

Objective-C Fundamentals



Christopher K. Fairbairn
Johannes Fahrenkrug
Collin Ruffenach

Objective-C Fundamentals

Objective-C Fundamentals

CHRISTOPHER K. FAIRBAIRN
JOHANNES FAHRENKRUG
COLLIN RUFFENACH



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

©2012 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Troy Mott
Technical editor: Amos Bannister
Copyeditor: Linda Kern
Proofreader: Katie Tennant
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781935182535

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – MAL – 17 16 15 14 13 12 11

brief contents

PART 1	GETTING STARTED WITH OBJECTIVE-C.....	1
1	■ Building your first iOS application	3
2	■ Data types, variables, and constants	28
3	■ An introduction to objects	55
4	■ Storing data in collections	74
PART 2	BUILDING YOUR OWN OBJECTS	95
5	■ Creating classes	97
6	■ Extending classes	124
7	■ Protocols	144
8	■ Dynamic typing and runtime type information	163
9	■ Memory management	177
PART 3	MAKING MAXIMUM USE OF FRAMEWORK FUNCTIONALITY	201
10	■ Error and exception handling	203
11	■ Key-Value Coding and NSPredicate	212
12	■ Reading and writing application data	228
13	■ Blocks and Grand Central Dispatch	257
14	■ Debugging techniques	276

contents

preface xv
acknowledgments xvi
about this book xviii
author online xxi
about the cover illustration xxii

PART 1 GETTING STARTED WITH OBJECTIVE-C1

1 Building your first iOS application 3

- 1.1 Introducing the iOS development tools 4
 - Adapting the Cocoa frameworks for mobile devices* 4
- 1.2 Adjusting your expectations 5
 - A survey of hardware specifications, circa mid-2011* 6
 - Expecting an unreliable internet connection* 7
- 1.3 Using Xcode to develop a simple Coin Toss game 7
 - Introducing Xcode—Apple’s IDE* 8
 - Launching Xcode easily* 8 ■ *Creating the project* 9
 - Writing the source code* 12
- 1.4 Hooking up the user interface 15
 - Adding controls to a view* 15 ■ *Connecting controls to source code* 17

- 1.5 Compiling the Coin Toss game 21
- 1.6 Taking Coin Toss for a test run 21
 - Selecting a destination* 22
 - *Using breakpoints to inspect the state of a running application* 23
 - *Running the CoinToss game in the iPhone simulator* 24
 - Controlling the debugger* 25
- 1.7 Summary 27

2 *Data types, variables, and constants* 28

- 2.1 Introducing the Rental Manager application 29
 - Laying the foundations* 29
- 2.2 The basic data types 32
 - Counting on your fingers—integral numbers* 32
 - Filling in the gaps—floating-point numbers* 35
 - Characters and strings* 37
 - *Boolean truths* 39
- 2.3 Displaying and converting values 40
 - NSLog and Format Specifiers* 40
 - *Type casts and type conversions* 43
- 2.4 Creating your own data types 44
 - Enumerations* 44
 - *Structures* 46
 - *Arrays* 48
 - The importance of descriptive names* 50
- 2.5 Completing Rental Manager v1.0, App Store here we come! 52
- 2.6 Summary 54

3 *An introduction to objects* 55

- 3.1 A whirlwind tour of object-oriented programming concepts 56
 - What's wrong with procedural-based languages such as C?* 56
 - What are objects?* 56
 - *What are classes?* 57
 - Inheritance and polymorphism* 57
- 3.2 The missing data type: id 58
- 3.3 Pointers and the difference between reference and value types 59
 - Memory maps* 59
 - *Obtaining the address of a variable* 59
 - Following a pointer* 60
 - *Comparing the values of pointers* 61

- 3.4 Communicating with objects 62
 - Sending a message to an object* 62
 - *Sending a message to a class* 63
 - *Sending nonexistent messages* 64
 - Sending messages to nil* 65
- 3.5 Strings 66
 - Constructing strings* 66
 - *Extracting characters from strings* 67
 - *Modifying strings* 68
 - Comparing strings* 69
- 3.6 Sample application 69
- 3.7 Summary 72

4 Storing data in collections 74

- 4.1 Arrays 75
 - Constructing an array* 75
 - *Accessing array elements* 76
 - Searching for array elements* 77
 - *Iterating through arrays* 79
 - Adding items to an array* 80
- 4.2 Dictionaries 82
 - Constructing a dictionary* 82
 - *Accessing dictionary entries* 84
 - *Adding key/value pairs* 85
 - Enumerating all keys and values* 86
- 4.3 Boxing 88
 - The NSNumber class* 89
 - *The NSValue class* 90
 - nil vs. NULL vs. NSNull* 90
- 4.4 Making the Rental Manager application data driven 91
- 4.5 Summary 94

PART 2 BUILDING YOUR OWN OBJECTS.....95

5 Creating classes 97

- 5.1 Building custom classes 98
 - Adding a new class to the project* 98
- 5.2 Declaring the interface of a class 99
 - Instance variables (ivars)* 100
 - *Method declarations* 101
 - Fleshing out the header file for the CTrentalProperty class* 105
- 5.3 Providing an implementation for a class 106
 - Defining method implementations* 106
 - *Accessing instance variables* 106
 - *Sending messages to self* 107
 - Fleshing out the method file for the CTrentalProperty class* 108

- 5.4 Declared properties 109
 - @property syntax* 109
 - *Synthesizing property getters and setters* 112
 - *Dot syntax* 113
- 5.5 Creating and destroying objects 115
 - Creating and initializing objects* 115
 - *init is pretty dumb* 116
 - Combining allocation and initialization* 118
 - Destroying objects* 119
- 5.6 Using the class in the Rental Manager application 120
- 5.7 Summary 123

6 *Extending classes* 124

- 6.1 Subclassing 124
 - What is subclassing?* 125
- 6.2 Adding new instance variables 127
- 6.3 Accessing existing instance variables 129
 - Manual getters and setters approach* 130
- 6.4 Overriding methods 131
 - Overriding the description method* 132
- 6.5 Class clusters 134
 - Why use class clusters* 134
 - *Multiple public clusters* 135
- 6.6 Categories 136
 - Extending classes without subclassing* 136
 - Using a category* 136
 - *Considerations when using categories* 138
- 6.7 Subclassing in your demo application 138
 - Creating and subclassing CTLease* 139
 - Creating CTPeriodicLease as a subclass of CTLease* 140
 - Creating CTFixedLease as a subclass of CTLease* 141
- 6.8 Summary 143

7 *Protocols* 144

- 7.1 Defining a protocol 145
- 7.2 Implementing a protocol 146
 - Creating the protocol method callers* 147
 - *Making a class conform to a protocol* 148

- 7.3 Important protocols 150
 - <UITableViewDataSource>* 150 ▪ *<UITableViewDelegate>* 153
 - <UIActionSheetDelegate>* 157 ▪ *NSXMLParser* 158
- 7.4 Summary 162

8 *Dynamic typing and runtime type information* 163

- 8.1 Static vs. dynamic typing 164
 - Making assumptions about the runtime type* 164
- 8.2 Dynamic binding 166
- 8.3 How messaging works 166
 - Methods, selectors, and implementations* 167
 - Handling unknown selectors* 169 ▪ *Sending a message to nil* 170
- 8.4 Runtime type information 171
 - Determining if a message will respond to a message* 171
 - Sending a message generated at runtime* 171
 - Adding new methods to a class at runtime* 173
- 8.5 Practical uses of runtime type introspection 174
- 8.6 Summary 176

9 *Memory management* 177

- 9.1 Object ownership 178
- 9.2 Reference counting 179
 - Releasing an object* 180 ▪ *Retaining an object* 181
 - Determining the current retain count* 182
- 9.3 Autorelease pools 184
 - What is an autorelease pool?* 185 ▪ *Adding objects to the autorelease pool* 185 ▪ *Creating a new autorelease pool* 185 ▪ *Releasing objects in a pool* 187
 - Why not use an autorelease pool for everything?* 187
- 9.4 Memory zones 190
- 9.5 Rules for object ownership 192
- 9.6 Responding to low-memory warnings 193
 - Implementing the *UIApplicationDelegate* protocol* 193
 - Overriding *didReceiveMemoryWarning** 194 ▪ *Observing the *UIApplicationDidReceiveMemoryWarningNotification** 197
- 9.7 Summary 199

PART 3 MAKING MAXIMUM USE OF FRAMEWORK FUNCTIONALITY.....201

10 *Error and exception handling* 203

- 10.1 NSError—handling errors the Cocoa way 204
Getting NSError to talk 204 ▪ *Examining NSError’s userInfo Dictionary* 205
- 10.2 Creating NSError objects 206
Introducing RentalManagerAPI 206 ▪ *Handling and displaying RentalManagerAPI errors* 209
- 10.3 Exceptions 210
Throwing exceptions 210 ▪ *Catching exceptions* 211
- 10.4 Summary 211

11 *Key-Value Coding and NSPredicate* 212

- 11.1 Making your objects KVC-compliant 213
Accessing properties via KVC 214 ▪ *Constructing key paths* 215
Returning multiple values 215 ▪ *Aggregating and collating values* 216
- 11.2 Handling special cases 217
Handling unknown keys 217 ▪ *Handling nil values* 218
- 11.3 Filtering and matching with predicates 219
Evaluating a predicate 219 ▪ *Filtering a collection* 220
Expressing your predicate condition 220 ▪ *More complex conditions* 221 ▪ *Using key paths in predicate expressions* 222 ▪ *Parameterizing and templating predicate expressions* 223
- 11.4 Sample application 224
- 11.5 Summary 227

