



Big Nerd
Ranch

5TH EDITION

Cocoa Programming for OS X

THE BIG NERD RANCH GUIDE

Aaron Hillegass, Adam Preble & Nate Chandler

Cocoa Programming for OS X: The Big Nerd Ranch Guide

by Aaron Hillegass, Adam Preble and Nate Chandler

Copyright © 2015 Big Nerd Ranch, LLC.

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

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

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

Exclusive worldwide distribution of the English edition of this book by

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

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

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

ISBN-10 0134077113
ISBN-13 978-0134077116

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

For Aaron's sons, Walden and Otto

~

For Adam's daughters, Aimee and Leah

~

For Nate's nieces and nephews

This page intentionally left blank

Acknowledgments

Creating this book required the efforts of many people. We want to thank them for their help. Their contributions have made this a better book than we could have ever written alone.

Thanks to the students who took the Cocoa programming course at the Big Nerd Ranch. They helped us work the kinks out of the exercises and explanations that appear here. Their curiosity inspired us to make the book more comprehensive, and their patience made it possible.

Thank you to all the readers of the first four editions who made such great suggestions on our forums (<http://forums.bignerdranch.com/>).

Thank you to our technical reviewers, Juan Pablo Claude, Chris Morris, Nick Teissler, Pouria Almassi, and John Gallagher, who made great additions and caught many of our most egregious errors.

Finally, a very big thank you to our support team in writing this book: Liz Holaday, for copy-editing; Chris Loper, whose excellent tool chain made writing this book that much easier; and most of all Susan Loper, whose collaboration helped us write the kind of book we believe this technology deserves.

This page intentionally left blank

Table of Contents

Introduction	xvii
About This Book	xvii
Prerequisites	xviii
Typographical conventions	xviii
What's new in the fifth edition?	xviii
The Story of Cocoa	xviii
NeXTSTEP and OpenStep	xix
From NeXTSTEP to OS X to iOS	xix
OSX, Unix, and Cocoa	xx
Introducing the Swift Language	xx
The Cocoa Frameworks	xxi
Tools for Cocoa Programming	xxi
Some Advice on Learning	xxii
1. Let's Get Started	1
Creating an Xcode Project	1
Getting around in Xcode	3
Application Design	4
Model-View-Controller	4
Creating the MainWindowController class	6
Creating the User Interface in Interface Builder	8
Adding view objects	9
Configuring view objects	11
XIB files and NIB files	14
Showing the Window	14
Making Connections	17
Creating an outlet	17
Connecting an outlet	18
Defining an action method	19
Connecting actions	20
Creating the Model Layer	22
Connecting the Model Layer to the Controller	24
Improving Controller Design	24
2. Swift Types	27
Introducing Swift	27
Types in Swift	27
Using Standard Types	28
Inferring types	30
Specifying types	30
Literals and subscripting	32
Initializers	33
Properties	34
Instance methods	34
Optionals	34
Subscripting dictionaries	36
Loops and String Interpolation	36

- Enumerations and the Switch Statement 38
 - Enumerations and raw values 39
- Exploring Apple’s Swift Documentation 39
- 3. Structures and Classes 41
 - Structures 41
 - Instance methods 43
 - Operator Overloading 44
 - Classes 45
 - Designated and convenience initializers 46
 - Add an instance method 46
 - Inheritance 49
 - Computed Properties 51
 - Reference and Value Types 53
 - Implications of reference and value types 53
 - Choosing between reference and value types 54
 - Making Types Printable 54
 - Swift and Objective-C 55
 - Working with Foundation Types 56
 - Basic bridging 56
 - Bridging with collections 57
 - Runtime Errors 58
 - More Exploring of Apple’s Swift Documentation 59
 - Challenge: Safe Landing 59
 - Challenge: Vector Angle 59
- 4. Memory Management 61
 - Automatic Reference Counting 61
 - Objects have reference counts 61
 - Deallocating objects in a hierarchy 62
 - Strong and Weak References 65
 - Strong reference cycles 65
 - Unowned references 67
 - What is ARC? 68
- 5. Controls 69
 - Setting up RGBWell 70
 - Creating the MainWindowController class 70
 - Creating an empty XIB file 71
 - Creating an instance of MainWindowController 74
 - Connecting a window controller and its window 75
 - About Controls 78
 - Working with Controls 79
 - A word about NSCell 80
 - Connecting the slider’s target and action 80
 - A continuous control 82
 - Setting the slider’s range values 83
 - Adding two more sliders 85
 - NSColorWell and NSColor 86
 - Disabling a control 88
 - Using the Documentation 88

Changing the color of the color well	91
Controls and Outlets	93
Implicitly unwrapped optionals	95
For the More Curious: More on NSColor	95
For the More Curious: Setting the Target Programmatically	96
Challenge: Busy Board	96
Debugging Hints	97
6. Delegation	99
Setting up SpeakLine	99
Creating and using an Xcode snippet	101
Creating the user interface	103
Synthesizing Speech	106
Updating Buttons	107
Delegation	109
Being a delegate	110
Implementing another delegate	112
Common errors in implementing a delegate	114
Cocoa classes that have delegates	115
Delegate protocols and notifications	115
NSApplication and NSApplicationDelegate	115
The main event loop	116
For the More Curious: How Optional Delegate Methods Work	116
Challenge: Enforcing a Window's Aspect Ratio	117
7. Working with Table Views	119
About Table Views	119
Delegates and data sources	120
The table view-data source conversation	120
SpeakLine's table view and helper objects	121
Getting Voice Data	121
Retrieving friendly names	122
Adding a Table View	123
Table view and related objects	124
Tables, Cells, and Views	126
Table cell views	127
The NSTableViewDataSource Protocol	128
Conforming to the protocol	128
Connecting the dataSource outlet	128
Implementing data source methods	129
Binding the text field to the table cell view	130
The NSTableViewDelegate Protocol	131
Making a connection with the assistant editor	132
Implementing a delegate method	133
Pre-selecting the default voice	133
Challenge: Make a Data Source	134
8. KVC, KVO, and Bindings	135
Bindings	136
Setting up Thermostat	136
Using bindings	137

- Key-value observing 139
- Making keys observable 140
- Binding other attributes 142
- KVC and Property Accessors 145
- KVC and nil 146
- Debugging Bindings 146
- Using the Debugger 147
 - Using breakpoints 148
 - Stepping through code 149
 - The LLDB console 151
 - Using the debugger to see bindings in action 152
- For the More Curious: Key Paths 153
- For the More Curious: More on Key-Value Observing 154
- For the More Curious: Dependent Keys 155
- Challenge: Convert RGBWell to Use Bindings 156
- 9. NSArrayController 157
 - RaiseMan’s Model Layer 158
 - RaiseMan’s View Layer 160
 - Introducing NSArrayController 160
 - Adding an Array Controller to the XIB 162
 - Binding the Array Controller to the Model 163
 - Binding the Table View’s Content to the Array Controller 164
 - Connecting the Add Employee Button 164
 - Binding the Text Fields to the Table Cell Views 165
 - Formatting the Raise Text Field 167
 - Connecting the Remove Button 169
 - Binding the Table View’s Selection to the Array Controller 169
 - Configuring RaiseMan’s Remove Button 171
 - Sorting in RaiseMan 171
 - How Sorting Works in RaiseMan 174
 - For the More Curious: The caseInsensitiveCompare(·) Method 175
 - For the More Curious: Sorting Without NSArrayController 176
 - For the More Curious: Filtering 177
 - For the More Curious: Using Interface Builder’s View Hierarchy Popover 178
 - Challenge: Sorting Names by Length 180
- 10. Formatters and Validation 181
 - Formatters 181
 - Formatters, programmatically 181
 - Formatters and a control’s objectValue 182
 - Formatters and localization 183
 - Validation with Key-Value Coding 183
 - Adding Key-Value validation to RaiseMan 183
 - For the More Curious: NSValueTransformer 187
- 11. NSUndoManager 189
 - Message Passing and NSInvocation 189
 - How the NSUndoManager Works 190
 - Using NSUndoManager 191
 - Key-Value Coding and To-Many Relationships 192

Adding Undo to RaiseMan	194
Key-Value Observing	195
Using the Context Pointer Defensively	196
Undo for Edits	197
Begin Editing on Insert	199
For the More Curious: Windows and the Undo Manager	201
12. Archiving	203
NSCoder and NSCoder	204
Encoding	204
Decoding	205
The Document Architecture	206
Info.plist and NSDocumentController	207
NSDocument	207
NSWindowController	210
Saving and NSKeyedArchiver	211
Loading and NSKeyedUnarchiver	211
Setting the Extension and Icon for the File Type	212
Application Data and URLs	215
For the More Curious: Preventing Infinite Loops	216
For the More Curious: Creating a Protocol	217
For the More Curious: Automatic Document Saving	218
For the More Curious: Document-Based Applications Without Undo	218
For the More Curious: Universal Type Identifiers	218
13. Basic Core Data	221
Defining the Object Model	221
Configure the Array Controller	223
Add the Views	225
Connections and Bindings	229
How Core Data Works	234
Fetching Objects from the NSManagedObjectContext	235
Persistent Store Types	236
Choosing a Cocoa Persistence Technology	237
Customizing Objects Created by NSArrayController	237
Challenge: Begin Editing on Add	238
Challenge: Implement RaiseMan Using Core Data	238
14. User Defaults	239
NSUserDefaults	239
Adding User Defaults to SpeakLine	240
Create Names for the Defaults	241
Register Factory Defaults for the Preferences	241
Reading the Preferences	242
Reflecting the Preferences in the UI	243
Writing the Preferences to User Defaults	243
Storing the User Defaults	244
What Can Be Stored in NSUserDefaults?	245
Precedence of Types of Defaults	246
What is the User's Defaults Database?	246
For the More Curious: Reading/Writing Defaults from the Command Line	247

- For the More Curious: NSUserDefaultsController 248
- Challenge: Reset Preferences 248
- 15. Alerts and Closures 249
 - NSAlert 249
 - Modals and Sheets 250
 - Completion Handlers and Closures 251
 - Closures and capturing 252
 - Make the User Confirm the Deletion 253
 - For the More Curious: Functional Methods and Minimizing Closure Syntax 256
 - Challenge: Don't Fire Them Quite Yet 256
 - Challenge: Different Messages for Different Situations 257
- 16. Using Notifications 259
 - What Notifications Are 259
 - What Notifications Are Not 259
 - NSNotification 259
 - NSNotificationCenter 259
 - Starting the Chatter Application 261
 - Using Notifications in Chatter 265
 - For the More Curious: Delegates and Notifications 268
 - Challenge: Beep-beep! 268
 - Challenge: Add Usernames 268
 - Challenge: Colored Text 268
 - Challenge: Disabling the Send Button 268
- 17. NSView and Drawing 271
 - Setting Up the Dice Application 271
 - Creating a view subclass 273
 - Views, Rectangles, and Coordinate Systems 274
 - frame 274
 - bounds 276
 - Custom Drawing 276
 - drawRect(·) 277
 - When is my view drawn? 278
 - Graphics contexts and states 278
 - Drawing a die face 279
 - Saving and Restoring the Graphics State 284
 - Cleaning up with Auto Layout 285
 - Drawing Images 286
 - Inspectable properties and designable views 289
 - Drawing images with finer control 290
 - Scroll Views 291
 - Creating Views Programmatically 293
 - For the More Curious: Core Graphics and Quartz 294
 - For the More Curious: Dirty Rects 295
 - For the More Curious: Flipped Views 295
 - Challenge: Gradients 295
 - Challenge: Stroke 295
 - Challenge: Make DieView Configurable from Interface Builder 295
- 18. Mouse Events 297

NSResponder	297
NSEvent	297
Getting Mouse Events	298
Click to Roll	299
Improving Hit Detection	300
Gesture Recognizers	301
Challenge: NSBezierPath-based Hit Testing	303
Challenge: A Drawing App	303
19. Keyboard Events	305
NSResponder	307
NSEvent	307
Adding Keyboard Input to DieView	307
Accept first responder	308
Receive keyboard events	308
Putting the dice in Dice	308
Focus Rings	309
The Key View Loop	310
For the More Curious: Rollovers	310
20. Drawing Text with Attributes	313
NSFont	313
NSAttributedString	314
Drawing Strings and Attributed Strings	316
Drawing Text Die Faces	317
Extensions	318
Getting Your View to Generate PDF Data	318
For the More Curious: NSFontManager	320
Challenge: Color Text as SpeakLine Speaks It	320
21. Pasteboards and Nil-Targeted Actions	323
NSPasteboard	324
Add Cut, Copy, and Paste to Dice	325
Nil-Targeted Actions	326
Looking at the XIB file	328
Menu Item Validation	329
For the More Curious: Which Object Sends the Action Message?	330
For the More Curious: UTIs and the Pasteboard	330
Custom UTIs	330
For the More Curious: Lazy Copying	330
Challenge: Write Multiple Representations	331
Challenge: Menu Item	331
22. Drag-and-Drop	333
Make DieView a Drag Source	333
Starting a drag	334
After the drop	336
Make DieView a Drag Destination	337
registerForDraggedTypes(____)	338
Add highlighting	338
Implement the dragging destination methods	338
For the More Curious: Operation Mask	339

- 23. NSTimer 341
 - NSTimer-based Animation 341
 - How Timers Work 343
 - NSTimer and Strong/Weak References 343
 - For the More Curious: NSRunLoop 343
- 24. Sheets 345
 - Adding a Sheet 345
 - Create the Window Controller 346
 - Set Up the Menu Item 348
 - Lay Out the Interface 349
 - Configuring the Die Views 352
 - Present the Sheet 353
 - Modal Windows 355
 - Encapsulating Presentation APIs 355
 - Challenge: Encapsulate Sheet Presentation 356
 - Challenge: Add Menu Item Validation 357
- 25. Auto Layout 359
 - What is Auto Layout? 359
 - Adding Constraints to RaiseMan 359
 - Constraints from subview to superview 360
 - Constraints between siblings 367
 - Size constraints 368
 - Intrinsic Content Size 370
 - Creating Layout Constraints Programmatically 371
 - Visual Format Language 371
 - Does Not Compute, Part 1: Unsatisfiable Constraints 373
 - Does Not Compute, Part 2: Ambiguous Layout 374
 - For the More Curious: Autoresizing Masks 375
 - Challenge: Add Vertical Constraints 376
 - Challenge: Add Constraints Programmatically 377
- 26. Localization and Bundles 379
 - Different Mechanisms for Localization 379
 - Localizing a XIB File 381
 - Localizing String Literals 385
 - Demystifying NSLocalizedString and genstrings 389
 - Explicit Ordering of Tokens in Format Strings 390
 - NSBundle 390
 - NSBundle’s role in localization 391
 - Loading code from bundles 393
 - For the More Curious: Localization and Plurality 393
 - Challenge: Localizing the Default Name for a Newly Added Employee 394
 - Challenge: Localizing the Undo Action Names 395
- 27. Printing 397
 - Dealing with Pagination 397
 - Adding Printing to RaiseMan 398
 - For the More Curious: Are You Drawing to the Screen? 402
 - Challenge: Add Page Numbers 403
 - Challenge: Persist Page Setup 403

