, 81**

- automatic reference counting, 107
- class files, 82-83
  - class declaration, 84
  - constant declaration, 85
  - declaring methods, 86-87
  - ending, 87
  - IBOutlet declarations, 85-86
  - import declaration, 83-84
  - variable properties declarations, 84-85
  - Xcode, 88
- decision making, 102
- declaring variables and constants, 89
  - arrays, 91-92
  - Boolean values, 91
  - constants, 95
  - convenience methods, 93-94
  - data types, 89
  - dictionaries, 92-93
  - integers and floating-point numbers, 89-90
  - object data types, 93
  - optional values, 95-97
  - strings, 90-91

- expressions, 102-103
  - if-then-else, 103-104
  - loops, 104-107
  - switch statements, 103-104
- memory management, 107
- methods
  - chaining, 101
  - closures, 102
  - syntax, 99-100
- object-oriented programming, 77-78
- properties versus variables, 189
- type casting, 95
- unwrapping, 97-98

**Swift optional binding, 98-99****Swift type conversion, 94****swipe recognizer, 621****swipe recognizer responding to, 627****swiping, 612****switch statements, 103-104****switches, 133, 282**

- adding, in FloraPhotographs project, 291

**symbol navigator, Xcode, 43****synchronize method, 532****syntax**

- expressions, 103
- methods, 99-100

**System Configuration, 125****System framework, 125****System Sound Services, 325-326, 796**

- accessing, 326-327
- alert sounds and vibrations, 327-328
- playing, 341-342

**SystemSoundIDs, 796**

## T

- tab bar controller classes, adding, 471**
- tab bar controllers, 445, 452-453**
  - adding, 471-472
  - projects, 469
    - LetsTab project. See LetsTab project
  - storyboards, 454-455
    - adding tab bar scenes, 456-457
    - setting tab bar item attributes, 455-456
  - tab bar items, 453-454
  - tab bars, 453-454
- tab bar images, 456**
- tab bar item attributes, setting, 455-456**
- tab bar item badge, incrementing, 477-478**
- tab bar item images, adding, 471**
- tab bar items, 453-454**
- tab bar relationships, creating, 472-473**
- tab bar scenes**
  - adding, 456-457
  - sharing data, 457-458
- tab bars, 453-454**
- Tabbed Application, 454**
- tabbed editing, Xcode, 50**
- table attributes, setting, 488-489**
- table cells, 487**
  - creating, 516-517
- table section constants, 499**
- table view controllers, 486**
- table view data methods, creating, 515-516**
- table view data source protocols, FlowerColorTable project, 503-505**
- table view delegate protocols, FlowerColorTable project, 505-507**

## table views, 486

- adding, 488
  - data source protocols, 491-494
  - delegate protocols, 494-495
  - prototype cell attributes, 489-491
  - setting table attributes, 488-489

## tables

- FlowerColorTable project
  - implementation overview, 499
  - setting up, 499
- grouped tables, 486
- indexed tables, 486
- plain tables, 486
- table view app, FlowerColorTable project. See FlowerColorTable project

**tableView:cellForRowAtIndexPath, 491**

**tableView:numberOfRowsInSection, 491**

**tableView:titleForHeaderInSection, 491**

**tap gesture recognizers, responding to, 627**

**tap recognizer, 619-620**

**tapping, 612-613**

**task completion for long-running tasks, backgrounding, 785-786**

**task-specific background processing, 785, 795**

- adding audio, 795-796
- background modes, adding, 799-800
- Cupertino project, adding audio directions, 796-799

## templates

- button templates. See button templates
- Master-Detail Application template. See Master-Detail Application template