12 *Reading and writing application data* 228

- 12.1 Core Data history 229
What does Core Data do? 229
- 12.2 Core Data objects 231
Managed object context 231 ▪ *Persistent store coordinator* 231
Managed object model 232 ▪ *Persistent object store* 232

- 12.3 Core Data resources 232
 - Core Data entities* 232
 - *Core Data attributes* 233
 - Core Data relationships* 234
- 12.4 Building the PocketTasks application 234
 - Examining the Xcode Core Data template* 234
 - Building the data model* 235
 - *Defining the relationships* 236
 - Creating Person entities in pure code* 237
 - *Fetching Person entities in pure code* 239
 - *Adding a master TableView* 240
 - Adding and deleting people* 243
 - *Managing tasks* 246
 - Using model objects* 249
- 12.5 Beyond the basics 251
 - Changing the data model* 251
 - *Performance* 253
 - Error handling and validation* 253
- 12.6 Summary 256

13 **Blocks and Grand Central Dispatch** 257

- 13.1 The syntax of blocks 258
 - Blocks are closures* 260
 - *Blocks and memory management* 262
 - Block-based APIs in Apple's iOS frameworks* 264
- 13.2 Performing work asynchronously 265
 - Meet GCD* 266
 - *GCD fundamentals* 266
 - Building RealEstateViewer* 267
 - *Making the image search asynchronous* 271
 - *Making the image loading asynchronous* 273
- 13.3 Summary 274

14 **Debugging techniques** 276

- 14.1 Building an application, complete with bugs 277
- 14.2 Understanding NSLog 278
- 14.3 Bringing memory leaks under control with Instruments 281
- 14.4 Detecting zombies 284
- 14.5 Summary 286

- appendix A Installing the iOS SDK* 288
- appendix B The basics of C* 293
- appendix C Alternatives to Objective-C* 312
- index* 327

preface

Having been involved in the development of applications on a variety of mobile platforms for more than 10 years, I knew the iPhone was something exciting when it was first introduced back in 2008. From a consumer viewpoint, it had the intangible and hard-to-define elements required to make a compelling device that you just wanted to keep coming back to and interact with. To the user, the device “felt right” and it was a pleasure to use rather than simply being a means to an end to achieve a singular task.

As new and refreshing as the iPhone user experience was, the development tools that supported it were also rather unique. For developers without prior exposure to Apple products, the platform was full of new terms, tools, and concepts to grok. This book is designed to provide an introduction to these technologies, with emphasis on covering only those features available for use by iOS applications. For someone learning a new environment, there’s nothing worse than reading a section of a book and attempting to implement what you learn in an application of your own design, only to realize that the Objective-C or Cocoa feature discussed is only applicable to desktop Mac OS X applications.

I hope you enjoy reading this book and you’ll remember its tips while you develop the next iTunes App Store Top 10 application!

CHRISTOPHER FAIRBAIRN

acknowledgments

A technical book has more than what first meets the eye. A significant number of skills are required to make sure not only that it is technically correct, but that it reads well, looks good, and is approachable by the intended audience. Thus, we thank the entire Manning staff, without whom this book would not exist in its present form. They did more than just correct our errors and polish our words; they also helped make integral decisions about the organization and the contents of the book—decisions that improved it dramatically.

At Manning Publications, we'd like to thank Emily Macel who helped us at an early stage to shape and focus our writing style. Thanks also to Troy Mott, our acquisitions editor, who initially approached us to develop the book and who stayed with us every step of the way. And thanks to Amos Bannister for expertly tech editing the final manuscript during production and for testing the code.

Finally, we'd like to thank the reviewers who generously agreed to read our manuscript as we worked on it; they improved the book immensely: Ted Neward, Jason Jung, Glenn Stokol, Gershon Kagan, Cos DiFazio, Clint Tredway, Christopher Haupt, Berndt Hamboeck, Rob Allen, Peter Scott, Lester Lobo, Frank Jania, Curtis Miller, Chuck Hudson, Carlton Gibson, Emeka Okereke, Pratik Patel, Kunal Mittal, Tyson Maxwell, TVS Murthy, Kevin Butler, David Hanson, Timothy Binkley-Jones, Carlo Bottiglieri, Barry Ezell, Rob Allen, David Bales, Pierre-Antoine Grégoire, Kevin Munc, Christopher Schultz, Carlton Gibson, Jordan Duval-Arnould, Robert McGovern, Carl Douglas, Dave Mateer, Fabrice Dewasmes, David Cuillerier, Dave Verwer, and Glen Marcus.

Christopher would like to thank his fiancée Michele for giving support and encouragement while he worked on this book. She is in many ways an unsung fourth “author” and has contributed greatly. Also, he would like to thank the staff at Manning for their understanding in a trying year involving burglaries, setbacks, and no less than three significant earthquake events. Last but not least, he is thankful for all the support from the important people in his life.

Johannes would like to thank Troy Mott for getting him on board with this project, and Aaron Hillegass for helping him get started with Mac development in the first place, and for being an all-around nice guy. Most of all, he’d like to thank his loving and ever-supportive wife Simone (hey, he already *did* get rid of some of his nerd T-shirts!) and his parents Fred and Petra.

Collin would like to thank Manning Publications for giving him the opportunity to work on this book and the language he is so passionate about. He acknowledges Aaron Hillegass for being a dedicated evangelist for this fantastic language and all its platforms; most of what he knows about Objective-C can be attributed to Aaron’s work. He would like to thank Panic, OmniGraffle, Delicious Library, Rouge Amoeba, MyDreamApp.com, and all the other inspiring software development companies that set such a high bar in the mobile space with their fantastic desktop software. He also thanks ELC Technologies for being so supportive in this endeavor. Thanks to his parents Debbie and Steve for all of their support, and his brothers Brett and Stephen for helping hash out ideas for the book. A big thanks goes to his girlfriend Caitlin for helping him stay dedicated and focused. And finally, he would like to thank Brandon Trebitowski, author with Manning Publications, for his dedication to this platform and for educating young developers.

about this book

Objective-C Fundamentals is an introductory book, intended to complement other books focused on iPhone and iPad application development such as *iOS 4 in Action*. While many books have been written on how to develop iOS applications, most focus on the individual APIs and frameworks provided by the device, rather than the unique language, *Objective-C*, which is a cornerstone of Apple's development platform. To truly master the platform, you must have a strong grip on the language, and that is what this book intends to provide. *Objective-C Fundamentals* is a book that focuses on learning Objective-C in the context of iOS application development. No time is spent discussing aspects or elements of the language that are not relevant to iOS. All examples are fully usable on your own iOS-powered device. We encourage you to read this book straight through, from chapter 1 to chapter 14. This process will introduce the platform, discuss how to program for the iPhone and iPad, and walk you through the entire process step by step.

The audience

We've done our best to make this book accessible to everyone who is interested in creating successful iOS applications using the native Objective-C–based development tools.

If you want to learn about iOS programming, you should have some experience with programming in general. It'd be best if you've worked with C or at least one object-oriented language before, but that's not a necessity. If you haven't, you may find the introduction to the C programming language in appendix B helpful, and you should expect to do some research on your own to bolster your general programming

skills. There's no need to be familiar with Objective-C, Cocoa, or Apple programming in general. We'll give you everything you need to become familiar with Apple's unique programming style. You'll probably have a leg-up if you understand object-oriented concepts; but it's not necessary (and again, you'll find an introduction in chapter 3).

Roadmap

Chapter 1 introduces the tools surrounding Objective-C and iOS application development, and covers the creation of a basic game, ready to run on your device.

Chapter 2 kicks things off by highlighting how data is stored and represented within an Objective-C–based application.

Chapter 3 looks at how Objective-C takes small quantities of data and packages them with logic to form reusable components called *classes*.

Chapter 4 shifts the focus by taking a look at some of the classes, provided out of the box by Cocoa Touch, that can be used to store multiple pieces of related data.

Chapter 5 covers how to create your own custom classes and objects. Learning how to create your own classes is an important building block to becoming a productive developer.

Chapter 6 takes a look at how you can build on top of the foundations provided by an existing class to create a more specialized or customized version of a class without needing to rewrite all of its functionality from scratch.

Chapter 7 discusses how classes can be defined to provide specific functionality, without resorting to requiring all classes to inherit from a common base class. This concept is provided with a language construct called a *protocol*.

Chapter 8 looks deeply at some of the aspects of Objective-C that make it unique. The important distinction between message sending and method invocation is discussed and some powerful programming techniques are demonstrated.

Chapter 9 covers how to keep track of memory allocation within an Objective-C application. Since no automatic garbage collector is available, simple rules are discussed which will allow you to expertly craft applications without introducing memory leaks.

Chapter 10 looks at `NSError` and at some real-life use cases for exceptions, which tools will help you deal with errors gracefully.

Chapter 11 covers Key Value Coding (KVC) and `NSPredicate`-based queries, which are a surprisingly flexible way to filter, search and sort data within Cocoa Touch–based applications.

Chapter 12 gets you started with Core Data and teaches you everything you'll need to know to leverage Core Data for all of your most common data persistence needs.

Chapter 13 introduces a language construct called a *block* and demonstrates this by showing how Grand Central Dispatch (GCD) can be used to simplify multithreaded programming, since it takes care of all the complicated heavy lifting for us.

No application is perfect first time around, so chapter 14 rounds out the book with a discussion on debugging techniques that can help resolve unwanted logic errors and memory leaks quickly and efficiently.