28. Web Services	405
Web Services APIs	405
RanchForecast Project	406
NSURLSession and asynchronous API design	409
NSURLSession, HTTP status codes, and errors	413
Add JSON parsing to ScheduleFetcher	414
Lay out the interface	416
Opening URLs	418
Safely Working with Untyped Data Structures	419
For the More Curious: Parsing XML	420
Challenge: Improve Error Handling	421
Challenge: Add a Spinner	421
Challenge: Parse the XML Courses Feed	421
29. Unit Testing	423
Testing in Xcode	423
Your First Test	425
A Note on Literals in Testing	428
Creating a Consistent Testing Environment	428
Sharing Constants	430
Refactoring for Testing	431
For the More Curious: Access Modifiers	434
For the More Curious: Asynchronous Testing	435
Challenge: Make Course Implement Equatable	436
Challenge: Improve Test Coverage of Web Service Responses	437
Challenge: Test Invalid JSON Dictionary	437
30. View Controllers	439
NSViewController	440
Starting the ViewControl Application	441
Windows, Controllers, and Memory Management	444
Container View Controllers	444
Add a Tab View Controller	445
View Controllers vs. Window Controllers	446
Considerations for OS X 10.9 and Earlier	447
Challenge: SpeakLineViewController	447
Challenge: Programmatic View Controller	447
Challenge: Add a Window Controller	448
31. View Swapping and Custom Container View Controllers	449
View Swapping	449
NerdTabViewController	449
Adding Tab Images	454
Challenge: Boxless NerdTabViewController	455
Challenge: NerdSplitViewController	455
Challenge: Draggable Divider	455
32. Storyboards	457
A New UI for RanchForecast	457
Adding the course list	462
Adding the web view	465
Connecting the Course List Selection with the Web View	466

- Creating the CourseListViewControllerDelegate 468
- Creating the parent view controller 468
- For the More Curious: How is the Storyboard Loaded? 470
- 33. Core Animation 471
 - CALayer 471
 - Scattered 472
 - Implicit Animation and Actions 476
 - More on CALayer 477
 - Challenge: Show Filenames 478
 - Challenge: Reposition Image Layers 478
- 34. Concurrency 479
 - Multithreading 479
 - A Deep Chasm Opens Before You 479
 - Improving Scattered: Time Profiling in Instruments 481
 - Introducing Instruments 481
 - Analyzing output from Instruments 484
 - NSOperationQueue 484
 - Multithreaded Scattered 484
 - Thread synchronization 485
 - For the More Curious: Faster Scattered 486
 - Challenge: An Even Better Scattered 487
- 35. NSTask 489
 - ZIPspector 489
 - Asynchronous Reads 493
 - iPing 494
 - Challenge: .tar and .tgz Files 497
- 36. Distributing Your App 499
 - Build Configurations 499
 - Preprocessor Directives: Using Build Configurations to Change Behavior 500
 - Creating a Release Build 503
 - A Few Words on Installers 505
 - App Sandbox 505
 - Entitlements 505
 - Containers 506
 - Mediated file access and Powerbox 506
 - The Mac App Store 507
 - Receipt Validation 507
 - Local receipt verification 507
 - Server-based verification 508
- 37. Afterword 511
- Index 513

Introduction

If you are developing applications for OS X, or are hoping to do so, this book will be your foundation and will help you understand Cocoa, the set of frameworks for developing applications for OS X. You, the developer, are going to love developing for OS X because Cocoa will enable you to write full-featured applications in a more efficient and elegant manner.

About This Book

This book covers the major design patterns of Cocoa and includes an introduction to the Swift language. It will also get you started with the most commonly-used developer tools: Xcode and Instruments. After reading this book, you will understand these major design patterns which will enable you to understand and use Apple's documentation – a critical part of any Cocoa developer's toolkit – as well as build your own Cocoa applications from scratch.

This book teaches ideas and provides hands-on exercises that show these ideas in action. Each chapter will guide you through the process of building or adding features to an application.

Often, we will ask you to do something and explain the details or theory afterward. If you are confused, read a little more. Usually, the help you seek will be only a paragraph or two away.

Because of the hands-on nature of the book, it is essential that you do the exercises and not just read the words. Doing the exercises will help build the kind of solid understanding that will enable you to develop on your own when you are finished with this book. You will also learn a great deal from making mistakes, reading error messages, and figuring out what went wrong – practical experience you can't get from reading alone. At first, you may want to stick with what we show you, but later in the book when you are more comfortable with the environment, you should feel free to experiment with the exercises and add your own ideas.

Most chapters end with one or two challenge exercises. These exercises are important to do as well. Taking on these challenges gives you the opportunity to test your skills and problem-solve on your own.

You can get help with this book at bignerdranch.com/books, where you will find errata and downloadable solutions for the exercises. You can also post questions and find relevant conversations on the Big Nerd Ranch forums at forums.bignerdranch.com.

We ask that you not use the downloadable solutions as a shortcut for doing the exercises. The act of typing in code has far more impact on your learning than most people realize. By typing the code yourself (and, yes, making mistakes), you will absorb patterns and develop instincts about Cocoa programming, and you will miss out on these benefits if you rely on the solutions or copy and paste the code instead.

There is a lot of code in this book. Through that code, we will introduce you to the idioms of the Cocoa community. Our hope is that by presenting exemplary code, we can help you to become more than a Cocoa developer – a stylish Cocoa developer.

Most of the time, Cocoa fulfills the following promise: Common things are easy, and uncommon things are possible. If you find yourself writing many lines of code to do something rather ordinary, you are probably on the wrong track. There is a popular adage in the community which you should bear in mind: *Don't fight the framework*. Cocoa is opinionated and you will benefit greatly from adapting your way of doing things to its way of doing things.

Prerequisites

This book is written for programmers and assumes that you are familiar with basic programming concepts (like functions, variables, and loops) as well as object-oriented concepts (like classes, objects, and inheritance). If you do not fit this profile, you will find this book tough going. You are not expected to have any experience with Mac programming.

One of the challenges of learning Cocoa programming is learning the Swift language. If you have a basic foundation in programming and know something about objects, you will find learning Swift to be easy. This book includes three chapters to introduce to you to the language. Then you will learn more Swift as you build Cocoa applications throughout the book. If you would prefer a gentler introduction, start with Apple's *The Swift Programming Language*, available in the iBooks store or from developer.apple.com/swift, offers a more gentle introduction. Or, if you can wait until Summer 2015, you can read *Swift Programming: The Big Nerd Ranch Guide* first.

This is a hands-on book and assumes that you have access to OS X and the developer tools. The book requires OS X Yosemite (10.10) or higher. The exercises are written for Xcode 6.3 and Swift 1.2.

We strongly recommend that you join Apple's Mac Developer Program at developer.apple.com/programs. Joining the program gives you access to pre-release versions of Xcode and OS X. These can be very useful when trying to stay ahead of Apple's development curve. In addition, you must be a member of the developer program to distribute your apps on the App Store.

Typographical conventions

To make the book easier to follow, we have used several typographical conventions.

In Swift, class names are always capitalized. In this book, we have also made them appear in a monospaced bold font. In Swift, method names start with a lowercase letter. Here, method names will also appear in a monospaced bold font. For example, you might see “The class `NSWindowController` has the method `showWindow(_ :)`.”

Other literals, including instance variable names that you would see in code, will appear in a regular monospaced font. Also, filenames will appear in this same font. Thus, you might see “In `MyClass.swift`, set the optional `favoriteColor` to `nil`.”

Code samples in this book appear in the regular monospaced font. New portions, which you will need to type yourself, will appear in bold. Code that you should delete is struck-through.

What's new in the fifth edition?

This fifth edition includes technologies introduced in OS X 10.8, 10.9, and 10.10. It is updated for Xcode 6.3 and Swift 1.2. It includes coverage of Swift basics, Auto Layout, unit testing, view controllers and expanded coverage of view swapping, storyboards, modernized localization and web services APIs, JSON parsing, Key-Value Validation, and a strong emphasis on demonstrating best practices for application architecture.

The Story of Cocoa

Once upon a time, two guys named Steve started a company called Apple Computer in their garage. The company grew rapidly, so they hired an experienced executive named John Sculley to be its CEO.

After a few conflicts, John Sculley moved Steve Jobs to a position where he had no control over the company. Steve Jobs left to form another computer company, NeXT Computer.

NeXT hired a small team of brilliant engineers. This small team developed a computer, an operating system, a printer, a factory, and a set of development tools. Each piece was years ahead of competing technologies. Unfortunately, the computer and the printer were commercial failures. In 1993, the factory closed, and NeXT Computer, Inc. became NeXT Software, Inc. The operating system and the development tools continued to sell under the name NeXTSTEP.

NeXTSTEP and OpenStep

NeXTSTEP was very popular with scientists, investment banks, and intelligence agencies. These groups found that NeXTSTEP enabled them to turn their ideas into applications faster than any other technology. In particular, NeXTSTEP had three important features:

a Unix-based operating system

NeXT decided to use Unix as the core of NeXTSTEP. It relied on the source code for BSD Unix from the University of California at Berkeley. Why Unix? Unix crashed much less frequently than Microsoft Windows or Mac OS and came with powerful, reliable networking capabilities.

a powerful window server

A *window server* takes events from the user and forwards them to the applications. The application then sends drawing commands back to the window server to update what the user sees. One of the nifty things about the NeXT window server is that the drawing code that goes to the window server is the same drawing code that would be sent to the printer. Thus, a programmer has to write the drawing code only once, and it can then be used for display on the screen or printing.

If you have used Unix machines before, you are probably familiar with the X window server. The window server for OS X is completely different but fulfills the same function as the X window server: It gets events from the user, forwards them to the applications, and puts data from the applications onto the screen.

an elegant set of libraries and tools

NeXTSTEP came with a set of libraries and tools to enable programmers to deal with the window server in an elegant manner. The libraries were called frameworks. In 1993, the frameworks and tools were revised and renamed OpenStep.

Programmers loved OpenStep because they could experiment more easily with new ideas. In fact, Tim Berners-Lee developed the first web browser and web server on NeXTSTEP using the OpenStep libraries and tools. Securities analysts could code and test new financial models much more quickly. Colleges could develop the applications that made their research possible. We do not know what the intelligence community was using it for, but they bought thousands of copies of OpenStep.

From NeXTSTEP to OS X to iOS

For many years, Apple Computer had been working to develop an operating system with many of the same features as NeXTSTEP. This effort, known as Project Copland, gradually spun out of control,

and Apple finally decided to pull the plug and buy the next version of Mac OS instead. After surveying the existing operating systems, Apple selected NeXTSTEP. Because NeXT was small, Apple simply bought the whole company in December 1996. In 1997, Steve Jobs returned to Apple.

NeXTSTEP became Mac OS X, and OpenStep became Cocoa. In 2001, the first desktop version of Mac OS X was released with several more to follow. In 2012, Apple dropped the “Mac,” and the operating system became known as OS X.

The mutation of NeXTSTEP didn’t stop with OS X. iOS, the operating system for iPhones and iPads, is based on OS X, and iOS’s Cocoa Touch is built on the same foundations as Cocoa. As a developer you will find that your knowledge transfers well between the two: the design patterns are identical, and many of the APIs are very similar if not the same.

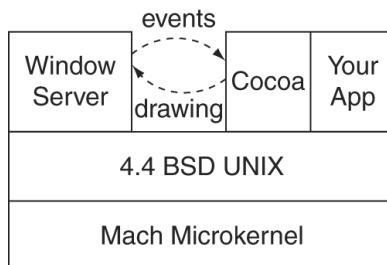
OSX, Unix, and Cocoa

OS X is Unix underneath, and you can get all the standard Unix programs (such as the Apache Web server) on OS X. It is extremely stable, and the user interface is spectacular.

(Apple has made the source code to the Unix part of OS X available under the name Darwin. A community of developers continues to work to improve Darwin. You can learn more about Darwin at www.macosforge.org.)

As shown in Figure 1, the window server and your application are Unix processes. Cocoa is your application’s interface to the window server to receive events and draw to the screen. At the same time it has access to the Unix layer where it can make lower level calls.

Figure 1 Where is Cocoa?



Introducing the Swift Language

Programming in Cocoa was initially done in a language called *Objective-C*. Objective-C is an extension of the C programming language that adds constructs for object-oriented programming. In that respect it bears a superficial resemblance to C++, but the two are extremely different. Unlike C++, Objective-C is weakly typed and extremely powerful. With power comes responsibility: Objective-C also allows programmers to make ridiculous errors.

Over the past several years, Apple’s engineers have gone to heroic lengths to make Objective-C faster and add more modern features, but in order to move forward, a new language was needed, free of the limitations of the past. Swift, developed by a small team led by Chris Lattner, was the answer. Apple introduced Swift in 2014.

Swift maintains the expressiveness of Objective-C while introducing a syntax that is significantly more rich, succinct, and – in the opinion of some – readable. It emphasizes type safety and introduces advanced features such as optionals and generics. Swift is much stricter than Objective-C and will not allow you to make as many ridiculous errors.

Although we will focus on Swift, you can still write Cocoa code in Objective-C, even alongside Swift, compiling the two in the same project.

Most importantly, Swift allows the use of these new features while relying on the same tested, elegant Cocoa frameworks that developers have built upon for years and years.

The Cocoa Frameworks

A *framework* is a collection of classes that are intended to be used together. That is, the classes are compiled together into a reusable library of binary code. Any related resources are put into a directory with the library. The directory is given the extension `.framework`. You can find the built-in frameworks for your machine in `/System/Library/Frameworks`. Cocoa is made up of three frameworks:

- *Foundation*: Every object-oriented programming language needs the standard value, collection, and utility classes. Strings, dates, lists, threads, and timers are in the Foundation framework. All Cocoa apps, from command-line tools to fully-featured GUI apps, use Foundation. Foundation is also available on iOS.
- *AppKit*: All things related to the user interface are in the AppKit framework. These include windows, buttons, text fields, events, and drawing classes. AppKit is built on top of Foundation and is used in practically every graphical application on OS X.
- *Core Data*: Core Data makes it easy to save objects to a file and then reload them into memory. It is a *persistence* framework.

In addition to the three Cocoa frameworks, over a hundred frameworks ship with OS X. The frameworks offer a wide variety of features and functionality. For example, AVFoundation is great for working with audio and video, AddressBook provides an API to the user's contacts (with their permission), and SpriteKit is a full-featured 2D game engine with physics. You can pick and choose from these frameworks to suit the needs of your application. You can also create your own frameworks from the classes that you create. Typically, if a set of classes is used in several applications, you will want to turn them into a framework.

This book will focus on the Cocoa frameworks and especially Foundation and AppKit because they will form the basis of most Cocoa applications that you will write. Once you have mastered these, other frameworks will be easier to understand.

Tools for Cocoa Programming

Xcode is the IDE (integrated development environment) used for Cocoa development. Xcode is available for free on the Mac App Store. Pre-release versions can be downloaded at developer.apple.com/mac. (You will need to join Apple's Mac Developer Program to access these.) We strongly recommend using Xcode 6.3 with Swift 1.2 or later for the exercises in this book.

Xcode tracks all the resources that go into an application: code, images, sounds, and so on. You edit your code in Xcode, and Xcode compiles and launches your application. Xcode can also be used to

invoke and control the debugger. Behind the scenes, swift (Apple’s Swift compiler) will be used to compile your code, and LLDB (Low Level Debugger) will help you find your errors.

Inside Xcode, you will use the *Interface Builder* editor as a GUI builder to lay out windows and add UI elements to those windows. But Interface Builder is more than a simple GUI builder. In Interface Builder, you can create objects and edit their attributes. Most of those objects are UI elements from the AppKit framework such as buttons and text fields, but some will be instances of classes that you create.

You will use *Instruments* to profile your application’s CPU, memory, and filesystem usage. Instruments can also be used to debug memory-management issues. Instruments is built on top of dtrace, which makes it possible to create new instruments.

Some Advice on Learning

All sorts of people come to our class: the bright and the not so bright, the motivated and the lazy, the experienced and the novice. Inevitably, the people who get the most from the class share one characteristic: they remain focused on the topic at hand.