Single View Application template. See Single View Application template

Tabbed Application, 454

## testing

apps, FloraPhotographs project, 303

conditions, iOS Simulator, 68-69

transformations, Playground feature, 632-633

user notifications, 324-325

web views, 285-286

text, plain versus attributed, 227

text field attributes, customizing keyboard displays, 228-229

text fields, 134, 216

adding, to FieldbuttonFun project, 225

editing attributes, 225-227

text view attributes, editing, 231-232

text views, 216, 219

adding to FieldbuttonFun project, 230-231

tilt, 642

ColorTilt project. See ColorTilt project

Time, date picker attributes, 406

timeIntervalSinceDate, 420

timers, initializing, 803

timeZone, 793

TODO, 47-48

toggleAnimation method, 269

toggleFlowerDetail, 288

toggleSwitch, 540

toggling to universal applications, 392-393

toolbar, Xcode, 33

toolbars, 401

bar button items, 403-404

role of, 401-402

## tools

Apple Developer tools, 23

Cocoa Touch, 24

Model-View-Controller (MVC), 24

Swift, 23

bar button items, attributes, 404

editing tools. See editing tools

size classes, 821

Top Layout Guide, 151

top/bottom layout guides, 581

## touch

background touch, keyboard hiding, 242

gesture recognizers, adding, 612-613

## tracing

iOS application lifecycle, 126-127

lifecycle of iOS apps, 126-127

trailing, 579

trailing space, 581

Traits attributes, 166

transformations, Playground feature, 632-633

transition styles, segues, 370

transition types, 360

transitioning, to fullscreen view, media player, 671

triggering counter updates, LetsTab project, 478-479

tutorials, accessing, 880-884

tweaking interfaces, FlowerDetail project, 510-512

updating detail scenes, 511-512

updating master scenes, 510

web view outlets, 512

type casting, 95

type conversion, 94

type method, OOP (object-oriented programming), 80

## U

### UI elements, 401

- pickers. See pickers

- toolbars. See toolbars

UI image picker controller delegate, 680-682

UIAlertController, 318

UIApplication, 128

UIBackgroundTaskIdentifier, 801

UIButton class, 132, 282

UIColor object, 540

UINavigationController, 129

UIDatePicker. See date pickers, 134

UIDevice, requesting orientation notifications, 644

UIDeviceOrientation, 644

UIDeviceOrientationDidChange  
Notification, 649

UIImage initialization method, 268

UIKit, 120

UILabel, 93, 132

UILocalNotification, 784-785, 792

UILongPressGestureRecognizer, 612

UInt32(), 327

UIPanGestureRecognizer, 612

UIPicker, 134

UIPickerViewDelegate, 409

UIPinchGestureRecognizer, 612

UIPopoverPresentationController, 134-135

UIPopoverPresentationController  
Delegate Protocol, 374

UIResponder, 128-129

UIRotationGestureRecognizer, 612, 622

UIScreenEdgePanGesture  
Recognizer, 612

UIScrollView, 312

UISegmentedControl, 133

UISlider, 133

UIStepper, 133-134

UINavigationController, 376

UISwipeGestureRecognizer, 612

UISwitch, 133

UITabBar, 453

UITabBarItem, 453

UITableViewDataSource, 491

UITapGestureRecognizer, 612

UITextField, 134

UITextView, 134

UIView, 128

UIViewController, 129

UIWebView, 312

UIWindow, 128

universal applications, 574, 815-816

- app icons, 817

- configuring projects as, 816-817

- device models, 819

- launch screens, 818

- size classes, 818-820

  - storyboards, 827-833

  - tools, 821-827

- storyboards, 818

- targets, 819

- toggling to universal applications, 392-393

unreadable status bar, ImageHop project, 274

unwind, 351

unwind destinations, determining dynamically, 368

unwind segues, 366-368, 389

unwinding back to the initial scene, Modal Editor  
project, 386

**unwrapping, 97-98**

- exclamation mark (!), 284

- update filters, Core Location, 756

- Update Frames, 382

- updates, source control projects, 874

**updating**

- counters, 803-804

- Cupertino project, application logic, 771-776

- date output, 422-423

- detail scenes, 511-512

- displaying, 803-804

- master scenes, 510

- plist files, 766

- user interfaces, Cupertino project, 769-770

- URLs (NSURL), 131

- user defaults, 530-531

- reading/writing, 531-532

**user interfaces**

- adding object to views, 156-157

- Object Library, 154-155

- updating, Cupertino project, 769-770

**user notifications, 317**

- alert controllers, 318

- action sheets, 322-324

- alerts, 318-321

- responding to actions, 321-322

- GettingAttention project. See GettingAttention project

- System Sound Services, 325-326

- accessing, 326-327

- alert sounds and vibrations, 327-328

- testing, 324-325