The appendixes contain additional information that didn't fit with the flow of the main text. Appendix A outlines how to enroll in the iOS Developer Program and set up your physical iPhone or iPad device in order to run your own applications on them. Appendix B provides a basic overview of the C programming language that Objective-C is a descendant of. This will be ideal for developers with little experience of a C-based language and those that have previously only developed in languages such as Ruby, Python, or Java. Appendix C outlines some of the alternatives you can use to develop iOS applications, and compares their advantages and disadvantages to Objective-C.

Writing this book was truly a collaborative effort. Chris wrote chapters 1 through 5, 8, 9, 11, 14, and appendixes B and C. Johannes contributed chapters 10, 12, and 13, and appendix A; and Collin was responsible for chapters 6 and 7.

Code conventions and downloads

Code examples appear throughout this book. Longer listings appear under clear listing headings, and shorter listings appear between lines of text. All code is set in a monospace font like `this` to differentiate it from the regular font. Class names have also been set in code font; if you want to type it into your computer, you'll be able to clearly make it out.

With the exception of a few cases of abstract code examples, all code snippets began life as working programs. You can download the complete set of programs from www.manning.com/Objective-CFundamentals. You'll find two ZIP files there, one for each of the SDK programs. We encourage you to try the programs as you read; they include additional code that doesn't appear in the book and provide more context. In addition, we feel that seeing a program work can elucidate the code required to create it.

The code snippets in this book include extensive explanations. We often include short annotations beside the code; and sometimes numbered cueballs beside lines of code link the subsequent discussion to the code lines.

Software requirements

An Intel-based Macintosh running OS X 10.6 or higher is required to develop iOS applications. You also need to download the Xcode IDE and iOS SDK. Xcode is available for purchase in the Mac App Store and the iOS SDK is freely downloadable.

However, the best approach to obtaining Xcode and developing iOS applications is to pay a yearly subscription fee for the iOS Developer Program (<http://developer.apple.com/programs/ios/>). This will provide free access to Xcode and iOS SDK downloads as well as enable testing and deployment of applications on real iPhone and iPad devices, and the iTunes App Store.

author online

Purchase of *Objective-C Fundamentals* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/Objective-CFundamentals. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the Author Online forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the cover illustration

On the cover of *Objective-C Fundamentals* is “A man from Tinjan, Istria,” a village in the interior of the peninsula of Istria in the Adriatic Sea, off Croatia. The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian’s retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

In this region of Croatia, the traditional costume for men consists of black woolen trousers, jacket, and vest decorated with colorful embroidered trim. The figure on the cover is wearing a lighter version of the costume, designed for hot Croatian summers, consisting of black linen trousers and a short, black linen jacket worn over a white linen shirt. A gray belt and black wide-brimmed hat complete the outfit.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It’s now hard to tell apart the inhabitants of different continents, let alone of different hamlets or towns separated by only a few miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by illustrations from old books and collections like this one.

Part 1

Getting started with Objective-C

Becoming an iOS application developer can require mastering a number of new tools and technologies such as the Xcode IDE and the Objective-C programming language. Although plenty of step-by-step how-to tutorials are available online for developing small example applications, such as a photo viewer or RSS news feed display application and so on, these typically don't provide much in the way of background information to enable you to develop applications of your own design.

In this part of the book, you'll develop a small game as a learning exercise to become familiar with the development tools surrounding the Objective-C language. As you progress through these chapters, you'll discover more of the meaning and purpose behind the individual steps and tasks outlined in developing the game so you can see the theory and purpose behind each step.

Toward the end of this part, you'll reach a stage where you can confidently create a new project within Xcode and describe the purpose of each file and the meaning behind the various code snippets found within them.

Building your first iOS application



This chapter covers

- Understanding the iOS development environment
- Learning how to use Xcode and Interface Builder
- Building your first application

As a developer starting out on the iOS platform, you're faced with learning a lot of new technologies and concepts in a short period of time. At the forefront of this information overload is a set of development tools you may not be familiar with and a programming language shaped by a unique set of companies and historical events.

iOS applications are typically developed in a programming language called Objective-C and supported by a support library called Cocoa Touch. If you've already developed Mac OS X applications, you're probably familiar with the desktop cousins of these technologies. But it's important to note that the iOS versions of these tools don't provide exactly the same capabilities, and it's important to learn the restrictions, limitations, and enhancements provided by the mobile device. In some cases, you may even need to unlearn some of your desktop development practices.

While developing iOS applications, most of your work will be done in an application called *Xcode*. Xcode 4, the latest version of the IDE, has Interface Builder (for creating the user interface) built directly into it. Xcode 4 enables you to create, manage, deploy, and debug your applications throughout the entire software development lifecycle. When creating an application that supports more than one type of device powered by the iOS, you may wish to present slightly different user interfaces for specific device types while powering all variants via the same core application logic underneath. Doing so is easier if the concept of model-view-controller separation is used, something that Xcode 4 can help you with.

This chapter covers the steps required to use these tools to build a small game for the iPhone, but before we dive into the technical steps, let's discuss the background of the iOS development tools and some of the ways mobile development differs from desktop and web-based application development.

1.1 *Introducing the iOS development tools*

Objective-C is a strict superset of the procedural-based C programming language. This fact means that any valid C program is also a valid Objective-C program (albeit one that doesn't make use of any Objective-C enhancements).

Objective-C extends C by providing object-oriented features. The object-oriented programming model is based on sending messages to objects, which is different from the model used by C++ and Java, which call methods directly on an object. This difference is subtle but is also one of the defining features that enables many of Objective-C's features that are typically more at home in a dynamic language such as Ruby or Python.

A programming language, however, is only as good as the features exposed by its support libraries. Objective-C provides syntax for performing conditional logic and looping constructs, but it doesn't provide any inherent support for interacting with the user, accessing network resources, or reading files. To facilitate this type of functionality without requiring it to be written from scratch for each application, Apple includes in the SDK a set of support libraries collectively called *Cocoa Touch*. If you're an existing Java or .NET developer, you can view the Cocoa Touch library as performing a purpose similar to the Java Class Library or .NET's Base Class Libraries (BCL).

1.1.1 *Adapting the Cocoa frameworks for mobile devices*

Cocoa Touch consists of a number of frameworks (commonly called *kits*). A framework is a collection of classes that are grouped together by a common purpose or task. The two main frameworks you use in iPhone applications are Foundation Kit and UIKit. Foundation Kit is a collection of nongraphical system classes consisting of data structures, networking, file IO, date, time, and string-handling functions, and UIKit is a framework designed to help develop GUIs with rich animations.

Cocoa Touch is based on the existing Cocoa frameworks used for developing desktop applications on Mac OS X. But rather than making Cocoa Touch a direct line-by-line

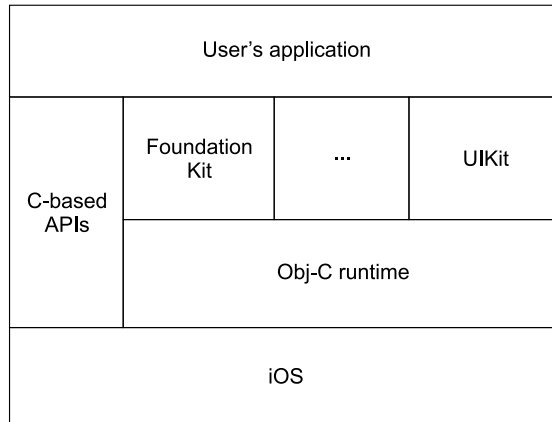


Figure 1.1 The software runtime environment for iOS applications, showing the operating system, Objective-C runtime, and Cocoa Touch framework layers

port to the iPhone, Apple optimized the frameworks for use in iPhone and iPod Touch applications. Some Cocoa frameworks were even replaced entirely if Apple thought improvements in functionality, performance, or user experience could be achieved in the process. UIKit, for example, replaced the desktop-based AppKit framework.

The software runtime environment for native iOS applications is shown in figure 1.1. It's essentially the same software stack for desktop applications if you replace iOS with Mac OS X at the lowest level and substitute some of the frameworks in the Cocoa layer.

Although the Cocoa Touch frameworks are Objective-C–based APIs, the iOS development platform also enables you to access standard C-based APIs. The ability to reuse C (or C++) libraries in your Objective-C applications is quite powerful. It enables you to reuse existing source code you may have originally developed for other mobile platforms and to tap many powerful open source libraries (license permitting), meaning you don't need to reinvent the wheel. As an example, a quick search on Google will find existing C-based source code for augmented reality, image analysis, and barcode detection, to name a few possibilities, all of which are directly usable by your Objective-C application.

1.2 Adjusting your expectations

With a development environment that will already be familiar to existing Mac OS X developers, you may mistakenly think that the iPhone is just another miniature computing device, similar to any old laptop, tablet, or netbook. That idea couldn't be any further from the truth. An iPhone is more capable than a simple cell phone but less so than a standard desktop PC. As a computing device, it fits within a market space similar to that of netbooks, designed more for casual and occasional use throughout the day in a variety of situations and environments than for sustained periods of use in a single session.

1.2.1 A survey of hardware specifications, circa mid-2011

On taking an initial look at an iPhone 4, you'll undoubtedly notice the 3.5-inch screen, 960 x 640 pixels, that virtually dominates the entire front of the device. Its general size and the fact that the built-in touch screen is the only way for users to interact with the device can have important ramifications on application design. Although 960 x 640 is larger than many cell phones, it probably isn't the screen on which to view a 300-column-by-900-row spreadsheet.