The first trick to maintaining focus is to get enough sleep: ten hours of sleep each night while you are studying new ideas. Before dismissing this idea, try it. You will wake up refreshed and ready to learn. *Caffeine is not a substitute for sleep.*

The second trick is to stop thinking about yourself. While learning something new, many students will think, “Damn, this is hard for me. I wonder if I am stupid.” Because stupidity is such an unthinkable terrible thing in our culture, they will then spend hours constructing arguments to explain why they are intelligent yet having difficulties. The moment you start down this path, you have lost your focus.

Aaron used to have a boss named Rock. Rock earned a degree in astrophysics from Cal Tech, but never had a job that used his knowledge of the heavens. When asked if he regretted getting the degree, he replied, “Actually, my degree in astrophysics has proved to be very valuable. Some things in this world are just hard. When I am struggling with something, I sometimes think ‘Damn, this is hard for me. I wonder if I am stupid,’ and then I remember that I have a degree in astrophysics from Cal Tech; I must not be stupid.”

Before going any further, assure yourself that you are not stupid and that some things are just hard. Armed with this affirmation and a well-rested mind, you are ready to conquer Cocoa.

3

Structures and Classes

At this point you should be somewhat familiar with using Swift's standard types: strings, arrays, enums, etc. It is time to move on to bigger and better things: defining your own types. In this chapter, you will build a simple 2D physics simulation. You will create your own structure and a few classes, and you will learn about the differences between them.

Structures

In Cocoa, structures are typically used to represent groupings of data. For example, there is `NSPoint`, which represents a point in 2D space with an X and a Y value. As your first structure you will create a 2D vector structure.

Create a new playground. From Xcode's File menu, select `New... → Playground`. Name the playground `Physics` and save it with the rest of your projects.

Start by defining the **Vector** structure:

```
import Cocoa

struct Vector {
    var x: Double
    var y: Double
}
```

Much like C structures, Swift structures are composite data types. They are composed of one or more fields, or *properties*, each of which has a specified type. A few lines down, create an instance of **Vector** and access its properties:

```
let gravity = Vector(x: 0.0, y: -9.8) // {x 0, y -9.8000000000000001}
gravity.x // 0
gravity.y // -9.8000000000000001
```

You just used Swift's *automatic initializer* to create an instance of this structure. The automatic initializer has a parameter for each property in the structure. If you were to add a `z` field, this code would cause a compiler error because it lacks a `z` parameter. (Do not worry about the zeros; that is just typical floating point fun.)

You can provide your own initializers, but when you do, the automatic initializer is no longer provided. Go back to **Vector** and add an initializer that takes no parameters and initializes `x` and `y` to `0`.

```
struct Vector {
    var x: Double
    var y: Double

    init() {
        x = 0
        y = 0
    }
}
```

Initializers in Swift use the `init` keyword, followed by the parameter list, and then the body of the initializer. Within the body, the `x` and `y` properties are assigned directly.

An initializer *must* initialize all of the properties of its structure.

As we warned, defining this initializer has caused the automatic one to vanish, causing an error in the playground. You can easily define it manually, however:

```
struct Vector {
    var x: Double
    var y: Double

    init() {
        x = 0
        y = 0
    }

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }
}
```

A Swift programmer would say that this initializer takes two parameters, `x` and `y`, both of type `Double`.

What is `self`? It represents the instance of the type that is being initialized. Using `self.propertyName` is usually unnecessary (you did not use it in `init()`), but because the initializer's parameter names match the names of the properties you must use `self` to tell the compiler that you mean the property and not the parameter.

Before continuing, let's make an improvement. As the **Vector** structure stands, its two initializers have independent code paths. It would be better to have them use one code path by having the parameterless initializer call the initializer which takes both `x` and `y`.

```
struct Vector {
    var x: Double
    var y: Double

    init() {
        x = 0
        y = 0
        self.init(x: 0, y: 0)
    }

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }
}
```


A single code path for initialization is not required for structures, but it is a good habit to get into as you will use it when working with classes.

Instance methods

Methods allow you to add functionality to your data types. In Swift, you can add methods to structures as well as classes (and enums!). *Instance* methods operate within the context of a single instance of the type. Add an instance method for multiplying a vector by a scalar:

```
struct Vector {
    ...

    init(x: Double, y: Double) {
        self.x = x
        self.y = y
    }

    func vectorByAddingVector(vector: Vector) -> Vector {
        return Vector(x: self.x + vector.x,
                      y: self.y + vector.y)
    }
}
```

The `func` keyword in the context of a structure indicates that this is a method. It takes a single parameter of type `Double` and returns an instance of `Vector`.

Try this new method out:

```
let gravity = Vector(x: 0.0, y: -9.8) // {x 0, y -9.8000000000000001}
gravity.x
gravity.y
let twoGs = gravity.vectorByAddingVector(gravity) // {x 0, y -19.6}
```

What is the name of this method? In conversation you would call it `vectorByAddingVector`, but in this text we include parameters, like this: `vectorByAddingVector(_:)`. By default, the first parameter of a method is not named – thus the underscore.

Why not name the first parameter? Because the convention – inherited from Objective-C and Cocoa – is that the base name of the method includes the name of the first parameter, in this case `Vector`. Suppose you added another parameter to that method. What would it look like?

```
func vectorByAddingVector(vector: Vector, numberOfTimes: Int) -> Vector {
    var result = self
    for _ in 0..

```

This method would be called `vectorByAddingVector(_:numberOfTimes:)`. Note that there is a colon for each parameter.

This can lead to verbose method names, but the code actually becomes very readable. No guessing or relying on the IDE to tell you what the third parameter is!

By default, each parameter's internal name is the same as its external name (except the first parameter, that is). In `vectorByAddingVector(_:numberOfTimes:)`, the second parameter is named

numberOfTimes. That is certainly very descriptive, but you might prefer to use a shorter name (like times) within the method. In that case you would explicitly set the internal parameter name like this:

```
func vectorByAddingVector(vector: Vector, numberOfTimes times: Int) -> Vector {
    var result = self
    for _ in 0..
```

The method’s signature has not changed. For those calling it, its name is still **vectorByAddingVector(_:numberOfTimes:)**, but internally you have the satisfaction of using the name you want.

Using self in instance methods

As in initializers, self represents the instance that the method is being called on. As long as there is no conflict with named parameters or local variables, however, it is entirely optional, so we prefer to leave it off. Make this change to **vectorByAddingVector(_:)**.

```
struct Vector {
    ...

    func vectorByAddingVector(vector: Vector) -> Vector {
        return Vector(x: self.x + vector.x,
            y: self.y + vector.y)
        return Vector(x: x + vector.x,
            y: y + vector.y)
    }
}
```

Operator Overloading

By overloading operators you can make your own types work with common (and even uncommon) operators. This ability falls deep beyond the “with great power comes great responsibility” line. However, vectors are a natural and respectable application for this technique.

To define an operator overload you simply add a function that takes the appropriate types. To start with, instead of calling **vectorByAddingVector(_:)**, it would be nice to use the + operator. Overload + and * for adding and scaling vectors, respectively.

```
struct Vector {
    ...

    func +(left: Vector, right: Vector) -> Vector {
        return left.vectorByAddingVector(right)
    }
    func *(left: Vector, right: Double) -> Vector {
        return Vector(x: left.x * right , y: left.y * right)
    }
}
```

Now you can very succinctly manipulate vectors:

```
let twoGs = gravity.vectorByAddingVector(gravity)
let twoGs = gravity + gravity
let twoGsAlso = gravity * 2.0
```

Note that the order of types for binary operators like `*` and `+` is important. In order to write `2.0 * gravity` you will need to implement another operator overload function:

```
func *(left: Double, right: Vector) -> Vector {
    return right * left
}
```

Classes

Now that you have the beginnings of a robust vector type, let's put it to work. Your 2D physics simulation will consist of two classes: **Particle**, which represents a single moving object within the simulation, and **Simulation**, which contains an array of **Particle** instances.

Classes are very similar to structures. They have a lot of the same features: initializers, properties, computed properties, and methods. They have a significant difference, however, which we will discuss once the simulation is up and running.

Start by defining the **Particle** class in your playground. The position is not important, as long as it is above or below (but not inside!) the **Vector** structure. A **Particle** has three **Vector** properties: position, velocity, and acceleration.

```
struct Vector {
    ...
}

class Particle {
    var position: Vector
    var velocity: Vector
    var acceleration: Vector
}
```

Classes and structures differ significantly in terms of initializers. Most noticeably, classes do not have automatic initializers, so you will see a compiler error: Class 'Particle' has no initializers.

Fix this by adding an initializer to **Particle**:

```
class Particle {
    var position: Vector
    var velocity: Vector
    var acceleration: Vector

    init(position: Vector) {
        self.position = position
        self.velocity = Vector()
        self.acceleration = Vector()
    }
}
```

You do not need to provide a parameter for every property in a class like you did in **Vector**'s `init(x:y:)`. You just need to initialize everything. As with initializers for structures, a class's

initializer must initialize all of its properties before returning or performing any other tasks. By requiring this of initializers the Swift compiler guarantees that every instance is fully initialized before it is put to work.

Another approach is to give properties default values:

```
class Particle {  
    var position: Vector  
    var velocity: Vector = Vector()  
    var acceleration: Vector = Vector()  
  
    init(position: Vector) {  
        self.position = position  
    }  
}
```

In a simple case like this, there is not a clear benefit to either approach.

Designated and convenience initializers

Like structures, classes can have multiple initializers. At least one of them will be the *designated initializer*. Remember how you refactored **Vector**'s `init()` to call `init(x:y:)`? A designated initializer is an initializer which other, non-designated initializers – *convenience initializers* – must call. The rule of thumb with designated initializers is that they are typically the one with the most parameters. Most classes will only have one designated initializer.

The `init(position:)` initializer is the **Particle** class's designated initializer. Add a convenience initializer:

```
class Particle {  
    ...  
  
    init(position: Vector) {  
        self.position = position  
        self.velocity = Vector()  
        self.acceleration = Vector()  
    }  
  
    convenience init() {  
        self.init(position: Vector())  
    }  
}
```

There is an exception to these designated initializer rules: required initializers, which you will see in Chapter 12.

Add an instance method

A particle has a position, velocity, and acceleration. It should also know a little about particle dynamics – specifically, how to update its position and velocity over time. Add an instance method, `tick(_:)`, to perform these calculations.

```

class Particle {
    ...

    convenience init() {
        self.init(position: Vector())
    }

    func tick(dt: NSTimeInterval) {
        velocity = velocity + acceleration * dt
        position = position + velocity * dt
        position.y = max(0, position.y)
    }
}

```

The `tick(_:)` method takes an `NSTimeInterval` parameter, `dt`, the number of seconds to simulate. `NSTimeInterval` is an alias for `Double`.

Below the definition of `Particle`, define the `Simulation` class, which will have an array of `Particle` objects and its own `tick(_:)` method:

```

class Particle {
    ...
}

class Simulation {

    var particles: [Particle] = []
    var time: NSTimeInterval = 0.0

    func addParticle(particle: Particle) {
        particles.append(particle)
    }

    func tick(dt: NSTimeInterval) {
        for particle in particles {
            particle.acceleration = particle.acceleration + gravity
            particle.tick(dt)
            particle.acceleration = Vector()
        }
        time += dt
    }
}

```

The `Simulation` class has no initializers defined since all of its properties have default values. The `for-in` loop iterates over the contents of the `particles` property. The `tick(_:)` method applies constant acceleration due to gravity to each of the particles before simulating them for the time interval.

Before you warm up the simulator and add a particle, add a line to evaluate `particle.position.y`. You will use this shortly with the playground's Value History. Additionally, add some code to remove particles once they drop below `y = 0`:

```
class Simulation {
  ...

  func tick(dt: NSTimeInterval) {
    for particle in particles {
      particle.acceleration = particle.acceleration + gravity
      particle.tick(dt)
      particle.acceleration = Vector()
      particle.position.y
    }
    time += dt
    particles = particles.filter { particle in
      let live = particle.position.y > 0.0
      if !live {
        println("Particle terminated at time \(self.time)")
      }
      return live
    }
  }
}
```

The last chunk of code filters the `particles` array, removing any particles that have fallen to the ground. This is a *closure*, and it is OK if you do not understand it at this point. You will learn more about closures in Chapter 15.

Now you are ready to run the simulator. Create an instance of the simulator and a particle, add the particle to the simulation, and see what happens.

```
class Simulation {
  ...
}

let simulation = Simulation()

let ball = Particle()
ball.acceleration = Vector(x: 0, y: 100)
simulation.addParticle(ball)

while simulation.particles.count > 0 && simulation.time < 500 {
  simulation.tick(1.0)
}
```

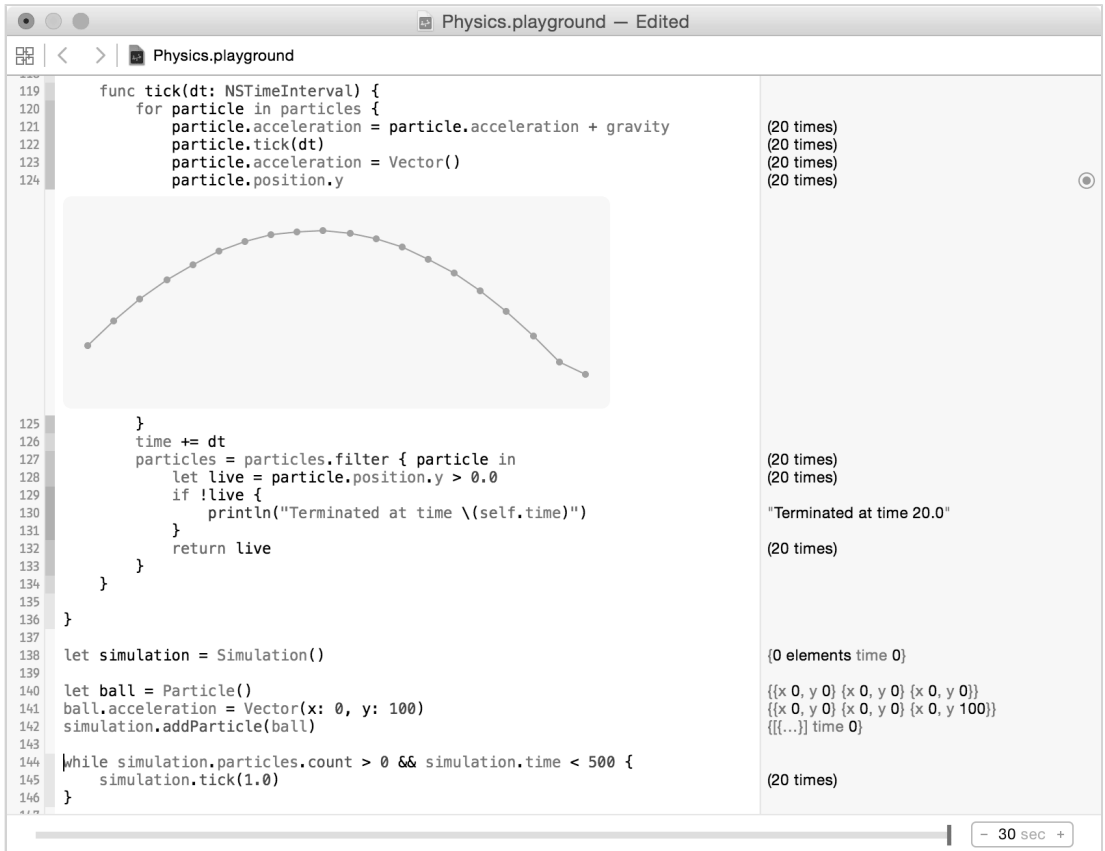
You should see the playground tally up (20 times) on a number of lines. If the playground runs the simulation continuously, you can stop it by commenting out the `while` loop. Select the three lines and hit Command-/ to toggle the comment marks:

```
// while simulation.particles.count > 0 && simulation.time < 500 {
//   simulation.tick(1.0)
// }
```

Double-check your code against the listings above, in particular the lines that filter the `particles` array and the line that increments `time`.