- vibrations, 341-343

- userAcceleration, 646

- utility, Xcode, 33

**V**

- Values property, 552, 556

- Variable List, accessing, 852-853

- variable properties declarations, 84-85

**variable property**

- for image view size, adding, 616-617

- OOP (object-oriented programming), 80

- variable states, debuggers, 847-848

**variables**

- AllInCode project, 602

- BestFriend project, 730-731

- ColorTilt project, 652-653

- Cupertino project, 759, 769

- CustomPicker project, 427-428

- DateCalc project, 414-415

- declaring, 89

- arrays, 91-92

- Boolean values, 91

- convenience methods, 93-94

- data types, 89

- dictionaries, 92-93

- integers and floating-point numbers, 89-90

- object data types, 93

- optional values, 95-97

- strings, 90-91

- FloraPhotographs project, 288

- FlowerColorTable project, 499

- FlowerDetail project, 509-510

- Gestures project, 616

- ImageHop project, 255

- LetsNavigate project, 463

- LetsTab project, 472

- MediaPlayground project, 685

- Modal Editor project, 381
- OOP (object-oriented programming), 80
- Orientation project, 647
- planning, 197-199
- versus properties, 189
- SlowCount project, 801
- verifying connections, Connections Inspector, 625**
- Version editor, viewing revisions, 875-876**
- vertical constraints, 581**
- verticalAccuracy, 753**
- vibrations, 326-328, 341-343**
- View Controller icon, 150-151**
- View Controller Scene, 150**
- View Controller Simulated Size, resetting, 309**
- view controller subclasses, 355-356**
- view controllers, 129, 351, 445-446**
  - associating, 461-462
    - LetsTab project, 472
    - with new scenes, 379-380
  - exits, 381
  - mail compose view controller delegate, 719
  - multiscene development, 446
  - MVC (Model-View-Controller), 188
    - @IBAction, 189-190
    - @IBOutlet, 188-189
- view hierarchy, checking, 856-858**
- View icon, 151**
- ViewController.swift, 194**
- viewDidAppear method, 438, 690**
- viewDidLoad, 439, 545, 626, 843-844**
- viewing**
  - constraints, via Size Inspector, 582-585
  - NSLog output, 839-841
  - provisioning profiles, 22
  - revisions, source control projects, 874-877

- views, 351**
  - adding gesture recognizers, 619-622
  - creating in Swapper project,
  - designing, Cupertino project, 760-761
  - MVC (Model-View-Controller), 187-188
  - scrolling views. *See* scrolling views, 286
  - swapping, designing rotatable and resizable interfaces, 577-578
  - web views, 283
    - loading remote content, 284-285
    - supported content types, 284
    - testing, 285-286
- views (UIView), 128**
- virtual keyboards, 217**
- Voiceover, 165**

## W

- waitView outlet, 762**
- warnings, fixing, with issue navigator, 54-57**
- watchpoints, 851-852**
- web view outlets, FlowerDetail project, 512**
- web views, 283**
  - adding in FloraPhotographs project, 292
  - detail web views, hiding/showing, 298-300
  - loading remote content, 284-285
  - setting attributes, FloraPhotographs project, 293-294
  - supported content types, 284
  - testing, 285-286
- while loops, 106-107**
- whiteButton.png, 220**
- Wi-Fi, 9**

**willRotateToInterfaceOrientation:duration method,**  
578

**windows object (UIWindow), 128**

**Windows options for development, 10**

**working copies**

Git, 870-871

source control, 864

**writing**

code, Interface Builder (IB), 178

data, direct file system access, 537-538

user defaults, 531-532

## X-Y-Z

**Xcode, 24, 29-30, 207**

app icons, 60-62

building apps, 52

choosing build scheme, 52-53

correcting errors with issue navigator, 54-57

running, 53-54

code

activating tabbed editing, 50

adding marks, to do's and fix me's, 47-48

assistant editor, 48-49

code completion, 44-46

editing, 42

navigating, 42

searching with search navigator, 46-47

snapshots, 50-51

symbol navigator, 43

constraints, 579

device orientations, 59

documentation, 135-136

browsing, 138

navigating, 139-140

searching, 137-138

setting up documentation downloads,  
136-137

installing, 14-16

launch images/screens, 62-63

managing project properties, 58

projects, 30

adding new assets catalogs, 38

adding resources, 37

choosing project types, 30-32

getting your bearings, 33-34

navigating, 34-35

removing files and resources, 37-38

Xcode Asset Catalog, 38

Quick Help, 140-141

activating Quick Help Inspector, 141-142

interpreting results, 142-143

resources, 180

settings bundles, creating, 549-556

**Xcode 6 Debug View Hierarchy, 856**

**Xcode Asset Catalog, 38**

**Xcode editor, 42**

**Xcode slicing tool, 219**