As an example of the kind of hardware specifications you can expect to see, table 1.1 outlines the specifications of common iPhone, iPod Touch, and iPad models available in mid-2010. In general, the hardware specifications lag behind those of desktop PCs by a couple of years, but the number of integrated hardware accessories that your applications can take advantage of, such as camera, Bluetooth, and GPS, is substantially higher.

Table 1.1 Comparison of hardware specifications of various iPhone and iPod Touch devices

Feature	iPhone 3G	iPhone 3GS	iPhone 4	iPad	iPad2
RAM	128 MB	256 MB	512 MB	256 MB	512 MB
Flash	8–16 GB	16–32 GB	16–32 GB	16–64 GB	16–64 GB
Processor	412 MHz ARM11	600 MHz ARM Cortex	1 GHz Apple A4	1 GHz Apple A4	1 GHz dual-core Apple A5
Cellular	3.6 Mbps	7.2 Mbps	7.2 Mbps	7.2 Mbps (optional)	7.2 Mbps (optional)
Wi-Fi	Yes	Yes	Yes	Yes	Yes
Camera	2 MP	3 MP AF	5 MP AF (back) 0.3 MP (front)	—	0.92 MP (back) 0.3 MP (front)
Bluetooth	Yes	Yes	—	Yes	Yes
GPS	Yes (no compass)	Yes	—	Yes (3G models only)	Yes (3G models only)

Although it's nice to know the hardware capabilities and specifications of each device, application developers generally need not concern themselves with the details. New models will come and go as the iOS platform matures and evolves until it becomes difficult to keep track of all the possible variants.

Instead, you should strive to create an application that will adapt at runtime to the particular device it finds itself running on. Whenever you need to use a feature that's present only on a subset of devices, you should explicitly test for its presence and programmatically deal with it when it isn't available. For example, instead of checking if your application is running on an iPhone to determine if a camera is present, you would be better off checking whether a camera is present, because some models of iPad now come with cameras.

1.2.2 Expecting an unreliable internet connection

In this age of cloud computing, a number of iOS applications need connectivity to the internet. The iOS platform provides two main forms of wireless connectivity: local area in the form of 802.11 Wi-Fi and wide area in the form of various cellular data standards. These connection choices provide a wide variability in speed, ranging from 300 kilobits to 54 megabits per second. It's also possible for the connection to disappear altogether, such as when the user puts the device into flight mode, disables cellular roaming while overseas, or enters an elevator or tunnel.

Unlike on a desktop, where most developers assume a network connection is always present, good iOS applications must be designed to cope with network connectivity being unavailable for long periods of time or unexpectedly disconnecting. The worst user experience your customers can have is a “sorry, cannot connect to server” error message while running late to a meeting and needing to access important information that shouldn't require a working internet connection to obtain.

In general, it's important to constantly be aware of the environment in which your iOS application is running. Your development techniques may be shaped not only by the memory and processing constraints of the device but also by the way in which the user interacts with your application.

That's enough of the background information. Let's dive right in and create an iOS application!

1.3 Using Xcode to develop a simple Coin Toss game

Although you might have grand ideas for the next iTunes App Store smash, let's start with a relatively simple application that's easy to follow without getting stuck in too many technical details, allowing the unique features of the development tools to shine through. As the book progresses, we dig deeper into the finer points of everything demonstrated. For now the emphasis is on understanding the general process rather than the specifics of each technique.

The application you develop here is a simple game that simulates a coin toss, such as is often used to settle an argument or decide who gets to go first in a competition. The user interface is shown in figure 1.2 and consists of two buttons labeled Heads and Tails. Using these buttons, the user can request that a new coin toss be made and call the desired result. The iPhone simulates the coin toss and updates the screen to indicate if the user's choice is correct.

In developing this game, the first tool we need to investigate is Xcode.

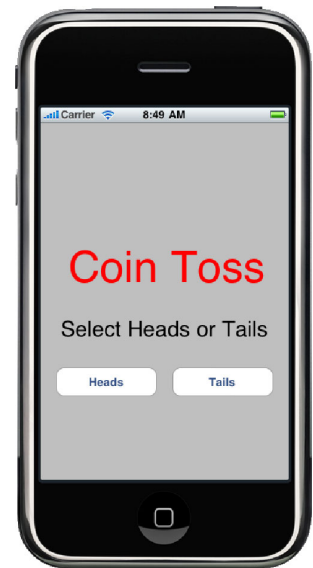


Figure 1.2 Coin Toss sample game

1.3.1 Introducing Xcode—Apple's IDE

As mentioned earlier in this chapter, Xcode is an IDE that provides a comprehensive set of features to enable you to manage the entire lifecycle of your software development project. Creating the initial project, defining your class or data model, editing your source code, building your application, and finally debugging and performance-tuning the resultant application are all tasks performed in Xcode.

Xcode is built on the foundation of several open source tools: LLVM (the open source Low-Level Virtual Machine), GCC (the GNU compiler), GDB (the GNU debugger), and DTrace (instrumentation and profiling by Sun Microsystems).

1.3.2 Launching Xcode easily

Once you install the iOS software development kit (SDK), the first challenge to using Xcode is locating the application. Unlike most applications that install in the /Applications folder, Apple separates developer-focused tools into the /Developer/Applications folder.

The easiest way to find Xcode is to use the Finder to open the root Macintosh HD folder (as shown in figure 1.3). From there, you can drill down into the Developer folder and finally the Applications subfolder. As a developer, you'll practically live within Xcode, so you may wish to put the Xcode icon onto your Dock or place the folder in the Finder sidebar for easy access.

Once you locate the /Developer/Applications folder, you should be able to easily locate and launch Xcode.

It's important to note that Xcode isn't your only option. Xcode provides all the features you require to develop applications out of the box, but that doesn't mean you can't complement it with your own tools. For example, if you have a favorite text editor in which you feel more productive, it's possible to configure Xcode to use your external text editor in favor of the built-in functionality. The truly masochistic among you could even revert to using makefiles and the command line.

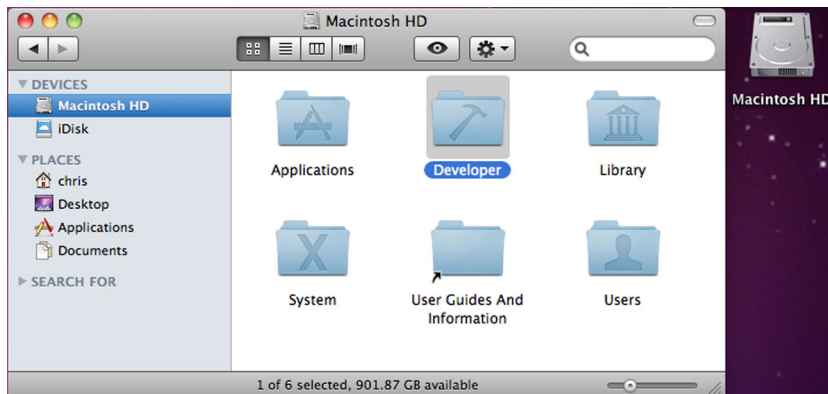


Figure 1.3 A Finder window showing the location of the Developer folder, which contains all iPhone developer-related tools and documentation

Help! I don't see the Xcode application

If you don't have a /Developer folder or you can't see any references to iPhone or iPad project templates when Xcode is launched, refer to appendix A for help on how to download and install the required software.

1.3.3 Creating the project

To create your first project, select the New Project option in the File menu (Shift-Command-N). Xcode displays a New Project dialog similar to the one displayed in figure 1.4.

Your first decision is to choose the type of project you want to create. This is done by selecting a template that determines the type of source code and settings Xcode will automatically add to get your project started.

For the Coin Toss game, you want the View-based Application template. You first select Application under the iOS header in the left pane, and then select View-based Application. Then click Next in the lower-right corner, which prompts you to name the project and allows you to specify the company identifier required to associate the application with your iOS Developer account. For this project, use the name CoinToss and enter a suitable company identifier.

Xcode uses the product name and company identifier values to produce what is called a *bundle identifier*. iOS uniquely identifies each application by this string. In

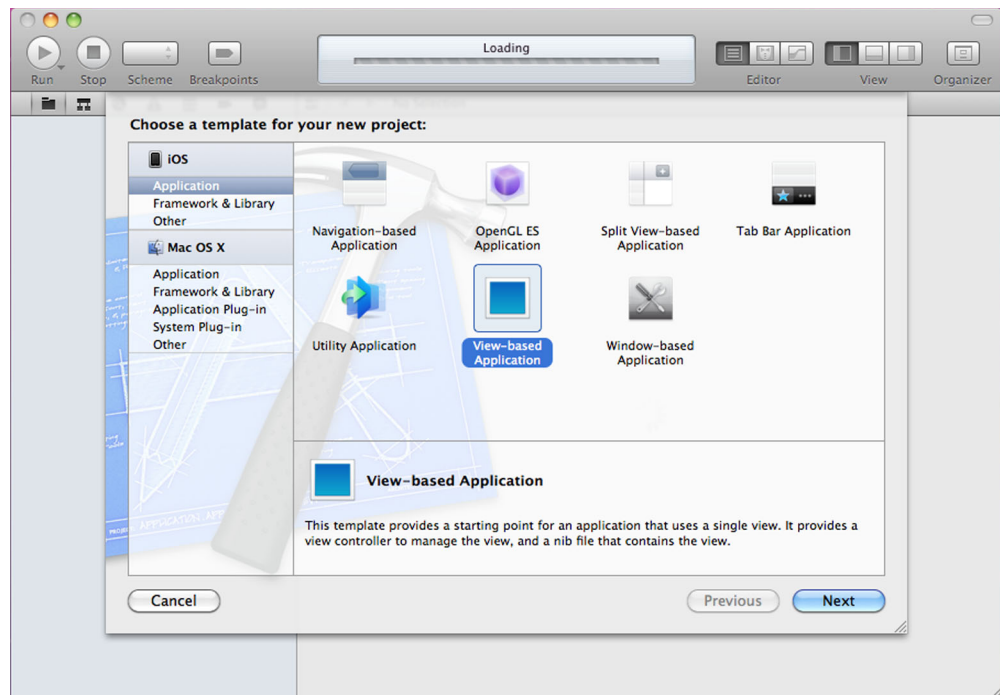


Figure 1.4 The New Project dialog in Xcode showing the View-based Application template

order for the operating system to allow the CoinToss game to run, its bundle identifier must match up with one included in a provisioning profile that's been installed on the device. If the device can't find a suitable profile, it refuses to run the application. This is how Apple controls with an iron fist which applications are allowed in its ecosystem. If you don't have a suitable company identifier or are unsure what to enter here, follow the instructions in appendix A before proceeding with the rest of this chapter.