Once you have the simulation running as expected, click the Variables View circle in the playground sidebar on the line that reads `particle.position.y`, as shown in Figure 3.1. A graph will appear, showing the Y values of the particle over time. The X axis on this graph represents iterations over time and not the X coordinate of the particle.

Figure 3.1 Graph data history of `particle.position.y`



Inheritance

Suppose you wanted to simulate a particle that had different behavior than the `Particle` class you have already implemented: a rocket that propels itself with thrust over a certain period of time. Since `Particle` already knows about physics, it would be natural to extend and modify its behavior through subclassing.

Define the `Rocket` class as a subclass of `Particle`.

```
class Rocket: Particle {  
  
    let thrust: Double  
    var thrustTimeRemaining: NSTimeInterval  
    let direction = Vector(x: 0, y: 1)  
  
    convenience init(thrust: Double, thrustTime: NSTimeInterval) {  
        self.init(position: Vector(), thrust: thrust, thrustTime: thrustTime)  
    }  
  
    init(position: Vector, thrust: Double, thrustTime: NSTimeInterval) {  
        self.thrust = thrust  
        self.thrustTimeRemaining = thrustTime  
        super.init(position: position)  
    }  
  
}
```

The `thrust` property represents the magnitude of the rocket's thrust. `thrustTimeRemaining` is the number of seconds that the thrust will be applied for. `direction` is the direction that the thrust will be applied in.

Take a minute to go through the initializers you just typed in. Which is the designated initializer? (Remember the rule of thumb about designated initializers?)

In order to guarantee that a class's properties are initialized, initializers are only inherited if a subclass does not add any properties needing initialization. Thus, **Rocket** provides its own initializers and calls the superclass's designated initializer.

Next you will override the `tick(_:)` method, which will do a little math to calculate the acceleration due to thrust and apply it before calling the superclass's – **Particle**'s – `tick(_:)` method.

```
class Rocket: Particle {  
    ...  
  
    init(position: Vector, thrust: Double, thrustTime: NSTimeInterval) {  
        self.thrust = thrust  
        self.thrustTimeRemaining = thrustTime  
        super.init(position: position)  
    }  
  
    override func tick(dt: NSTimeInterval) {  
        if thrustTimeRemaining > 0.0 {  
            let thrustTime = min(dt, thrustTimeRemaining)  
            let thrustToApply = thrust * thrustTime  
            let thrustForce = direction * thrustToApply  
            acceleration = acceleration + thrustForce  
            thrustTimeRemaining -= thrustTime  
        }  
        super.tick(dt)  
    }  
  
}
```

Finally, create an instance of **Rocket** and add it to the simulation in place of the ball:

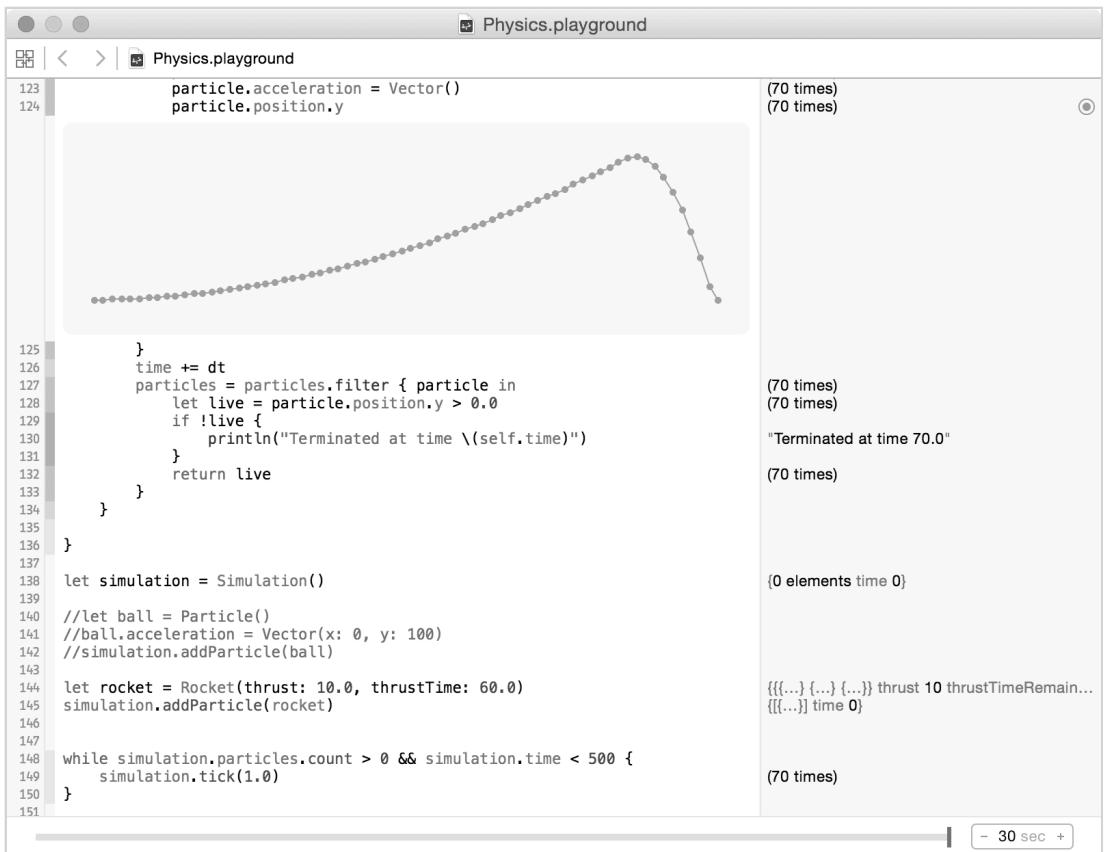

```
let simulation = Simulation()

let ball = Particle()
ball.acceleration = Vector(x: 0, y: 100)
simulation.addParticle(ball)
// let ball = Particle()
// ball.acceleration = Vector(x: 0, y: 100)
// simulation.addParticle(ball)

let rocket = Rocket(thrust: 10.0, thrustTime: 60.0)
simulation.addParticle(rocket)
```

The simulation will run for 70 “seconds” with these parameters. The Value History shows quite a different profile! (Figure 3.2)

Figure 3.2 The rocket’s Y position over time



Note that inheritance is one key differentiator between classes and structures: structures do not support inheritance.

Computed Properties

It is frequently useful to find a vector’s length or magnitude. You could do this by adding a function returning a Double:

```
struct Vector {
    ...

    func length() -> Double {
        return sqrt(x*x + y*y)
    }
}
```

However, it is much more natural to think of this as a read-only property. In Swift the general term for this is *computed property*, which is in contrast to the *stored properties* you have been using so far. A read-only computed property version of `length` would look like this:

```
struct Vector {
    ...

    var length: Double {
        get {
            return sqrt(x*x + y*y)
        }
    }
}
```

This read-only computed property pattern (called a “getter”) is so common, in fact, that Swift provides a shorthand means of expressing it. Add this to **Vector**:

```
struct Vector {
    ...

    var length: Double {
        return sqrt(x*x + y*y)
    }
}
```

At other times it is useful to have a getter *and* setter for a computed property. This tends to be used to alias other properties or to transform a value before it is used elsewhere. For example, you could abstract the setting of the `textField` from the `RandomPassword` with a computed property:

```
class MainWindowController: NSWindowController {

    @IBOutlet weak var textField: NSTextField!

    var generatedPassword: String {
        set {
            textField.stringValue = newValue
        }
        get {
            return textField.stringValue
        }
    }

    ...

    @IBAction func generatePassword(sender: AnyObject) {
        let length = 8
        generatedPassword = generateRandomString(length)
    }
}
```

Computed properties do not have any storage associated with them. If you need to store a value, you must create a separate stored property for it.

Reference and Value Types

Structures and classes are far more alike in Swift than they are in most languages. However, there is one major difference in how they operate: classes are *reference types*; structures, enums, and tuples are *value types*.

What does it mean to be a value type? For one thing, a value type is always treated as *a single value*, even if it is composed of several individual values via its properties.

In practical terms, this means that when a value type is assigned or passed as a parameter, a copy is made. The following code demonstrates the effect with the **Vector** structure:

```
var vector0 = Vector(x: 0, y: 0)      vector0 = {x 0, y 0}
var vector1 = vector0                vector0 = {x 0, y 0}, vector1 = {x 0, y 0}
vector0.x = 1                        vector0 = {x 1, y 0}, vector1 = {x 0, y 0}
```

When `vector0` is assigned to `vector1`, the entire value of `vector0` is copied into the memory represented by `vector1`. When `vector0` is changed, `vector1` is unaffected.

Contrast this behavior with classes, which, again, are reference types:

```
let ball0 = Particle()              ball0      -> Particle: {x 0, y 0} ...
let ball1 = ball0                   ball0, ball1 -> Particle: {x 0, y 0} ...
ball0.particle.x = 1                ball0, ball1 -> Particle: {x 1, y 0} ...
```

Even though you assign `ball0` to `ball1`, there is still only one **Particle** instance in existence; no copies are made. The `ball0` constant is a *reference* to the instance, and when `ball0` is assigned to `ball1`, `ball1` is then a reference to the same instance.

(A reference is similar to a pointer in C-based languages. However, a pointer stores the actual memory address of the object and you can access that address directly. A Swift reference does not provide direct access to the address of the object being referenced.)

There is another reference type. Functions are types so that a function can be passed in to other functions as a defined parameter, or even assigned to a property. This is the basis of closures, which you saw briefly earlier in this chapter, and which you will see again in Chapter 15.

Implications of reference and value types

Passing by reference instead of by value has two main implications. The first has to do with mutability. With a value type the code manipulating that value has complete control over it.

Reference types, however, are much different: any part of the software that has a reference to an instance of a reference type can change it. In object oriented programming this can be desirable, but in complex (and especially multithreaded) software it is a liability. An object being changed “behind your back” can cause crashes at best and strange or difficult-to-debug behavior at worst.

Swift constants help further illustrate this point. A constant value type cannot be changed once it is defined, period:

```
let vector: Vector
if someCondition {
    vector = Vector(x: 0, y: 1)
}
else {
    vector = Vector(x: 0, y: -1)
}
vector.x = 1 // Error: Immutable value 'vector' may only be initialized once
```

A constant reference provides no such protection. Only the reference itself is constant.

```
let cannonball = Particle()
cannonball.velocity = Vector(x: 100, y: 5) // No error!
```

Note that constants within a class or structure are constant. The **Rocket** class's thrust property, defined with `let` and given an initial value in the initializer, cannot be changed:

```
let rocket = Rocket(thrust: 10.0, thrustTime: 60.0)
rocket.thrust = 3 // Error: Cannot assign 'thrust' in 'rocket'
```

Choosing between reference and value types

How does a Cocoa programmer decide whether to use a structure or a class for their new type? In order to answer that you will need to know how the type will be used.

The vast majority of Cocoa is built on reference types: subclasses of the Objective-C base class **NSObject**, which provides a lot of important functionality for Cocoa. As such, large portions of your app, namely the controller and view layers, will also need to descend from **NSObject**.

The model layer is where the answer gets fuzzy. Model-oriented Cocoa technologies such as KVC, KVO, and Bindings also depend on **NSObject**, so many app's models will also. For other apps whose models are perhaps more heavy on logic and computation, and less about binding to the UI, you are free to choose the Swift type that makes the most sense for the problem you are trying to solve. Do you want the shared mutable state provided by reference types, or do you prefer the safety of value types? Both have their advantages and costs.

Cocoa, with its deep roots in MVC and Objective-C, will always rely heavily on reference types. In comparison to Objective-C, however, Swift takes great strides in making value types powerful. As a Cocoa programmer, both will be important tools in your belt.

Making Types Printable

If you use a **Vector** value in string interpolation, you will not get a very pleasing result:

```
println("Gravity is \ (gravity).") // Gravity is __lldb_expr_247.Vector."
```

You can improve this by conforming to the `Printable` protocol, which looks like this:

```
protocol Printable {
    var description: String { get }
}
```

We will cover protocols in more detail in Chapter 6, but the short version is that a protocol defines a set of properties or methods. In order to conform to a protocol, your type must implement the required properties and methods.

To conform to `Printable`, you must implement a read-only computed property called `description` to return a `String`. Start by declaring that `Vector` conforms to `Printable`:

```
struct Vector {
struct Vector: Printable {
    var x: Double
    var y: Double
```

Finally, implement `description`:

```
struct Vector: Printable {
    ...
    var description: String {
        return "\(\(x), \(\(y))"
    }
}
```

Your `Vectors` now look great in strings:

```
println("Gravity is \(\(gravity).")           "Gravity is (0.0, -9.8)."
```

Swift and Objective-C

Although you will write your classes in Swift, the classes in the Cocoa frameworks are written in Objective-C. Swift was designed to work seamlessly with Objective-C classes. While you can write Cocoa apps in pure Swift, without a line of Objective-C, it is important to have a basic understanding of how Objective-C works.

Objective-C methods (which are only available on classes, not structures) are not called like functions or like Swift methods. Instead of calling a method on an object, Objective-C sends the object a *message*.

A message consists of a receiver, selector, and any parameters. The *selector* is the name of the method you want executed. The *receiver* is the object that you want to execute that method. Here is an example of sending a message in Objective-C:

```
NSString *newString;
newString = [originalString stringByReplacingOccurrencesOfString: @"Mavericks"
                                                         withString: @"Yosemite"];
```

In this example, the receiver is `originalString`, an instance of `NSString`, and the selector is `stringByReplacingOccurrencesOfString:withString:`. The parameters are the two `NSString` literals.

Note that the selector in the message is “the name of the method.” It is not the method itself or even a reference to it. You can think of a selector as a glorified string.

Objective-C classes know how to receive a message, match the selector with a method of the same name, and execute the method. Or they can do something else with the selector, like forward it in a message to another class. Relying on selectors and message-passing is relatively unique among languages in modern use, and its dynamic nature made the powerful design patterns of Cocoa, and later iOS, possible.

Calling a method is a cut-and-dried process. Either the object implements the method or it does not, and this can be determined at compile time. Passing a message, on the other hand, is dynamic. At runtime, the object is asked, “Does your class implement a method with this name?” If yes, the method with that name is executed. If no, the message is run up the inheritance hierarchy: The superclass is asked, “Do you have a method with this name?” If that class does not have the method, then its superclass is asked, and so on. If the message reaches **NSObject** at the top of the hierarchy, and **NSObject** says, “No, I do not have a method with that name,” then an exception occurs, and your app will halt.

You are developing in Swift, which means that you are not writing message sends in code; you are calling methods. These Swift methods have to be named in such a way that the Swift compiler can turn a method call into a message send when the receiver is an Objective-C object.

If you were to write the above message send in Swift, it would look like this:

```
let newString = originalString.stringByReplacingOccurrencesOfString("Mavericks"  
                                                                    withString: "Yosemite")
```

Remember, the Swift method has two parameters in the parameter list, but only one has a name. This is why you see methods named in this text and in Apple’s documentation with underscores where you expect a parameter name. For example, this method is listed as **stringByReplacingOccurrencesOfString(_:withString:)**.

Working with Foundation Types

In Objective-C, a number of familiar types are implemented as classes as part of the Foundation framework: **NSString**, **NSNumber**, **NSArray**, **NSDictionary**, and **NSSet**. Because Cocoa was built for Objective-C, you will often run into these classes when working with the Cocoa APIs. The good news is that Apple has made transitioning between the Swift and Foundation (Objective-C) counterparts relatively painless. They are *toll-free bridged*, meaning that there is minimal computational cost in converting between the two.