Once all the details are entered, deselect the Include Unit Tests check box and click Next, which prompts you to select where you want the project and generated source code files to be saved.

Help! I don't see any iOS-related options

If you see no iOS-based templates in the New Project dialog, it's possible you haven't correctly installed the iOS SDK. The copy of Xcode you're running is probably from a Mac OS X Install DVD or perhaps was downloaded directly from the Apple Developer Connection (ADC) website and is suitable only for development of desktop applications.

Installing the iOS SDK as outlined in appendix A should replace your copy of Xcode with an updated version that includes support for iPhone and iPad development.

You may wonder what other kinds of projects you can create. Table 1.2 lists the most common iOS project templates. Which template you choose depends on the type of user interface you want your application to have. But don't get too hung up on template selection: the decision isn't as critical as you may think. Once your project is created,

Table 1.2 Project templates available in Xcode for creating a new iOS project

Project type	Description
Navigation-based Application	Creates an application similar in style to the built-in Contacts application with a navigation bar across the top.
OpenGL ES Application	Creates an OpenGL ES–based graphics application suitable for games and so on.
Split View–based Application	Creates an application similar in style to the built-in Mail application on the iPad. Designed to display master/detail-style information in a single screen.
Tab Bar Application	Creates an application similar in style to the built-in Clock application with a tab bar across the bottom.
Utility Application	Creates an application similar in style to the built-in Stocks and Weather applications, which flip over to reveal a second side.
View-based Application	Creates an application that consists of a single view. You can draw and respond to touch events from the custom view.
Window-based Application	Creates an application that consists of a single window onto which you can drag and drop controls.

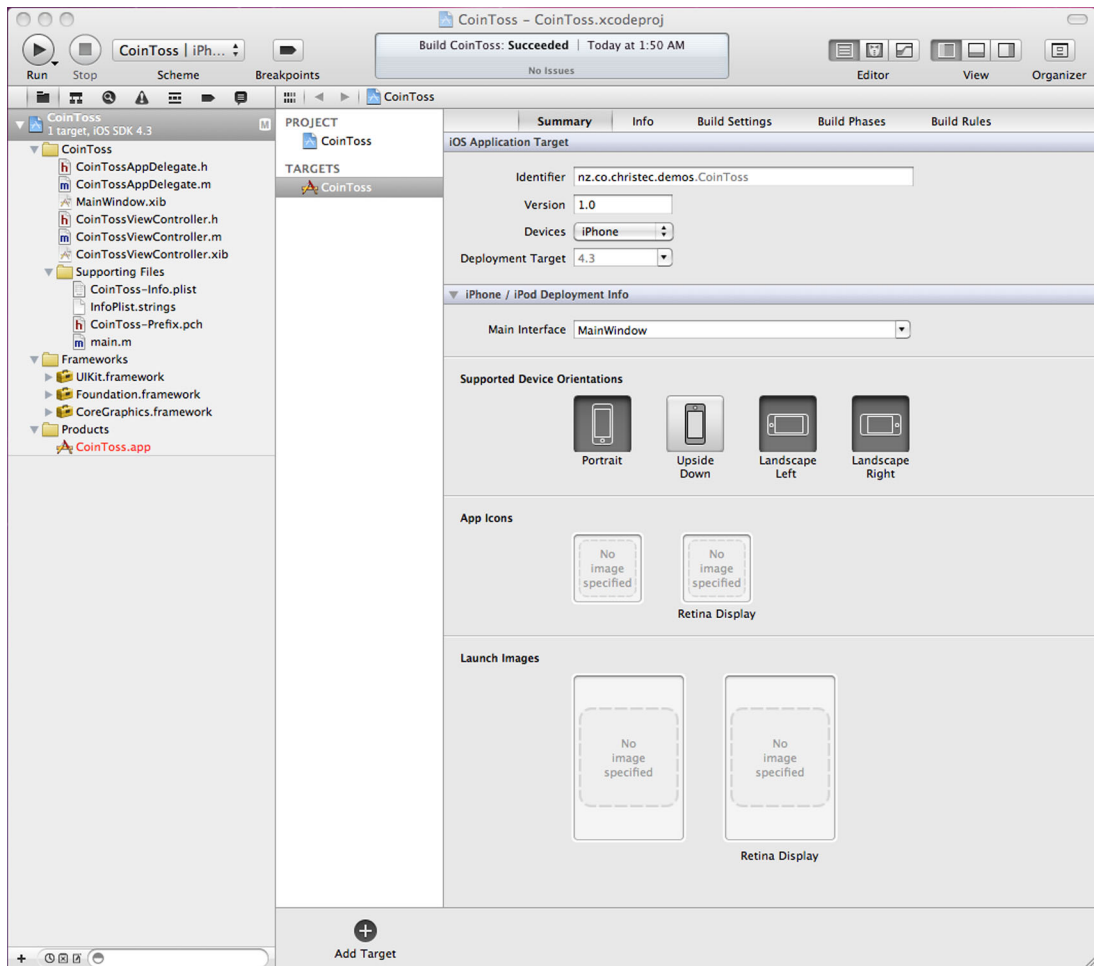


Figure 1.5 Main Xcode window with the CoinToss group fully expanded to show the project's various source code files

you can alter the style of your application—it just won't be as easy because you won't have the project template automatically inserting all of the required source code for you; you'll need to write it yourself.

Now that you've completed the New Project dialog, a project window similar to the one in figure 1.5 is displayed. This is Xcode's main window and consists of a Project Navigator pane on the left and a large, context-sensitive editor pane on the right.

The pane on the left lists all the files that make up your application. The group labeled CoinToss represents the entire game, and if you expand this node, you can drill down into smaller subgroups until you eventually reach the files that make up the project. You're free to create your own groupings to aid in organizing the files in any manner that suits you.

When you click a file in the left pane, the right pane updates to provide an editor suitable for the selected file. For *.h and *.m source code files, a traditional source code text editor is presented, but other file types (such as *.xib resource files) have more complex graphical editors associated with them.

Some groups in the left pane have special behaviors associated with them or don't represent files at all. For example, the items listed under the Frameworks group indicate pre-compiled code libraries that the current project makes use of.

As you become more comfortable with developing applications in Xcode, you'll become adept at exploring the various sections presented in the Project Navigator pane. To begin your discovery, let's write the source code for your first class.

1.3.4 *Writing the source code*

The View-based Application template provides enough source code to get a basic game displayed on the iPhone—so basic, in fact, that if you ran the game right now, you would simply see a gray rectangle on the screen.

Let's start implementing the game by opening the CoinTossViewController.h file in the Xcode window and using the text editor to replace the contents with the following listing.

Listing 1.1 CoinTossViewController.h

```
#import <UIKit/UIKit.h>

@interface CoinTossViewController : UIViewController {
    UILabel *status;
    UILabel *result;
}

@property (nonatomic, retain) IBOutlet UILabel *status;
@property (nonatomic, retain) IBOutlet UILabel *result;

- (IBAction)callHeads;
- (IBAction)callTails;

@end
```

Don't worry if the contents of listing 1.1 don't make much sense to you. At this stage, it's not important to understand the full meaning of this code. Learning these sorts of details is what the rest of the book is designed for—all will be revealed in time!

For now, let's focus on understanding the general structure of an Objective-C-based project. Objective-C is an object-oriented language, meaning that a large portion of your coding efforts will be spent defining new classes (types of objects). Listing 1.1 defines a new class called CoinTossViewController. By convention, the definition of a class is kept in a header file that uses a *.h file extension.

In the CoinTossViewController header file, the first two lines declare that the class stores the details of two UILabel controls located somewhere in the user interface. A UILabel can display a single line of text, and you use these labels to display the results of the coin toss.

The second set of statements allows code external to this class to tell you which specific UILabels you should be using. Finally, you specify that your class responds to two messages called `callHeads` and `callTails`. These messages inform you when the user has called heads or tails and a new coin toss should be initiated.

A header (*.h) file specifies what you can expect a class to contain and how other code should interact with it. Now that you've updated the header file, you must provide the actual implementation of the features you've specified. Open the matching `CoinTossViewController.m` file, and replace its contents with that of the following listing.