Basic bridging

Swift types are automatically bridged to their Foundation counterparts:

```
let string = "Howdy"  
let objcString: NSString = string
```

The reverse is not true, however:

```
let swiftString: String = objcString // Error!
```

Instead, you must explicitly cast it using `as`:

```
let swiftString: String = objcString as String // Ok!
```

Another class that you may see is **NSNumber**. Because Foundation collections can only store objects, in order to store numbers they must be represented by an object. **NSNumber** is the class that Objective-C programmers use for this task. Swift numbers also bridge easily with **NSNumber**:

```
let objcNumber: NSNumber = 3  
let swiftNumber = objcNumber as Int
```

Bridging with collections

Bridging with collections is similar, but a wrinkle emerges when casting from a Foundation array back to Swift:

```
let array = [1, 2, 4, 8]
let objcArray: NSArray = array           // So far so good...
let swiftArray: [Int] = objcArray as [Int] // Error!
```

You may be surprised to learn that Foundation collections can hold any kind of object – that is, the collection’s contents do not have to be of the same type! You will see the Swift type `AnyObject` used with these collections, like this: `[AnyObject]`. (If you are familiar with Objective-C, `AnyObject` has the same meaning as `id`.)

The solutions to this problem are similar to unwrapping optionals: there are safe and unsafe paths. The unsafe path is to use `as!`, the forced cast operator:

```
let swiftArray: [Int] = objcArray as! [Int]
```

As with forced unwrapping, if the type cannot be cast successfully your app will crash. If you are certain that the type is correct, such as when the value is coming from a known API, this is a reasonable assumption to make.

If you are not so certain, you should use the optional type casting operator `as?`, which will evaluate to `nil` if the values cannot be safely cast:

```
if let swiftArray: [Int] = objcArray as? [Int] {
    ...
}
```

This situation is most commonly seen with Cocoa APIs using **NSDictionary**: it is typical for the keys to all be **NSStrings**, but the types of the values commonly differ depending on the key. We will further discuss how to handle these untyped collections safely in Chapter 28.

Suppose you were working with an Objective-C class that supplied a dictionary. When Swift imports the class, it does a basic level of conversion, but it does not know what type the method actually returns, so it is shown as `[NSObject : AnyObject]!`:

```
class NSProcessInfo: NSObject {
    ...
    var environment: [NSObject : AnyObject]! { get }
    ...
}
```

To work with this API you first need to know the actual types contained in the dictionary, which can usually be found in the documentation. You will then need to safely cast the result to Swift types:

```
let processInfo = NSProcessInfo()
if let environment = processInfo.environment as? [String : String] {
    if let path: String = environment["PATH"] {
        println("Path is: \(path)")
    }
}
```

It is important to remember that Swift strings and collections are value types and the Foundation types are all reference types. While Swift's compiler can enforce the constant-ness of an array, with `NSArray` the same array object may be referenced by many parts of an application.

The Foundation classes we have discussed so far are all *immutable*, meaning that they cannot be changed – equivalent to being defined with Swift's `let`. Each of them has a mutable subclass: `NSMutableArray`, `NSMutableString`, and so forth. This has less of an impact on Swift code, but it is important to watch out for if you are working with a significant body of Objective-C code. Because it is a reference type, an instance of `NSMutableArray` could be changed by any code that has a reference to it.

Runtime Errors

Despite your best efforts, things will sometimes go wrong while your app is running. In Cocoa, these errors fall into two categories.

Programmer errors are situations that should never happen, which means that they are the result of, well, a mistake you made. (We say they should never happen... but we have made plenty of these.) Examples include not meeting the precondition of a method (the index was not within the array's bounds), performing an illegal operation (such as force-casting incorrectly or force-unwrapping a `nil`), or sending a message to an Objective-C object that does not understand it.

Swift alerts you to programmer errors by trapping, which results in stopping the program. Cocoa APIs use Objective-C exceptions. A trap is typically accompanied by a `fatal` error line in the console, while exceptions have much longer output showing the full stack. Note that Swift does not presently support exceptions.

Recoverable errors, on the other hand, are errors that your application can check for, deal with, and move on from. Examples include being unable to contact a remote server, errors parsing data, or lacking hardware capabilities.

Recoverable errors will be communicated to your code through the return values of methods (such as a `nil` return). For more sophisticated APIs, especially those involving I/O, an `NSError` object will be used. You will learn about `NSError` in Chapter 12.

You can code defensively and check preconditions in your own code using `assert()` and `fatalError()`. For example:

```
let condition: Bool = ...
assert(condition, "Condition was not met")
```

`fatalError()` is useful in methods that are declared to return a value. The Swift compiler requires that all code paths return a value – unless you call a *noreturn* function like `fatalError()`:

```
func openFortuneCookie() -> String {
    if let cookie = cookie {
        return cookie.fortune
    }
    else {
        fatalError("Must have cookie!")
        // No return statement
    }
}
```


More Exploring of Apple's Swift Documentation

Your homework for this chapter is to browse through the Classes and Structures, Properties, Methods, and Initialization sections of Apple's *The Swift Programming Language* guide. You should also tackle the challenge exercises given below.

Challenge: Safe Landing

Your investors have pointed out that rockets are expensive and letting them plummet to the ground does not enhance reusability. Enhance the **Rocket** class to deploy a parachute in order to slow its descent once it is descending (i.e., shows negative velocity on the Y axis) and reaches a certain altitude.

Challenge: Vector Angle

Your **Vector** structure should be able to report its angle in radians. Add a read-only, computed property to it called `angle`.

The angle of a vector can be expressed in Swift as:

```
atan2(y, x)
```

This page intentionally left blank

Index

Symbols

(_:), meaning of, 43
.icns file, 212
.lproj files (localization), 379
.tar files, 497
.tgz files, 497
.xcdatamodeld (Core Data), 221
// MARK:, 105
@IBAction, 20
@IBDesignable, 290
@IBInspectable, 289
@IBOutlet, 18
@NSApplicationMain, 116

A

accents, typing, 384
acceptsFirstResponder (**NSResponder**), 307
acceptsMouseMovedEvents (**NSWindow**), 310
access modifiers, 427, 434, 435
actions
 (see also connections, controls, **NSControl**, outlets)
 and AnyObject, 81
 connecting in Interface Builder, 20
 defined, 17
 feature of **NSControl**, 78
 and menu items, 263, 264
 as messages, 78, 82
 methods for, 19, 81
 nil-targeted, 326-328
 and **NSApplication**, 330
 setting programmatically, 96
actions (**CALayer**), 476
addChildViewController(_:)
 (**NSViewController**), 454
addObserver(_:selector:name:object:)
 (**NSNotificationCenter**), 260
addSubview(_:)
 (**NSView**), 449
alerts, 249-251
alpha values (**NSColor**), 91
ambiguous layouts (Auto Layout), 374, 375
animations, 471-477
 and timers, 341
AnyObject, 81, 454
API Reference, 88
App Store (distribution), 507-509
AppDelegate
 about, 115, 116
 role, 5
 and window controllers, 14, 24-26
append(_:), 34
AppKit (framework), xxi, 71, 276
Apple Developer Programs, xviii
application architecture
 basic, 14, 26
 document-based, 158, 206-210
 master/detail, 444
 with multiple view controllers, 466
 and MVC, 4-6
 single-window, 70-77, 101-103
 and view controllers, 439, 440, 444-446
 and window controllers, 26, 74-77, 446
application bundles, 390, 489
applications
 (see also application architecture, projects)
 App Store, using, 507-509
 build configurations for, 499
 containers for, 506
 copy protection for, 507
 custom file extensions for, 212-214
 custom file icons for, 212-214
 distributing, 504, 507-509
 document-based, 158
 entitlements of, 505
 and event loop, 116
 exporting, 504
 installers for, 505
 launching, 116
 lifecycle methods, 115
 localizing, 379-389
 locations for data, 215, 216
 mediated file access, 506
 and multiple threads, 479
 packaging, 505
 printing from, 397-403
 and release builds, 503, 504
 sandboxing, 505-507
 storage for, 215, 216
 and system resources, 505
 unit testing, 423
ARC (Automatic Reference Counting), 61, 65, 68
 (see also memory management)

archivedDataWithRootObject(_:), 211
archiving
 about, 203
 build targets, 503
 decoding, 205, 206
 and document architecture, 206
 encoding, 204, 205
 loading objects, 211
 NSCoder, 204-206
 NSData, 211
 NSKeyedArchiver, 211
 NSKeyedUnarchiver, 211
 preventing infinite loops in, 216, 217
 saving objects, 211
 vs. Core Data, 237
 XIB files, 14
ARRepeat (NSEvent), 307
arrangedObjects (NSArrayController), 161, 169, 173
array controllers
 (see also **NSArrayController**)
 about, 160-164
 customizing, 237
 filtering with, 177, 178
 immediate fetching, 224
 labeling in Interface Builder, 224
 and model abstractions, 162
 and **NSManagedObjectContext**, 223
 sorting with, 171-175
 as target of controls, 164, 165
arrays
 about, 31-33
 append(_:), 34
 count, 34
 filtering, 177
 memory management of, 63
 and **NSArray**, 57
 reverse(), 34
 subscripting, 32
 and traps, 33
as, 56
assert(), 58
assertions (unit testing), 424-427, 434
assistant editor, 132
associated values (enums), 411
astrophysics degrees, xxii
attributes (Core Data), 221-224
attributes (views), 11-13

attributes inspector (Xcode), 11, 83
Auto Layout
 (see also constraints)
 adding constraints, 365
 ambiguous layouts, 374, 375
 vs. autosizing masks, 375, 376
 clip view warning, 124
 described, 359
 intrinsic content size, 370
 and **NSBox**, 449, 455
 unsatisfiable constraints, 373, 374
 Visual Format Language, 371-373
 visualizeConstraints(_:), 375
 with right-to-left languages, 371
auto-complete (Xcode), 111
automatic document saving, 218
automatic initializers, 41
autosizing masks, 375, 376
availableTypeFromArray(_:)
(**NSPasteboardItem**), 325

B

background threads, 412, 479, 484
Base.lproj, 387
beginCriticalSheet(_:completion...) (**NSWindow**), 345
beginDraggingSessionWithItems(...) (**NSView**), 334
beginSheet(_:completionHandler:)
(**NSWindow**), 345, 353
beginSheetModalForWindow(_:completion...)
(**NSAlert**), 251
bindings
 array controllers, 160-164
 benefits of, 136
 with Core Data, 221
 creating, 138, 139
 creating programmatically, 154
 debugging, 146, 152, 153
 and KVC/KVO, 136, 139, 142
 and **NSObject**, 159, 408
 patterns for, 231
 for table data, 130
 and value transformers, 187
 when to use, 145
bindings inspector (Xcode), 130
blocking

- (see also multithreading)
 - CPU-bound, 484
 - I/O-bound, 484
 - and modal windows, 251, 355
 - boldSystemFontOfSize(_:) (NSFont)**, 313
 - Bool, 31
 - boolean types, 31
 - boolForKey(_:) (NSUserDefaults)**, 240
 - bounds (**NSView**), 276
 - breakpoint navigator, 151
 - breakpoints, 148, 151
 - bridging, 56-58
 - build actions, 500
 - build configurations
 - changing app behavior with, 500-503
 - debug, 98, 499
 - debugging symbols in, 499
 - finding, 499
 - and Instruments, 481
 - and preprocessor directives, 500-503
 - release, 98, 499
 - setting flags in, 500-503
 - specifying, 500
 - build targets, 423, 503
 - bundles
 - application, 212, 390, 489
 - described, 390, 391
 - identifiers for, 216, 246
 - and localization, 380, 391, 392
 - main, 390
 - and strings files, 391
 - buttons
 - disabling, 108
 - in Interface Builder, 10
 - radio, 97
 - recessed, 143
 - titles for, 11
- ## C
- CAAnimation**, 472
 - CABasicAnimation**, 477
 - CAGradientLayer**, 478
 - CALayer**
 - about, 471
 - actions, 476
 - delegate of, 477
 - described, 472
 - subclasses, 478
 - CALayerDelegate**, 477
 - canvas (Interface Builder), 8
 - CAOpenGLLayer**, 478
 - capture lists, 252
 - caseInsensitiveCompare(_:)**, 175
 - CAShapeLayer**, 478
 - casting, 56-58
 - categories, 318
 - CATextLayer**, 478
 - CATransaction**, 472, 477
 - cell-based tables, 126
 - cells
 - and controls, 80
 - history in Cocoa, 80, 126
 - in table views, 126-128
 - CGFloat, 91, 92
 - CGRect
 - contains(_:)**, 300
 - characters (**NSEvent**), 307
 - checkboxes (**NSButton**), 97
 - Clang Static Analyzer, 68
 - class methods, 121
 - classes
 - (see also *individual class names*, initializers, methods, objects, properties, types)
 - about, 4
 - creating new, 6-8
 - defining, 45-48
 - extending, 318
 - and inheritance, 49-51
 - initializing, 46
 - making @IBDesignable, 290
 - prefixes for, 162
 - in product modules, 162
 - reference pages for, 88
 - vs. structures, 53, 54
 - clearContents() (NSPasteboard)**, 324
 - clickCount (**NSEvent**), 298
 - clip views, 125
 - closures, 53, 251-256
 - Cocoa
 - API reference, 88
 - classes in, 54, 55, 88
 - documentation, 88
 - frameworks in, xxi, 71
 - history of, xviii-xx
 - Cocoa Touch (framework), 512

- CocoaHeads, 512
- code snippet library, 101
- code snippets, 101-103
- color (**NSColorWell**), 89
- color wells, 87
- com.pkware.zip-archive, 490
- completion handlers
 - about, 251, 252
 - with asynchronous API, 409
 - implementing, 410-418
 - testing, 435, 436
- computed properties
 - about, 51-53
 - and KVC, 145
 - storage for, 146
- concludeDragOperation(_:)**
(NSDraggingDestination), 337
- concurrency, 479-481, 484-487
- conditionals
 - if-else, 105
 - if-let, 36
 - switch, 38
- connections (in Interface Builder), 17-22
 - (see also actions, outlets)
 - with assistant editor, 132
 - to File's Owner, 76
- connections inspector, 21
- connections panel, 18
- console
 - exceptions in, 184
 - importance in debugging, 98
 - LLDB (debugger), 151, 152
 - viewing in playground, 36
 - viewing in project, 82
- constants, 29, 30
- constraints (Auto Layout)
 - (see also Auto Layout)
 - adding in Interface Builder, 360-368
 - adding programmatically, 377
 - and ambiguous layouts, 374, 375
 - animating, 371
 - between siblings, 367, 368
 - creating in Interface Builder, 359, 364
 - creating programmatically, 371
 - debugging, 373-375
 - for positioning views, 359
 - priorities of, 363
 - size constraints, 368
 - subview-superview, 360-366
 - types of, 359
 - unsatisfiable, 373, 374
- containers (for applications), 506
- containers (view controllers), 445
- contains(_:)** (CGRect), 300
- content (**NSArrayController**), 161
- Content Array (array controllers), 161, 162
- content views, 4, 73
- contexts (graphics), 278, 284
- continuous (**NSControl**), 82
- controller layer (MVC), 5
 - (see also view controllers, window controllers)
- controllers
 - (see also Model-View-Controller)
- controls
 - (see also actions, **NSControl**)
 - about, 78, 79
 - and action messages, 78, 82
 - array controllers as targets, 164, 165
 - and cells, 80
 - creating programmatically, 372
 - enabling/disabling, 88
 - formatting, 182
 - making continuous, 82
 - outlets to, 93
 - and target-action, 78
- convertPoint(_:fromView:)** (NSView), 301
- convertPoint(_:toView:)** (NSView), 301
- copy protection, 507
- copying-and-pasting (implementing), 323-326
- Core Animation, 471-478
- Core Data
 - .xcdatamodeld, 221
 - attributes, 221-224
 - benefits of, 221, 237
 - with bindings, 221
 - data model inspector, 236
 - and data set size, 237
 - defining object model, 221-223
 - entities, 221-224
 - explained, 234
 - faulting, 237
 - fetch requests, 235
 - NSManagedObject**, 221
 - NSManagedObjectContext**, 223, 235, 237
 - NSManagedObjectModel**, 221, 235
 - NSPersistentDocument**, 235

- pros and cons, 237
 - relationships, 221
 - and SQLite, 236, 237
 - store types, 236
 - vs. archiving, 237
 - Core Graphics (framework), 276, 294
 - count (arrays), 34
 - createDirectoryAtURL(_:withIntermed...)**
(**NSFileManager**), 216
 - currentContextDrawingToScreen()**
(**NSGraphicsContext**), 402
 - cutting-and-pasting (implementing), 323-326
- ## D
- Dalrymple, Mark, 512
 - Darwin (Unix), xx
 - data model inspector (Core Data), 236
 - data sources (run loops), 493
 - data sources (table views), 120, 128-131
 - dataForType(_:)** (**NSPasteboardItem**), 325
 - dataOfType(_:error:)** (**NSDocument**), 208
 - dataSource (**NSTableView**), 120
 - dataSource (property)
 - exposed as outlet, 128
 - setting in Interface Builder, 128
 - dataWithPDFInsideRect(_:)** (**NSView**), 318
 - date formatters, 181, 416
 - date pickers, 227
 - debug builds, 98, 499
 - DEBUG compile-time value, 501
 - debug navigator, 148
 - debugger bar, 149
 - debugging
 - (see also debugging tools, errors, exceptions)
 - Auto Layout constraints, 373-375
 - bindings, 146, 152, 153
 - exceptions, 151
 - hints, 97, 98
 - stack trace, 148
 - stepping through methods, 149, 150
 - symbols, 499
 - with zombie objects, 98
 - debugging tools
 - breakpoints, 148, 151
 - debug navigator, 148
 - debugger, 147-151
 - LLDB (debugger) console, 151, 152
 - stack trace, 148, 149
 - variables view, 149
 - decodeBoolForKey(_:)** (**NSCoder**), 205
 - decodeDoubleForKey(_:)** (**NSCoder**), 205
 - decodeFloatForKey(_:)** (**NSCoder**), 205
 - decodeIntForKey(_:)** (**NSCoder**), 205
 - decodeObjectForKey(_:)** (**NSCoder**), 205
 - default: (switch statement), 38
 - defaultCenter()** (**NSNotificationCenter**), 260
 - defaults, 239, 240
 - delegate (property), 111-115
 - (see also delegate methods, delegation)
 - exposed as outlet, 131
 - setting in Interface Builder, 114, 131
 - delegate methods
 - (see also delegate, delegation)
 - and notifications, 115
 - optional, 110, 116
 - required, 110
 - types of, 113
 - using auto-complete for, 111
 - delegation
 - (see also delegate, delegate methods)
 - about, 110, 111
 - classes using, 115
 - errors in implementing, 114
 - NSWindowDelegate**, 113
 - and protocols, 110
 - steps in implementing, 111
 - vs. subclassing, 109, 110
 - and table views, 120
 - and web services, 410
 - dependent keys, 155
 - developer programs, xviii
 - dictionaries
 - about, 31, 32
 - accessing, 36
 - and **NSDictionary**, 57
 - subscripting, 36
 - didChangeValueForKey(_:)**, 141
 - didSet (property observer), 108
 - directories
 - (see also bundles)
 - (see also bundles, files)
 - .lproj, 379
 - application, 215, 216
 - as file wrappers, 207
 - localization, 379, 387

- project source, 386
- dirty rects, 278, 295
- dismissWithModalResponse(_:)**, 354
- distributing (applications), 504, 507-509
- DMG (disk image), 505
- dock (Interface Builder), 8
- Document** (template-created class), 158
- document architecture, 206-210
- document controllers, 207
- document outline (Interface Builder), 8, 127
- document-based applications, 158
 - and printing, 397
 - and responder chain, 328
- documentation
 - for Cocoa classes, 88-91
 - for protocols, 112
 - for Swift, 39
- documents
 - (see also document architecture, document controllers, files, **NSDocument**)
 - automatic saving of, 218
 - extensions for, 212-214
 - icons for, 212-214
 - loading, 209
 - and loading NIB files, 209
 - printing from, 397-403
 - saving, 207
- DOM parsing, 420
- Double, 31
- doubleValue, 78
- drag-and-drop, 333-339
- draggingEntered(_:)**
(NSDraggingDestination), 337, 339
- draggingExited(_:)** (NSDraggingDestination), 337
- draggingSession(_:endedAtPoint:operati...)**
(NSDraggingSource), 336
- draggingSession(_:sourceOperationM...)**, 334
- draggingUpdated(_:)**
(NSDraggingDestination), 337
- drawAtPoint(_:)** (NSAttributedString), 316
- drawAtPoint(_:withAttributes:)** (NSString), 316
- drawFocusRingMask()** (NSView), 309
- drawing
 - (see also animations, views)
 - and dirty rects, 278, 295
 - frameworks for, 294

- and graphics contexts, 278, 284
- images, 286-290
- and layers, 471
- PDF data, 318
- and points, 275
- printer vs. screen, 402
- views, 276-279
- drawInRect(_:)** (NSImage), 288
- drawInRect(_:fromRect:op...)** (NSImage), 290
- drawInRect(_:withAttributes:)** (NSString), 316
- drawLayer(_:inContext:)**, 477
- drawRect(_:)** (NSView), 277-279
- dynamic, 140, 141

E

- enabled, 88
- encodeBool(_:forKey:)** (NSCoder), 204
- encodeConditionalObject(_:forKey:)** (NSCoder), 216
- encodeDouble(_:forKey:)** (NSCoder), 204
- encodeFloat(_:forKey:)** (NSCoder), 204
- encodeInt(_:forKey:)** (NSCoder), 204
- encodeObject(_:forKey:)** (NSCoder), 204
- encodeWithCoder(_:)** (NSCoding), 204, 205
- endSheet(_:returnCode:)** (NSWindow), 345, 354
- entities (Core Data), 221-224
- entitlements (application), 505
- enumerate()**, 37
- enums (enumerations)
 - with associated values, 411
 - defined, 38
 - instance methods in, 43
 - nested, 411
 - and raw values, 39
 - and switch statements, 38
- errors
 - (see also debugging, exceptions, **NSError**)
 - Auto Layout, 374, 375
 - auto-saving, 199
 - with bindings, 146, 147
 - and completion handlers, 410
 - and enums, 411
 - in event-handling, 184
 - exceptions, 58
 - HTTP codes, 413, 414
 - in delegation, 114

- in playgrounds, 30
- with KVC, 183
- runtime, 58
- traps, 33, 58
- with untyped data, 419
- XCTFail()**, 434
- event loop, 116
- events
 - (see also mouse events)
 - errors in handling, 184
 - in event loop, 116
 - keyboard, 305-310
 - mouse (see mouse events)
- exception breakpoints, 151
- exceptions, 58, 151
 - (see also errors)
- expressions
 - evaluating with LLDB, 151
 - and string interpolation, 37
- extensions (of a class), 318

F

- factory defaults, 239
- fallthrough (switch statement), 38
- fatalError()**, 58
- faulting (Core Data), 237
- fetch requests (Core Data), 235
- file handles, 492, 493, 496
- file wrappers, 207
- File's Owner, 76
- files
 - (see also directories, documents)
 - copying, 462
 - custom extensions for, 212-214
 - custom icons for, 212-214
 - formats for pasting, 323
 - in project, 3
 - loading, 209
 - saving, 207
- fileWrapperOfType(_:error:) (NSDocument)**, 208
- fill() (NSBezierPath)**, 277
- filter()**, 177, 256
- filtering (array controllers), 177, 178
- find()**, 134
- first responder, 305-308, 328
 - (see also **NSResponder**, responder chain)

- flagsChanged(_:) (NSResponder)**, 307
- flipped views, 295
- Float, 31
- floatForKey(_:) (NSUserDefaults)**, 240
- floating-point types, 31, 33, 424
- floatValue, 78
- focus rings, 309
- fonts, 313, 314, 320
- for-in, 37
- forced unwrapping (of optionals), 35
- formatter (**NSControl**), 182
- formatters
 - about, 181-183
 - and controls, 182
 - date, 181
 - interaction with locale, 183
 - number, 167, 169, 182
 - vs. KVC validation, 183
 - writing custom, 183
- forwardInvocation(_:) (NSInvocation)**, 189
- Foundation (framework), xxi, 56, 159
- frame (**NSView**), 274-276
- frameworks
 - AppKit, xxi, 71
 - Cocoa, xxi, 71
 - Cocoa Touch, 512
 - Core Data, 221, 235, 237
 - Core Graphics, 276, 294
 - defined, xxi
 - documentation for, 88
 - for drawing, 294
 - Foundation, xxi, 56, 159
 - importing, 71, 159
 - Quartz, 294
 - shipped with OS X, xxi
 - XCTest, 424, 425
- func, 43
- functional programming, 256
- functions
 - (see also closures, initializers, methods)
 - for functional programming, 256
 - as types, 53

G

- generalPasteboard() (NSPasteboard)**, 324
- genstrings** (localization), 385, 387, 389
- gesture recognizers, 301, 302

Grand Central Dispatch (GCD) (multithreading), 486
graphics contexts, 278, 284
 drawing to screen, 402
graphics states, 278, 284
groups (project files), 3

H

Hashable, 31
helper objects, 110
hierarchies, view, 5
hit testing/detection, 300, 303
HTTP, 405
HTTP status codes, 413, 414

I

.icns file, 212
identity inspector, 76
if-else, 105
if-let, 36
image wells, 227, 228
images, 286-290
implicit animation, 476
implicitly unwrapped optionals, 95
importing frameworks, 71, 159
importing modules, 427
Info.plist, 207
init (keyword), 42
init(coder:) (NSCoding), 204-206
init(...) (see initializers)
initWithFirstResponder (NSWindow), 265, 310
initializers
 about, 33
 automatic, 41
 chaining, 42
 for classes, 45-47
 designated, 46, 206
 inheriting, 50, 206
 parameters, 42
 and properties, 42
 for standard types, 33, 34
 for structures, 41, 42
 writing, 41
inspectors
 attributes, 11
 bindings, 130
 connection, 21

 data model, 236
 identity, 76
installers (application), 505
instances, 33
Instruments, xxii, 481-484
Int, 31
integer types, 31
integerForKey(_:) (NSUserDefaults), 240
integerValue, 78
Interface Builder
 adding menu items, 319
 adding views in, 9-13
 assistant editor, 132
 connecting dataSource in, 128
 connecting delegate in, 114
 connecting objects in, 17-22, 76
 connections panel, 18
 copying and pasting in, 85, 86
 creating bindings in, 138, 139
 designing custom classes in, 290
 File's Owner, 76
 inspecting custom properties in, 289
 navigating, 8
 overview, xxii
 placeholders, 76
 view hierarchy popover, 178-180
internal (access modifier), 427, 434
interpretKeyEvents(_:) (NSResponder), 308
intrinsicContentSize, 286
invalidate() (NSTimer), 342
isEmpty (strings), 34

J

Jobs, Steve, xviii
JSON parsing, 414
jump bar (Xcode), 105

K

key paths, 153, 154
key view loop, 310
key windows, 305, 306, 327
key-value coding (see KVC)
key-value observing (see KVO)
key-value pairs
 (see also KVC, KVO)
 in dictionaries, 31
 in strings files (localization), 390

- key-value validation, 181, 183-186
 - keyboard events, 305-310
 - keyCode (**NSEvent**), 307
 - keyDown(_:)** (**NSResponder**), 307
 - keyPathsForValuesAffectingFullName()**, 155
 - keys
 - (see also KVC, KVO)
 - dependent, 155
 - in dictionaries, 31
 - in key-value coding, 135
 - making observable, 140
 - observing, 139
 - keyUp(_:)** (**NSResponder**), 307
 - Knight Rider, 109
 - knowsPageRange(_:)** (**NSView**), 398
 - KVC (key-value coding)
 - (see also key-value validation, KVO)
 - about, 135
 - and proxy objects, 193
 - and to-many relationships, 192
 - and bindings, 136
 - and computed properties, 145
 - empty string exceptions, 183
 - in undo, 192
 - method naming conventions, 193
 - methods, 135, 146
 - and `nil`, 146, 183-186
 - and predicates, 177
 - and property accessors, 145
 - and type safety, 142
 - validate(_:error:)**, 185
 - validation for, 183-186
 - KVO (key-value observing)
 - about, 139
 - and bindings, 139, 142
 - compliance, 140
 - dependent keys, 155
 - in undo, 195
 - methods, 141
 - and Swift, 140
- L**
- labelFontOfSize(_:)** (**NSFont**), 313
 - labels, 80
 - layer (**NSView**), 473
 - layers
 - animating, 476
 - creating, 473
 - drawing, 471
 - lazy copying (pasteboard), 330, 331
 - length (**NSRange**), 314
 - let, 29, 30
 - level indicators, 227, 228
 - libraries
 - code snippet, 101
 - object, 9, 10
 - lineToPoint()** (**NSBezierPath**), 278
 - literal values, 32, 33
 - in testing, 428
 - LLDB (debugger), xxi, 151, 152
 - (see also debugging)
 - loading documents, 209
 - loading NIB files, 75-77
 - loading objects, 211
 - loadView()** (**NSViewController**), 440, 465
 - `Localizable.strings`, 385, 387, 388
 - localization
 - adding localizations to projects, 381
 - and **NSBundle**, 391, 392
 - base directory, 387
 - described, 379
 - directories, 387
 - and formatters, 183
 - genstrings**, 387
 - global resources, 392
 - images, 379
 - language-specific resources, 392
 - and NIB files, 380
 - NSLocalizedString**, 385-390
 - of XIB files, 380, 382-385
 - and plurals, 393, 394
 - region-specific resources, 392
 - replacing string literals, 385
 - of resources (non-XIB), 379
 - and strings files, 379, 380, 382-390
 - token ordering in strings, 390
 - ways to achieve, 379, 380
 - location (**NSRange**), 314
 - `locationInWindow` (**NSEvent**), 297, 300
 - loops
 - event, 116
 - examining in Value History, 37
 - for, 37
 - for-in, 37
 - run, 343

in Swift, 36
.lproj files (localization), 379

M

Mac App Store (distribution), 507-509

Mac Developer Program, xviii

main bundle, 390

main thread, 479

managed object model (Core Data), 221

managedObjectContext (**NSArrayController**), 223

map(), 256

// MARK:, 105

master/detail interfaces, 444

maxValue (**NSSlider**), 83

mediated file access, 506

memory management

and arrays, 63

in closures, 252, 253

and delegate, 112

and Instruments, 483

manual reference counting, 68

need for, 61

and notifications, 267

of windows, 444

for reference types, 61

reference counting, 61-65

strong reference cycles, 65-67

strong references, 65

and timers, 343

unowned references, 67

for value types, 61

weak references, 65, 67

and zombie objects, 98

menu items

creating in Interface Builder, 319

disabling, 329

hooking up, 263, 264

and keyboard events, 310

and **NSDocument**, 207, 209

and **NSDocumentController**, 207

state, 329

targets of, 326-328

validating, 329

messageFontSize(_:) (**NSFont**), 313

messages

(see also methods)

action, 78

explained, 55, 56

and **NSInvocation**, 189

methods

(see also *individual method names*, initializers,

messages, properties)