Listing 1.2 CoinTossViewController.m

```
#import "CoinTossViewController.h"
#import <QuartzCore/QuartzCore.h>

@implementation CoinTossViewController

@synthesize status, result;

- (void) simulateCoinToss:(BOOL)userCalledHeads {
    BOOL coinLandedOnHeads = (arc4random() % 2) == 0;

    result.text = coinLandedOnHeads ? @"Heads" : @"Tails";

    if (coinLandedOnHeads == userCalledHeads)
        status.text = @"Correct!";
    else
        status.text = @"Wrong!";

    CABasicAnimation *rotation = [CABasicAnimation
        animationWithKeyPath:@"transform.rotation"];
    rotation.timingFunction = [CAMediaTimingFunction
        functionName:kCAMediaTimingFunctionEaseInEaseOut];
    rotation.fromValue = [NSNumber numberWithFloat:0.0f];
    rotation.toValue = [NSNumber numberWithFloat:720 * M_PI / 180.0f];
    rotation.duration = 2.0f;
    [status.layer addAnimation:rotation forKey:@"rotate"];

    CABasicAnimation *fade = [CABasicAnimation
        animationWithKeyPath:@"opacity"];
    fade.timingFunction = [CAMediaTimingFunction
        functionName:kCAMediaTimingFunctionEaseInEaseOut];
    fade.fromValue = [NSNumber numberWithFloat:0.0f];
    fade.toValue = [NSNumber numberWithFloat:1.0f];
    fade.duration = 3.5f;
    [status.layer addAnimation:fade forKey:@"fade"];
}

- (IBAction) callHeads {
    [self simulateCoinToss:YES];
}

- (IBAction) callTails {
    [self simulateCoinToss:NO];
}

- (void) viewDidLoad {
    self.status = nil;
}
```

1 Match with @property

2 Set up two objects

3 Affect the label

```

    self.result = nil;
}

- (void) dealloc {
    [status release];
    [result release];
    [super dealloc];
}

@end

```

4 Memory management

Listing 1.2 at first appears long and scary looking, but when broken down into individual steps, it's relatively straightforward to understand.

The first statement **1** matches up with the `@property` declarations in `CoinTossViewController.h`. The concept of properties and the advantage of synthesized ones in particular are explored in depth in chapter 5.

Most of the logic in the `CoinTossViewController.m` file is contained in the `simulateCoinToss:` method, which is called whenever the user wants the result of a new coin toss. The first line simulates a coin toss by generating a random number between 0 and 1 to represent heads and tails respectively. The result is stored in a variable called `coinLandedOnHeads`.

Once the coin toss result is determined, the two `UILabel` controls in the user interface are updated to match. The first conditional statement updates the result label to indicate if the simulated coin toss landed on heads or tails; the second statement indicates if the user correctly called the coin toss.

The rest of the code in the `simulateCoinToss:` method sets up two `CABasicAnimation` objects **2**, **3** to cause the label displaying the status of the coin toss to spin into place and fade in over time rather than abruptly updating. It does this by requesting that the `transform.rotation` property of the `UILabel` control smoothly rotate from 0 degrees to 720 degrees over 2.0 seconds, while the `opacity` property fades in from 0% (0.0) to 100% (1.0) over 3.5 seconds. It's important to note that these animations are performed in a declarative manner. You specify the change or effect you desire and leave it up to the framework to worry about any timing- and redrawing-related logic required to implement those effects.

The `simulateCoinToss:` method expects a single parameter called `userCalledHeads`, which indicates if the user expects the coin toss to result in heads or tails. Two additional methods, `callHeads` and `callTails`, are simple convenience methods that call `simulateCoinToss:` with the `userCalledHeads` parameter set as expected.

The final method, called `dealloc` **4**, deals with memory management-related issues. We discuss memory management in far greater depth in chapter 9. The important thing to note is that Objective-C doesn't automatically garbage collect unused memory (at least as far as the current iPhone is concerned). This means if you allocate memory or system resources, you're also responsible for releasing (or deallocating) it. Not doing so will cause your application to artificially consume more resources than it needs, and in the worst case, you'll exhaust the device's limited resources and cause the application to crash.

Now that you have the basic logic of the game developed, you must create the user interface in Xcode and connect it back to the code in the `CoinTossViewController` class.

1.4 Hooking up the user interface

At this stage, you can determine from the `CoinTossViewController` class definition that the user interface should have at least two `UILabel` controls and that it should invoke the `callHeads` or `callTails` messages whenever the user wants to call the result of a new coin toss. You haven't yet specified where on the screen the labels should be positioned or how the user requests that a coin toss be made.

There are two ways to specify this kind of detail. The first is to write source code that creates the user interface controls, configures their properties such as font size and color, and positions them onscreen. This code can be time consuming to write, and you can spend a lot of your time trying to visualize how things look onscreen.

A better alternative is to use Xcode, which allows you to visually lay out and configure your user interface controls and connect them to your source code. Most iOS project templates use this technique and typically include one or more `*.xib` files designed to visually describe the user interface. This project is no exception, so click the `CoinTossViewController.xib` file in the Project Navigator pane and notice that the editor pane displays the contents of the file (figure 1.6).

Along the left edge of the editor pane are some icons. Each icon represents an object that's created when the game runs, and each has a tooltip that displays its name. The wireframe box labeled `File's Owner` represents an instance of the `CoinTossViewController` class; the white rectangle represents the main view (or screen) of the application. Using Xcode, you can graphically configure the properties of these objects and create connections between them.

1.4.1 Adding controls to a view

The first step in defining the user interface for your game is to position the required user interface controls onto the view.

To add controls, find them in the Library window, which contains a catalog of available user interface controls, and drag and drop them onto the view. If the Library window isn't visible, you can open it via the `View > Utilities > Object Library` menu option (`Control-Option-Command-3`). For the Coin Toss game, you require two Labels and two Rounded Rect Buttons, so drag two of each control onto the view. The process of dragging and dropping a control onto the view is shown in figure 1.7.

After you drag and drop the controls onto the view, you can resize and adjust their positions to suit your aesthetics. The easiest way to change the text displayed on a button or label control is to double-click the control and begin typing. To alter other properties, such as font size and color, you can use the Attributes Inspector pane, which can be displayed via the `View > Utilities > Attributes Inspector` menu option (`Alt-Command-4`). While styling your view, you can refer back to figure 1.2 for guidance.

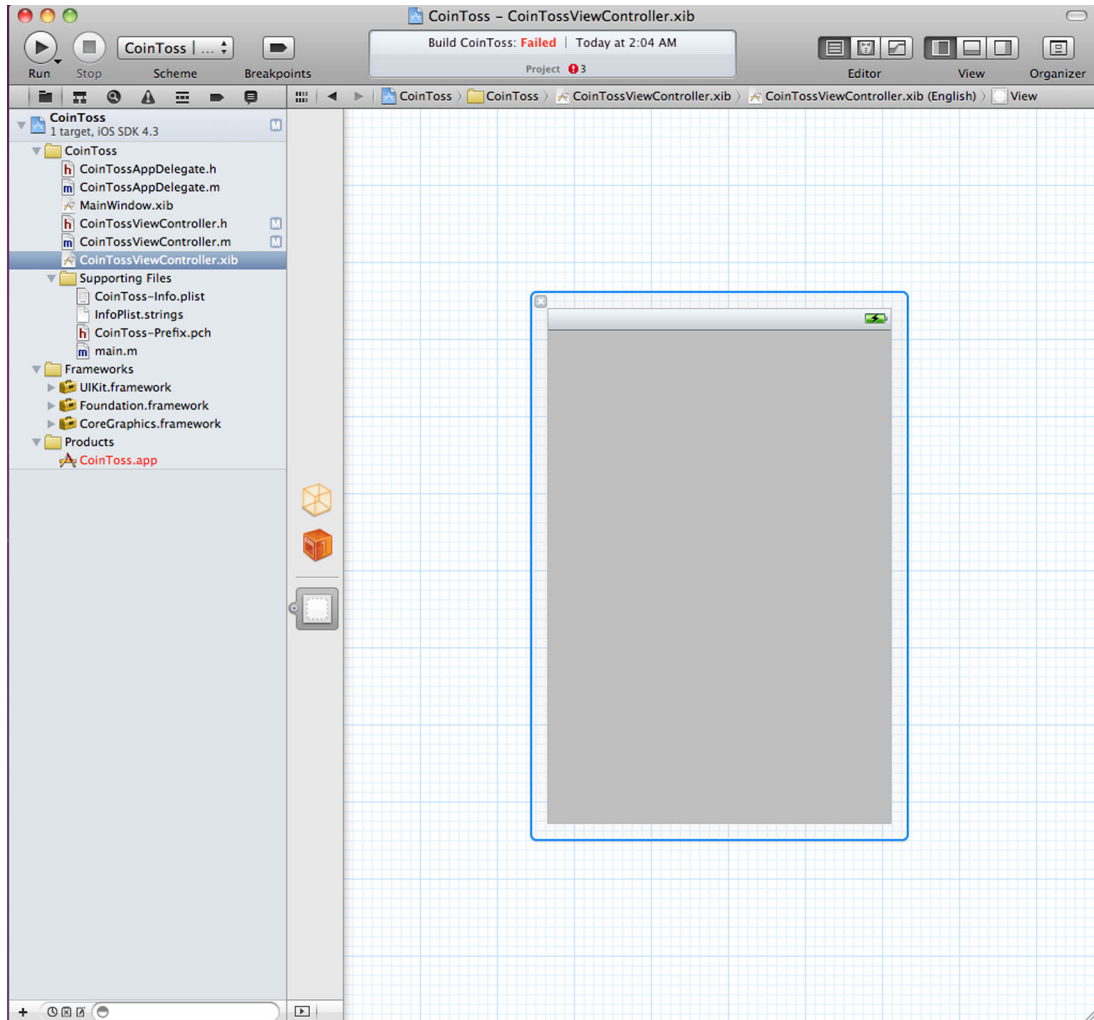


Figure 1.6 The main Xcode window demonstrating the editing of a *.xib file. Along the left edge of the editor you can see three icons, each representing a different object or GUI component stored in the *.xib file.