(**_:**), meaning of, 43

about, 43

action, 81

application lifecycle, 115

class, 121

in classes, 46

data source, 121, 129

defined, 27, 34

delegate, 268

in enums, 43

in extensions, 318

KVC, 135

KVO, 141

naming conventions, 43

optional, 110, 116

parameters, 43

in protocols, 110

required, 110

spelling errors in, 114

static, 27

stepping through, 149, 150

in structures, 43

minValue (**NSSlider**), 83

modal alerts, 249, 250

modal windows, 355

model key path, 139

model layer (MVC), 5

binding to array controller, 163, 164

encapsulating in web services, 409

and table views, 120

Model-View-Controller (MVC)

(see also application architecture, controller

layer, model layer, view layer)

defined, 4-6

and web services, 407, 409

modifierFlags (**NSEvent**), 297, 307

modules, 427

modules (product), 162

mouse events

(see also events, first responder, **NSEvent**)

checking click count, 298

double-clicks, 298

- gesture recognizers, 301, 302
- handler methods, 297
- hit testing, 300
- mouseDragged(_:)**, 335
- rollovers, 310, 311
- mouseDragged(_:) (NSResponder)**, 335
- mouseEntered(_:) (NSResponder)**, 310
- mouseExited(_:) (NSResponder)**, 310
- mouseMoved(_:) (NSResponder)**, 310
- moveToPoint()** (**NSBezierPath**), 278
- multithreading
 - background threads, 412, 479, 484
 - complexities in using, 479
 - considerations with mutable **Array**, 486
 - Grand Central Dispatch (GCD), 486
 - main thread, 479
 - mutex, 486
 - NSOperationQueue**, 484, 485
 - NSRecursiveLock**, 486
 - operation queues, 412
 - race conditions, 480, 481
 - thread synchronization, 485, 486
 - threads, 479
 - and web services, 409
- mutability, 53
- mutex (multithreading), 486
- MVC (see Model-View-Controller)

N

- navigators (Xcode)
 - about, 3
 - breakpoint, 151
 - debug, 148
 - project, 3
- needsDisplay (NSView)**, 278
- nested types, 411
- nextKeyView (NSView)**, 310
- nextResponder (NSResponder)**, 327
- NeXTSTEP, xviii-xx, 5
- NIB files
 - (see also XIB files)
 - defined, 14
 - loading, 75-77
 - and loading documents, 209
 - and localization, 380
 - names of, 24-26
 - naming conventions for, 72
- nil-targeted actions, 326-328
- notifications
 - about, 259
 - adding observers for, 260
 - constants for, 265
 - in delegate methods, 115, 268
 - and memory management, 267
 - observing, 259-261
 - posting, 260, 261, 266
 - registering for, 260, 266
 - removing observers of, 260, 267
 - responding to, 267
 - unregistering for, 260, 267
 - and web services, 410
- NS** prefix, 5
- NSAlert**, 249-251
- NSApplication**
 - (see also **AppDelegate**, applications)
 - about, 115, 116
 - in responder chain, 327
 - sendAction(_:to:from:)**, 330
- NSApplicationDelegate**, 115
- @NSApplicationMain**, 116
- NSArray**, 57
 - (see also arrays)
- NSArrayController**
 - (see also array controllers)
 - arrangedObjects**, 161, 169, 173
 - content, 161
 - managedObjectContext**, 223
 - selectionIndexes**, 161, 169
 - subclassing for custom objects, 237
- NSAttributedString**, 314-317
 - (see also **NSString**, strings)
- NSBezierPath**, 276
- NSBox**, 449, 452, 455
- NSBundle**, 390-393
- NSButton**, 10
 - (see also buttons)
- NSCell**, 80
- NSClipView**, 125
- NSCoder**, 204-206
- NSCoding** (protocol), 204-206
- NSColor**, 90, 91, 95
- NSColorWell**, 89
- NSComparisonResult**, 176
- NSControl**
 - (see also controls)

- action, 78
 - continuous, 82
 - enabled, 88
 - formatter, 182
 - inheritance hierarchy of, 78
 - setting target/action programmatically, 96
 - target, 78, 82
 - value properties, 78
- NSData**, 208, 209, 211
- NSDateFormatter**, 181
- NSDatePicker**, 227
- NSDictionary**, 57
 - (see also dictionaries)
- NSDistributedNotificationCenter**, 259
- NSDocument**
 - (see also documents, **NSDocumentController**)
 - about, 158, 207-209
 - and archiving, 206
 - dataOfType(_:error:)**, 208
 - fileWrapperOfType(_:error:)**, 208
 - NSDocumentChangeType, 218
 - printOperationWithSettings(_:error:)**, 397, 401
 - readFromData(_:ofType:error:)**, 209
 - readFromFileWrapper(_:ofType:error:)**, 209
 - readFromURL(_:ofType:error:)**, 209
 - in responder chain, 327
 - updateChangeCount(_:)**, 218
 - windowControllerDidLoadNib(_:)**, 209
 - writeToURL(_:ofType:error:)**, 208
- NSDocumentController**, 207
 - (see also **NSDocument**)
 - in responder chain, 327
- NSDraggingDestination (protocol), 337
- NSDraggingInfo (protocol), 337
- NSDraggingItem**, 335
- NSDraggingSource (protocol), 334
- NSDragOperation, 333
- NSError**, 208, 209
 - (see also errors)
- NSErrorPointer**, 208, 209
- NSEvent**
 - (see also events)
 - and keyboard events, 307
 - and mouse events, 297
- NSFetchRequest**, 235
- NSFileHandle**, 492, 493, 496
- NSFileManager**, 215
- NSFont**, 313, 314
- NSFontAttributeName (**NSAttributedString**), 315
- NSFontManager**, 320
- NSForegroundColorAttributeName (**NSAttributedString**), 315
- NSFormatter**, 183
- NSGradient**, 295, 338
- NSGraphicsContext**, 284, 402
- NSImage**
 - drawing on, 335
 - drawInRect(_:)**, 288
 - drawInRect(_:fromRect:op...)**, 290
- NSImageView**, 227, 228
- NSInvocation**, 189
- NSKeyedArchiver**, 204, 211
- NSKeyedUnarchiver**, 211
- NSLevelIndicator**, 227, 228
- NSLocalizedString** (localization), 385-390
- NSMakeRange()**, 314
- NSManagedObject** (Core Data), 221
- NSManagedObjectContext** (Core Data), 221, 223, 235, 237
- NSManagedObjectModel** (Core Data), 221, 235
- NSMatrix**, 97
- NSMenuItem** (see menu items)
- NSMutableAttributedString**, 314
- NSMutableURLRequest**, 406
- NSNotification**, 259
 - (see also notifications)
- NSNotificationCenter**
 - about, 259
 - commonly-used methods, 260
 - and memory management, 267
- NSNumber**, 56
- NSNumberFormatter**, 181, 182
- NSObject**
 - base Cocoa class, 54
 - required for bindings, 408
- NSOperationQueue** (multithreading), 412, 484, 485
- NSParagraphStyleAttributeName (**NSAttributedString**), 315
- NSPasteboard**, 323-325, 330, 331
- NSPasteboardItem**, 324, 325, 330, 331
- NSPasteboardReading (protocol), 324
- NSPasteboardWriting (protocol), 324

- NSPersistentDocument** (Core Data), 221, 235
- NSPipe**, 489, 492, 496, 497
- NSPoint**, 274
- NSPredicate**, 177, 235
- NSPrintInfo**, 403
- NSPrintOperation**, 397
- NSRange**, 314
- NSRect**, 274
- NSRecursiveLock** (multithreading), 486
- NSResponder**
 - about, 78
 - first responder methods, 307, 308
 - mouse event handler methods, 297, 298
 - mouseDragged(_:)**, 335
 - nextResponder, 327
 - responder chain, 327, 328
- NSRunLoop**, 343
- NSSavePanel**, 319
- NSScroller**, 125
- NSScrollView**, 291
- NSShadow**, 284
- NSShadowAttributeName** (**NSAttributedString**), 315
- NSSlider**, 79, 83
- NSSliderCell**, 80
- NSSortDescriptor**, 171-176
- NSSpeechSynthesizer**
 - implementing, 106, 107
 - voices available for, 121
- NSSpeechSynthesizerDelegate** (protocol), 110
- NSSplitView**, 445
- NSSplitViewController**, 444, 445
- NSStackView**, 453
- NSString**
 - (see also **NSAttributedString**, strings)
 - drawAtPoint(_:withAttributes:)**, 316
 - drawInRect(_:withAttributes:)**, 316
 - sizeWithAttributes(_:)**, 316
 - and String, 56
- NSSuperscriptAttributeName** (**NSAttributedString**), 315
- NSTableColumn**, 125
- NSTableHeaderView**, 125
- NSTableView**
 - (see also **NSTableViewDataSource**, **NSTableViewDelegate**, table views)
 - dataSource, 120
 - reloadData()**, 134
 - sortDescriptors, 173
- NSTableViewDataSource**
 - implementing, 128-131
 - numberOfRowsInTableView(_:)**, 121, 129
 - tableView(_:objectValueForTa...)**, 121, 129
- NSTableViewDelegate**, 131-133
- NSTabView**, 445
- NSTabViewController**, 445, 446, 449-455
- NSTask**, 489, 491, 496
- NSTextField** (see text fields)
- NSTextFieldCell**, 80
- NSTimer**, 341-343
- NSUnderlineColorAttributeName** (**NSAttributedString**), 315
- NSUnderlineStyleAttributeName** (**NSAttributedString**), 315
- NSUndoManager**, 191, 192, 201
- NSURL**, 406
- NSURLRequest**, 406
- NSURLSession**, 406, 413, 414
- NSURLSessionDataTask**, 409
- NSURLSessionTask**, 406, 409
- NSUserDefaults**, 239, 240
- NSValueTransformer**, 187
- NSView**
 - (see also **NSViewController**, views)
 - addSubview(_:)**, 449
 - beginDraggingSessionWithItems(...)**, 334
 - bounds, 276
 - convertPoint(_:fromView:)**, 301
 - convertPoint(_:toView:)**, 301
 - custom subclasses of, 271, 273-279
 - dataWithPDFInsideRect(_:)**, 318
 - drawFocusRingMask()**, 309
 - drawRect(_:)**, 277-279
 - flipped, 295
 - frame, 274-276
 - intrinsicContentSize, 286
 - knowsPageRange(_:)**, 398
 - layer, 473
 - needsDisplay, 278
 - nextKeyView, 310
 - _NSViewBackingLayer**, 478
 - rectForPage(_:)**, 398
 - registerForDraggedTypes(_:)**, 337, 338
 - removeFromSuperview()**, 449
 - setNeedsDisplayInRect(_:)**, 295
 - viewDidMoveToWindow()**, 310

- wantsLayer, 473
 - _NSViewBackingLayer**, 478
 - NSViewController**
 - (see also **NSView**, view controllers)
 - about, 440
 - loadView()**, 440, 465
 - nibName, 442
 - in responder chain, 327
 - view, 440
 - viewLoaded, 451
 - NSWindow**
 - (see also **NSWindowController**, **NSWindowDelegate**, windows)
 - beginCriticalSheet(_:completion:)**, 345
 - beginSheet(_:completionHandler:)**, 345, 353
 - endSheet(_:returnCode:)**, 345, 354
 - firstResponder, 305-308
 - initialFirstResponder, 310
 - sheet methods, 345
 - sheetParent, 354
 - visualizeConstraints(_:)**, 375
 - NSWindowController**
 - (see also **NSWindow**, window controllers)
 - and **NSDocument**, 210
 - in responder chain, 327
 - window, 75-77
 - windowDidLoad()**, 93, 94
 - windowNibName, 24, 25, 75
 - NSWindowDelegate** (protocol), 112, 113
 - NSWorkspace**, 418
 - NSXMLDocument**, 420
 - NSXMLNode**, 420
 - number formatters, 167, 169, 182
 - numberOfRowsInTableView(_:)**, 121, 129
- O**
- object library (Xcode), 9, 10
 - object-oriented programming, 4
 - objectForKey(_:)** (**NSUserDefaults**), 240
 - Objective-C
 - (see also Cocoa, KVC, KVO)
 - about, xx
 - and bindings, 408
 - dynamic, 140-142
 - in documentation, 91
 - messages, 55, 56, 189
 - reference types, 54
 - and Swift, 55
 - objects
 - (see also classes, methods, properties)
 - about, 4
 - and memory management, 61-65
 - objectValue(NSControl)**
 - about, 78
 - binding to, 130
 - formatting, 182
 - and table data, 127
 - observers (notifications), 259-261
 - observers (property), 108
 - OpenStep, xix
 - operation masks (drag-and-drop), 339
 - operation queues, 412, 484, 485
 - operators, overloading, 44
 - optional (protocol methods), 110, 116
 - optional binding, 36
 - optional chaining, 117
 - optionals
 - about, 34
 - as?, 57
 - and casting, 57
 - chaining, 117
 - and dictionary subscripting, 36
 - forced unwrapping of, 35
 - if-let, 36
 - implicitly unwrapped, 95
 - and optional binding, 36
 - syntax for, 34
 - unwrapping, 35
 - origin (NSRect), 274
 - OS X
 - (see also Cocoa)
 - frameworks for, xxi
 - history of, xix
 - as Unix-based, xx
 - outlets
 - (see also properties)
 - assistant editor, connecting with, 132
 - connecting in Interface Builder, 18, 19, 132
 - creating in code, 17
 - dataSource, 128
 - defined, 17
 - delegate, 114, 131
 - as implicitly unwrapped optionals, 95
 - as weak references, 67

when to use, 93
overloading operators, 44

P

packaging (applications), 505
pagination (printing multiple pages), 397
parameter names, 43
pasteboards, 323-325, 330, 331
pasting (implementing) (see pasteboards)
PDFs, generating, 318
performance issues, 482, 484
performDragOperation(_:)
(NSDraggingDestination), 337
persistence (see archiving, Core Data)
pipes, 492, 496, 497
placeholder text, 105
placeholders, 76
playgrounds (Xcode), 28-30
 errors in, 30
 Value History, 37
 viewing console in, 36
pointers, 53
points (in drawing), 275
postNotification(_:)
(NSNotificationCenter), 260
postNotificationName(_:object:)
(NSNotificationCenter), 260
Powerbox, 506
predicates, 177, 235
preferences (user), 239, 240
prepareForDragOperation(_:)
(NSDraggingDestination), 337
preprocessor directives, 500-503
pressure (NSEvent), 298
Printable (protocol), 54
printing, 397-403
printOperationWithSettings(_:error:)
(NSDocument), 397, 401
private (access modifier), 434
product modules, 162
programmer errors, 58
programming
 functional, 256
 object-oriented, 4
project navigator, 3
projects
 (see also applications)

 copying files into, 462
 creating, 1-3
 source directories of, 386
 targets in, 423
properties
 (see also methods, outlets)
 in attributes inspector, 83
 computed, 51-53, 146
 default values, 46
 defined, 34
 didSet, 108
 and extensions, 318
 initializing, 42, 46
 making @IBInspectable, 289
 shadowing, 17
 stored, 52
 willSet, 108
property observers, 108
propertyListForType(_:) (NSPasteboardItem),
325
protocols
 CALayerDelegate, 477
 conforming to, 110, 111
 creating, 217, 454
 defining roles with, 110
 documentation for, 112
 header files for, 114
 NSApplicationDelegate, 115
 NSCoding, 204-206
 NSDraggingDestination, 337
 NSDraggingInfo, 337
 NSDraggingSource, 334
 NSPasteboardReading, 324
 NSPasteboardWriting, 324
 NSSpeechSynthesizerDelegate, 110
 NSTableViewDataSource, 128-131
 NSTableViewDelegate, 131-133
 NSWindowDelegate, 113
 optional methods in, 110, 116
 Printable, 54
 reference pages for, 112
 required methods in, 110
public (access modifier), 428, 434