With the controls positioned on the user interface, the only task left is to connect them to the code you previously wrote. Remember that the class defined in the `CoinTossViewController.h` header file requires three things from the user interface:

- Something to send the `callHeads` or `callTails` messages whenever the user wishes to initiate a new coin toss
- A `UILabel` to display the results of the latest coin toss (heads or tails)
- A `UILabel` to display the status of the latest coin toss (correct or incorrect)

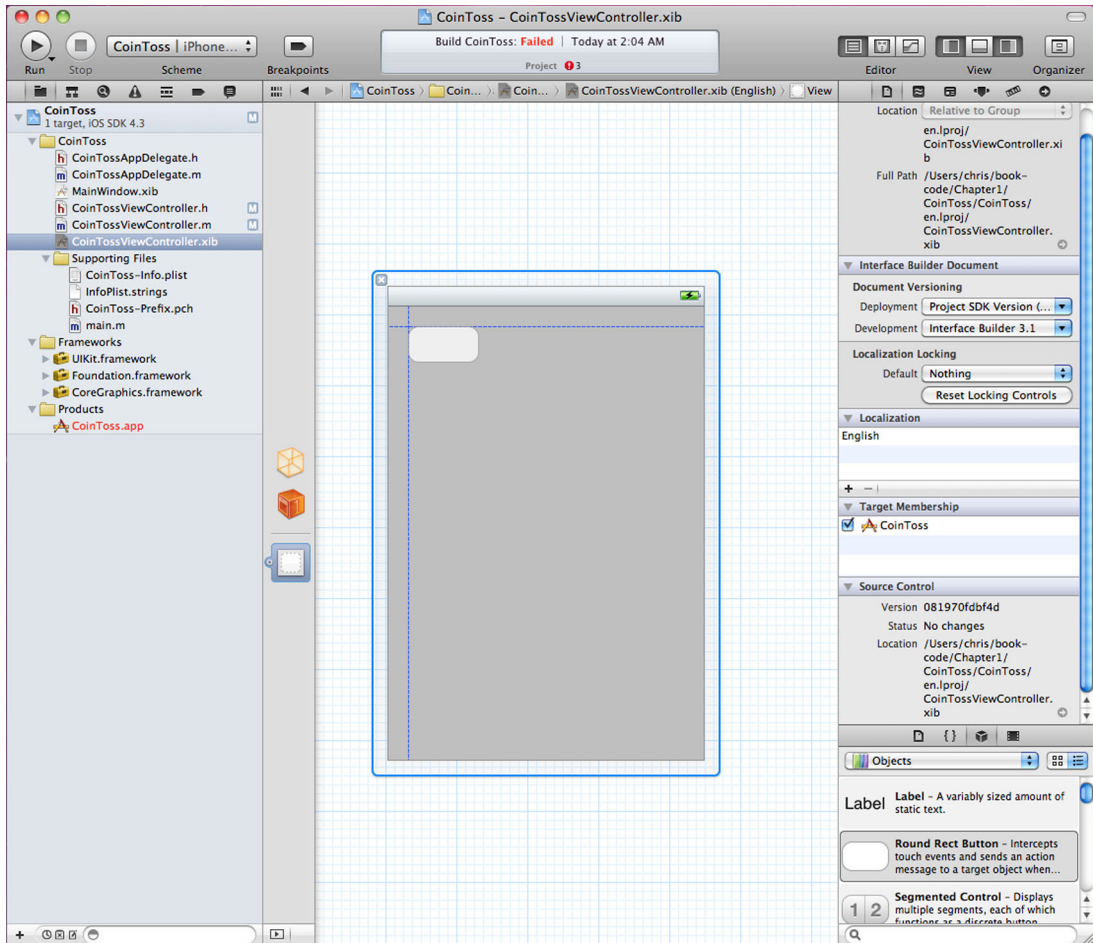


Figure 1.7 Dragging and dropping new controls onto the view. Notice the snap lines, which help ensure your user interface conforms to the iOS Human Interface Guidelines (HIG).

1.4.2 Connecting controls to source code

The user interface you just created meets these requirements, but the code can't determine which button should indicate that the user calls heads or tails (even if the text on the buttons makes it inherently obvious to a human). Instead, you must explicitly establish these connections. Xcode allows you to do so graphically.

Hold down the Control key and drag the button labeled Heads toward the icon representing the `CoinTossViewController` instance (File's Owner) located on the left edge of the editor. As you drag, a blue line should appear between the two elements.

When you let go of the mouse, a pop-up menu appears that allows you to select which message should be sent to the `CoinTossViewController` object whenever the

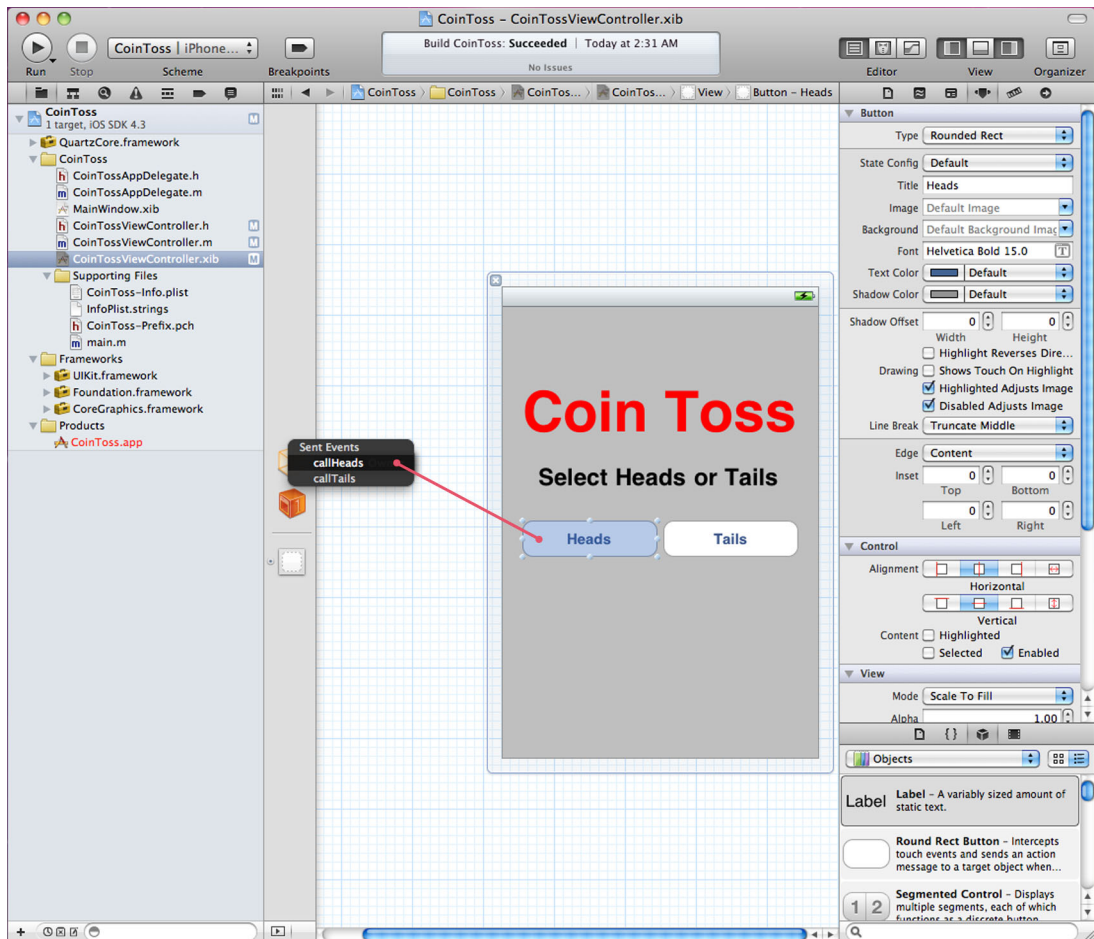


Figure 1.8 Visually forming a connection between the button control and the `CoinTossViewController` class by dragging and dropping between items

button is tapped, as shown in figure 1.8. In this case, you select `callHeads` because this is the message that matches the intent of the button.

You can repeat this process to connect the button labeled `Tails` to the `callTails` method. Making these two connections means that tapping either of the buttons in the user interface will cause the execution of logic in the `CoinTossViewController` class. Having these connections specified graphically rather than programmatically is a flexible approach because it enables you to quickly and easily try out different user interface concepts by swapping controls around and reconnecting them to the class.

If Xcode refuses to make a connection between a user interface control and an object, the most probable cause is a source code error, such as a simple typo or incorrect data type. In this case, make sure the application still compiles, and correct any errors that appear before retrying the connection.

With the buttons taken care of, you're left with connecting the label controls to the `CoinTossViewController` class to allow the code to update the user interface with the results of the latest coin toss.

To connect the label controls, you can use a similar drag-and-drop operation. This time, while holding down the Control key, click the icon representing the `CoinTossViewController` instance and drag it toward the label in the view. When you release the mouse, a pop-up menu appears that allows you to select which property of the `CoinTossViewController` class you want to connect the label control to. This process is demonstrated in figure 1.9. Using this process, connect the label titled Coin Toss to the `status` property and the label titled Select Heads or Tails to the `result` property.