Q

Quartz (framework), 294
Quick Help (Xcode), 30

R

race conditions (multithreading), 480, 481
 radio buttons (**UIButton**), 97
 Range, 37
 rawValue (enums), 39
readFromData(_:ofType:error:) (**NSDocument**), 209
readFromFileWrapper(_:ofType:error:) (**NSDocument**), 209
readFromURL(_:ofType:error:), 491
readFromURL(_:ofType:error:) (**NSDocument**), 209
readObjects(_:options:) (**NSPasteboard**), 324
 receipt validation, 507-509
 receivers, 55
 recoverable errors, 58
rectForPage(_:) (**NSView**), 398
 redo stack, 190
reduce(), 256
 reference counting, 61-65, 68
 reference types, 53, 54
 references, strong, 65
 references, unowned, 67
 references, weak, 65
registerDefaults(_:) (**NSUserDefaults**), 240
registerForDraggedTypes(_:), 337
registerForDraggedTypes(_:) (**NSView**), 337, 338
 relationships (Core Data), 221
 release builds, 98, 499, 503, 504
reloadData() (**NSTableView**), 134
removeFromSuperview() (**NSView**), 449
removeObjectForKey(_:) (**NSUserDefaults**), 240
removeObserver(_:) (**NSNotificationCenter**), 260
 representedObject, 454
resignFirstResponder() (**NSResponder**), 307
 resources
 (see also bundles)
 application access to, 505
 for future learning, 511, 512
 in bundles, 390, 391
 localizing, 379
 responder chain, 327, 328
restoreGraphicsState() (**NSGraphicsContext**), 284

reverse(), 34
 RoboCop, 109
 run loops, 343, 493
runModal() (**NSAlert**), 250
runModalForWindow(_:) (**NSApplication**), 355
 runtime errors, 58

S

sandboxing (applications), 505-507
saveGraphicsState() (**NSGraphicsContext**), 284
 saving documents, 207, 218
 saving objects, 211
 SAX parsing, 421
 scenes (storyboards), 457, 463, 470
scheduledTimerWithTimeInterval (**NSTimer**), 342
 Scheme Editor, 500, 502, 503
 scroll views, 125, 291
 scrollers, 125
 Sculley, John, xviii
 segues (storyboards), 457, 459
 selectionIndexes (**NSArrayController**), 161, 169
 selectors, 55
selectTab(_:), 451
selectTabAtIndex(_:), 451
 self
 in closures, 252
 in initializers, 42
 in instance methods, 44
 and property names, 17
sendAction(_:to:from:) (**NSApplication**), 330
 sender (action methods), 81
setBool(_:forKey:) (**NSUserDefaults**), 240
setData(_:forType:) (**NSPasteboardItem**), 325
setFloat(_:forKey:) (**NSUserDefaults**), 240
setInteger(_:forKey:) (**NSUserDefaults**), 240
setNeedsDisplayInRect(_:) (**NSView**), 295
setNilValueForKey(_:), 146
setObject(_:forKey:) (**NSUserDefaults**), 240
setPropertyList(_:forType:) (**NSPasteboardItem**), 325
 sets, 32, 33
setString(_:forType:) (**NSPasteboardItem**), 325
setUp(), 424, 429

- setValue(_: forKey:)**, 135
 - shadowing (properties), 17
 - shadows, drawing, 284
 - Shared User Defaults Controller, 139
 - sharedDocumentController()** (**NSDocumentController**), 207
 - sheetParent (**NSWindow**), 354
 - sheets
 - and alerts, 251
 - vs. modal window, 355
 - presenting, 353-355
 - Visible At Launch, 349
 - size (**NSAttributedString**), 316
 - size (**NSRect**), 274
 - sizeWithAttributes(_:)** (**NSString**), 316
 - sliders
 - about, 79
 - setting range of, 83
 - snippets (code), 101-103
 - sort descriptors, 171-176, 235
 - sortDescriptors (**NSTableView**), 173
 - sorting (array controllers), 171-175
 - sorting (table views), 176
 - speech synthesis, implementing, 106, 107
 - split view controllers, 459
 - springs (autoresizing), 375
 - SQLite, 236, 237
 - stack (memory), 149
 - stack trace, 148, 149
 - stacks, undo and redo, 190
 - standardUserDefaults()** (**NSUserDefaults**), 240
 - states, graphics, 278, 284
 - static methods, 27
 - stopModalWithCode(_:)** (**NSApplication**), 355
 - storage, application, 215, 216
 - store types (Core Data), 236
 - storyboards
 - about, 457
 - loading, 470
 - scenes, 457, 463, 470
 - segues, 457, 459
 - string interpolation, 37
 - stringForType(_:)** (**NSPasteboardItem**), 325
 - strings
 - (see also **NSAttributedString**, **NSString**)
 - initializers for, 33
 - interpolation, 37
 - isEmpty, 34
 - literal, 32
 - NSAttributedString**, 314-317
 - and **NSString**, 56
 - strings files (localization), 379, 380, 382-391
 - strings tables (localization), 385
 - stringValue, 78
 - stroke()** (**NSBezierPath**), 277, 278
 - strong reference cycles, 65-67
 - strong references, 65
 - structures, 41-44
 - vs. classes, 53, 54
 - struts (autoresizing), 375
 - subclassing
 - vs. extensions, 318
 - vs. helper objects, 110
 - subscripting
 - arrays, 32
 - dictionaries, 36
 - subviews (**NSView**), 63
 - Swift, 28
 - about, xx, 27
 - documentation for, 39
 - and Objective-C, 55
 - switch, 38
 - switch statements, 38
 - systemFontOfSize(_:)** (**NSFont**), 313
- ## T
- tab images, 454
 - tab view controllers, 445, 446, 449-455
 - table cell views
 - about, 127, 128
 - with checkbox, 226
 - different views in, 225-227
 - with formatters, 226
 - with images, 225
 - table columns, 125
 - table header views, 125
 - table view
 - delegate's role, 120
 - table view cells, 127
 - table views
 - (see also **NSTableView**, **NSTableViewDataSource**, **NSTableViewDelegate**)
 - about, 119-121

- Apple's guide to, 134
 - binding to array controllers, 160-164
 - bindings, data supplied from, 129
 - cell-based, 126
 - cells in, 126-128
 - and clip views, 125
 - as collections of classes, 124
 - columns in, 125
 - and data sources, 120, 128-131
 - data source methods vs. bindings, 129, 134
 - delegate for, 131-133
 - displaying data with bindings, 130
 - header views, 125
 - in Interface Builder, 123-128
 - and scroll views, 125
 - and scrollers, 125
 - Selection Indexes, 161
 - sorting in, 171-176
 - view-based, 126
 - tableView(_: objectValueForTa...)**, 121, 129
 - tableViewSelectionDidChange(_:)**, 133
 - Taligent, 69
 - .tar files, 497
 - target (**NSControl**)
 - setting programmatically, 96
 - target-action (**NSControl**), 78, 418
 - targets
 - (see also actions, controls)
 - about, 78
 - array controllers as, 164, 165
 - nil, 326-328
 - project, 503
 - as weak references, 82
 - targets (in projects), 423
 - tearDown()**, 424, 429
 - test fixtures, 431
 - testExample()**, 424
 - testing (see unit testing)
 - testPerformanceExample()**, 424
 - text fields
 - alignment of, 12
 - as table view cells, 127
 - behavior, setting, 12
 - cut, copy, and paste, 24
 - editable, 12
 - changing fonts, 12
 - in Interface Builder, 9
 - as labels, 80
 - and placeholder text, 105
 - selectable, 12
 - styles of, 80
 - .tgz files, 497
 - thread synchronization (multithreading), 485, 486
 - threads (see multithreading)
 - Time Profiler (Instruments), 481-484
 - timers, 341-343
 - timestamp (**NSEvent**), 298
 - titleBarFontOfSize(_:)** (**NSFont**), 313
 - toll-free bridging, 56
 - toolTipsFontOfSize(_:)** (**NSFont**), 313
 - top-level objects, 444
 - trailing closure syntax, 256
 - trops, 33
 - tuples, 37
 - type inference, 30
 - type safety, 142
 - types
 - (see also classes, enumerations, structures, UTIs)
 - boolean, 31
 - bridging, 56-58
 - casting, 56-58
 - floating-point, 31, 33
 - hashable, 31
 - inference of, 30
 - instances of, 33
 - integer, 31
 - nested, 411
 - reference, 53, 54
 - sets, 32, 33
 - specifying, 30
 - tuples, 37
 - values, 53, 54
 - types (**NSPasteboardItem**), 325
- ## U
- unarchiveObjectWithData(_:)**, 212
 - unarchiving (NIB files), 14, 75-77
 - undo
 - about, 189-192
 - implementing, 197-202
 - undo stack, 190
 - Unicode warning for `Localizable.strings`, 388
 - unit testing
 - about, 423

- assertions, 424-427
 - asynchronous tasks, 435, 436
 - planning for, 431
 - refactoring for, 431-434
 - test fixtures, 431
 - using constants in, 428-431
 - Unix, xix
 - unowned references, 67
 - updateChangeCount(_:)** (**NSDocument**), 218
 - URLForResource(_:withExtension:)** (**NSBundle**), 391
 - URLsForDirectory(_:inDomains:)** (**NSFileManager**), 215
 - user defaults, 239, 240
 - user interfaces
 - (see also **Interface Builder**, **views**)
 - with bindings, 136, 145
 - creating in **Interface Builder**, 8-13
 - master/detail, 444
 - as view layer in **MVC**, 4
 - user preferences, 239, 240
 - userFixedPitchFontOfSize(_:)** (**NSFont**), 313
 - userFontOfSize(_:)** (**NSFont**), 313
 - utilities (**Xcode**), 3
 - UTIs (universal type identifiers)
 - about, 218
 - exported, 215
 - and pasteboards, 324, 325, 330
- ## V
- validate(_:error:)**, 185
 - validateMenuItem(_:)** (**NSMenuValidation**), 329
 - validation
 - key-value, 181, 183-186
 - value transformers, 187
 - value transformers, 187
 - value types, 53, 54
 - valueForKey(_:)**, 135
 - var, 29
 - variables, 29
 - capturing in closures, 252
 - variables view, 49, 149
 - view (**NSViewController**), 440
 - view controllers, 459
 - (see also **NSViewController**, **views**)
 - about, 439-441
 - architecture, 466
 - benefits of using, 440, 444
 - container, 445
 - instantiating in storyboards, 470
 - loading views, 451
 - making reusable, 465
 - and memory management, 444
 - and NIB files, 442
 - pre-OSX 10.0, 447
 - reason for, 439, 440
 - split, 459
 - and swapping views, 449-455
 - tab, 445, 446, 449-455
 - views of, 440, 465
 - ways to connect multiple, 466
 - when to use, 446
 - vs. window controllers, 444, 446
 - view hierarchies, 5, 10
 - and responder chain, 327
 - view hierarchy popover, 178-180
 - view layer (**MVC**), 4, 5
 - (see also **views**)
 - view swapping, 449-455
 - view-based tables, 126
 - viewDidMoveToWindow()** (**NSView**), 310
 - viewLoaded** (**NSViewController**), 451
 - views
 - (see also **NSView**, **view controllers**)
 - adding in **Interface Builder**, 9-13
 - archiving, 14
 - attributes, configuring, 11-13
 - binding to array controller, 162
 - connecting in **Interface Builder**, 17-22, 132
 - content views, 73
 - copying and pasting, 85, 86
 - creating custom, 273-279
 - creating programmatically, 293, 294
 - described, 271
 - drawing, 276-279
 - and first-responder status, 305-308
 - flipped, 295
 - and focus rings, 309
 - hierarchies of, 5, 10
 - in **MVC**, 4, 5
 - and key view loop, 310
 - in NIB files, 14
 - positioning, 274-276
 - unarchiving, 14, 75-77

WKWebView, 465
in XIB files, 14
Visible At Launch, 73, 76, 349
Visual Format Language (Auto Layout), 371-373
visualizeConstraints(_:) (**NSWindow**), 375

W

wantsLayer (**NSView**), 473
weak, 65
weak references, 65, 67
web services
about, 405, 406
and asynchronous tasks, 409-413
and completion handlers, 409
and completion handlers, 410-418
and HTTP, 405
making requests, 406, 410
reusing classes with, 409
synchronous API, 409
testing asynchronous tasks, 435, 436
and threads, 409
ways to fetch asynchronously, 410
willChangeValueForKey(_:), 141
willSet (property observer), 108
window (**NSEvent**), 298
window (**NSWindowController**), 75-77
window controllers
(see also **NSWindowController**, windows)
and **AppDelegate**, 14, 24-26
and documents, 210
initializing, 17
instantiating in storyboards, 470
loading NIB files, 75-77
loading windows, 75-77
and NIB names, 24-26
vs. view controllers, 444, 446
when to use, 446
and windows, 75-77
window servers, xix
windowControllerDidLoadNib(_:)
(**NSDocument**), 209
windowDidLoad(), 93, 94
windowNibName, 25
windowNibName (**NSWindowController**), 24, 75
windows
(see also **NSWindow**, window controllers)
and content views, 4

described, 271
disabling resizing of, 125
first responders of, 305-308
key, 305, 306
loading, 75-77
modal, 355
resizing, 125
showing, 75
and view hierarchies, 5, 10
Visible At Launch, 73, 76, 349
and window controllers, 75-77
windowShouldClose(_:) (**NSWindowDelegate**),
113
WKWebView, 465
writeObjects(_:) (**NSPasteboard**), 324
writeToURL(_:ofType:error:) (**NSDocument**),
208

X

.xcdatamodeld (Core Data), 221
Xcode
(see also debugging tools, Interface Builder)
adding localizations in, 381
assistant editor, 132
attributes inspector, 11
auto-complete, 111
Cocoa documentation in, 88
code snippets in, 101-103
connections inspector, 21
creating classes in, 6-8
creating projects in, 1-3
data model inspector, 236
debugger, 147-151
files in, 3
groups, 3
identity inspector, 76
Instruments, 481-484
jump bar, 105
modules, 427
navigator area, 3
navigators, 3
overview, xxi
playgrounds, 28-30
project navigator, 3
project source directories, 386
project window, 3
Quick Help, 30

- saving files in, 14
- testing in, 423
- Time Profiler, 481-484
- using // MARK:, 105
- utilities area, 3
- variables view, 49, 149
- XCTAssert(expr)**, 424
- XCTAssertEqual(expr)**, 424
- XCTAssertEqualWithAccuracy(expr)**, 424
- XCTAssertFalse(expr)**, 424
- XCTAssertNotEqual(expr)**, 424
- XCTAssertNotEqualWithAccuracy(expr)**, 424
- XCTAssertNotNil(expr)**, 424
- XCTAssertTrue(expr)**, 424
- XCTest (framework), 424, 425
- XCTestCase**
 - setUp()**, 424, 429
 - tearDown()**, 424, 429
 - testExample()**, 424
 - testPerformanceExample()**, 424
- XCTFail()**, 424, 434
- XIB files
 - (see also NIB files)
 - archiving files in, 14
 - connections in, 17, 76
 - defined, 14
 - File's Owner, 76
 - localizing, 380, 382-385
 - naming conventions for, 72
 - and NIB files, 14
 - placeholders, 76
 - pronounced as “zib”, 8
- XML parsing, 420, 421

Z

- zip files
 - for distribution, 505
 - inspecting, 489
- zipinfo** utility, 489
- zombie objects, 98