When deciding which way you should form connections between objects, consider the flow of information. In the case of the button, tapping the button causes a method

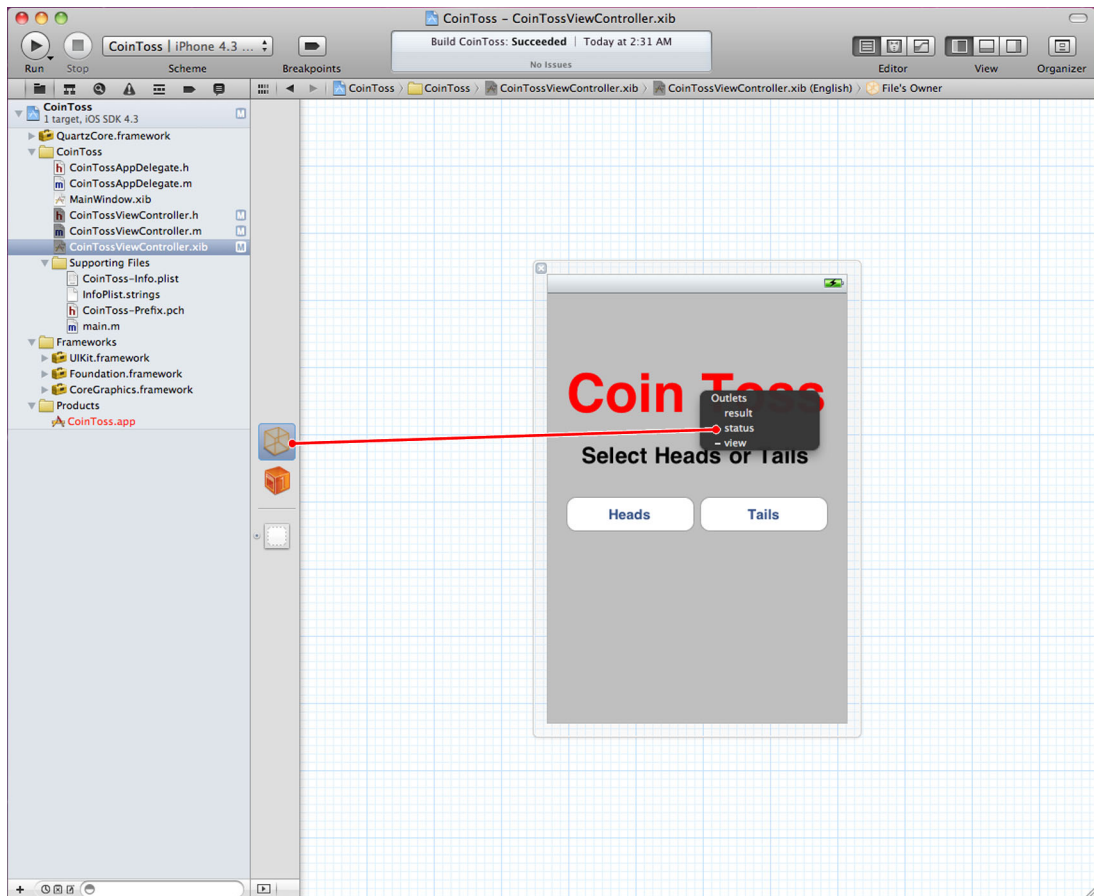


Figure 1.9 Visually forming a connection between the status instance variable and the label control in the user interface by dragging and dropping between the items (with the Control key held down)

in the application to be executed, whereas in the case of connecting the label, changing the value of the instance variable in the class should update the user interface.

You may wonder how Xcode determines which items to display in the pop-up menu. If you refer back to listing 1.1, the answer can be seen by way of the special `IBOutlet` and `IBAction` keywords. Xcode parses your source code and allows you to connect the user interface to anything marked with one of these special attributes.

At this stage, you may like to verify that you've correctly made the required connections. If you hold down the Control key and click the icon representing the `CoinTossViewController` instance, a pop-up menu appears allowing you to review how all the outlets and actions associated with an object are connected. If you hover the mouse over one of the connections, Xcode even highlights the associated object. This feature is shown in figure 1.10.

At this stage you're done with the user interface. You're now ready to kick the tires, check if you've made mistakes, and see how well your game runs.

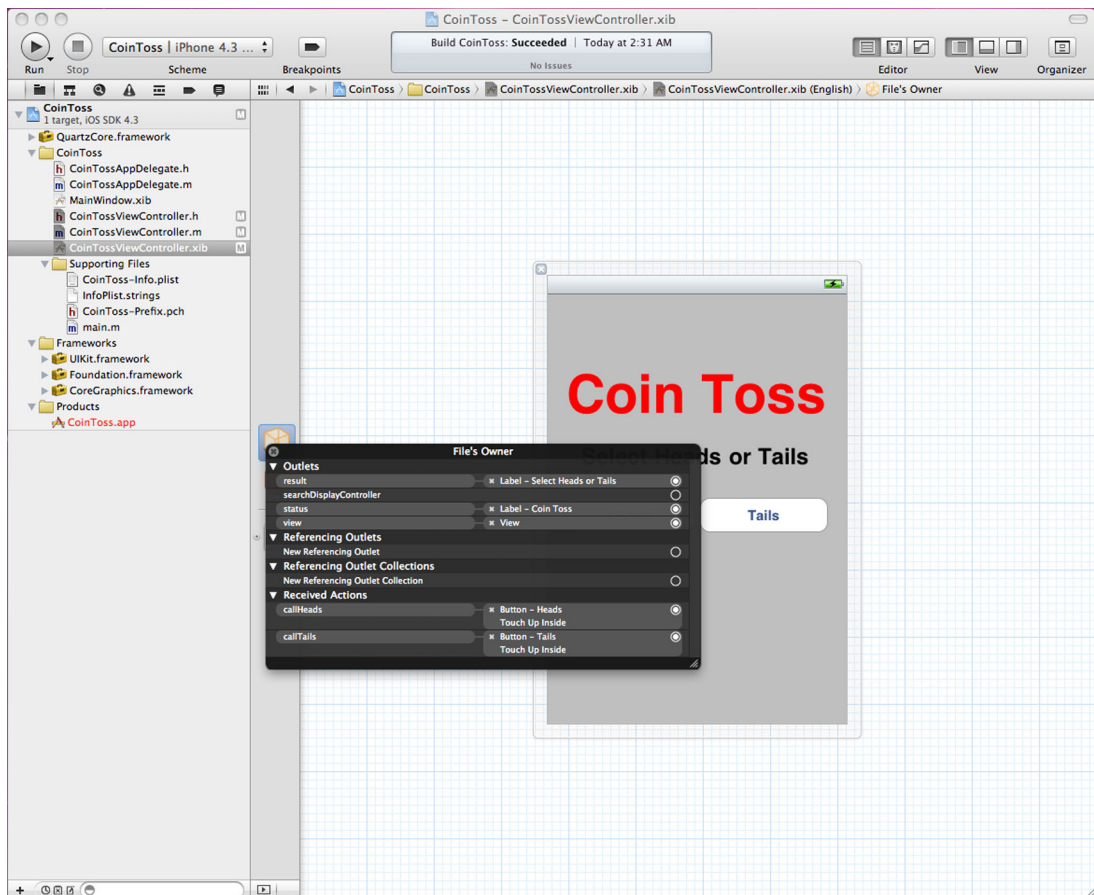


Figure 1.10 Reviewing connections made to and from the `CoinTossViewController` object

NIBs vs. XIBs

The user interface for an iOS application is stored in a .xib file. But in the documentation and Cocoa Touch frameworks, these files are commonly called *nibs*.

These terms are used pretty interchangeably: a .xib file uses a newer XML-based file format, which makes the file easier to store in revision control systems and so on.

A .nib, on the other hand, is an older binary format, which leads to more efficient file sizes, parsing speed, and so on.

The documentation commonly refers to NIB files instead of XIB files because, as Xcode builds your project, it automatically converts your *.xib files into the *.nib format.

1.5 Compiling the Coin Toss game

Now that you've finished coding your application, you need to convert the source code into a form useable by the iPhone. This process is called *compilation*, or *building* the project. To build the game, select Build from the Product menu (or press Cmd-B).

While the project is building, you can keep track of the compiler's progress by looking at the progress indicator in the middle of the toolbar. It should read "Build CoinToss: Succeeded." If you've made mistakes, you'll see a message similar to "Build CoinToss: Failed." In this case, clicking the red exclamation icon underneath the text (or pressing Cmd-4) displays a list of errors and warnings for you to resolve.

Clicking an error in this list displays the matching source code file with the lines containing errors highlighted, as illustrated in figure 1.11. After correcting the problem, you can build the application again, repeating this process until all issues are resolved.

When you compile the Coin Toss game, you should notice errors mentioning `kCAMediaTimingFunctionEaseInEaseOut`, `CAMediaTimingFunction`, and `CABasicAnimation`. To correct these errors, select the CoinToss project in the Project Navigator (topmost item in the tree view). In the editor that appears for this item, switch to the Build Phases tab and expand the Link Binary with Libraries section. The expanded region displays a list of additional frameworks that your application requires. For the user interface animations to work, you need to click the + button at the bottom of the window and select `QuartzCore.framework` from the list that appears.

To keep things tidy, once you add the `QuartzCore` framework reference, you may prefer to move it within the project navigator tree view so that it's located under the Frameworks section, alongside the other frameworks on which your application depends.

1.6 Taking Coin Toss for a test run

Now that you've compiled the game and corrected any obvious compilation errors, you're ready to verify that it operates correctly. You could run the game and wait for it to behave incorrectly or crash, but that would be rather slow going, and you would have to guess at what was happening internally. To improve this situation, Xcode provides

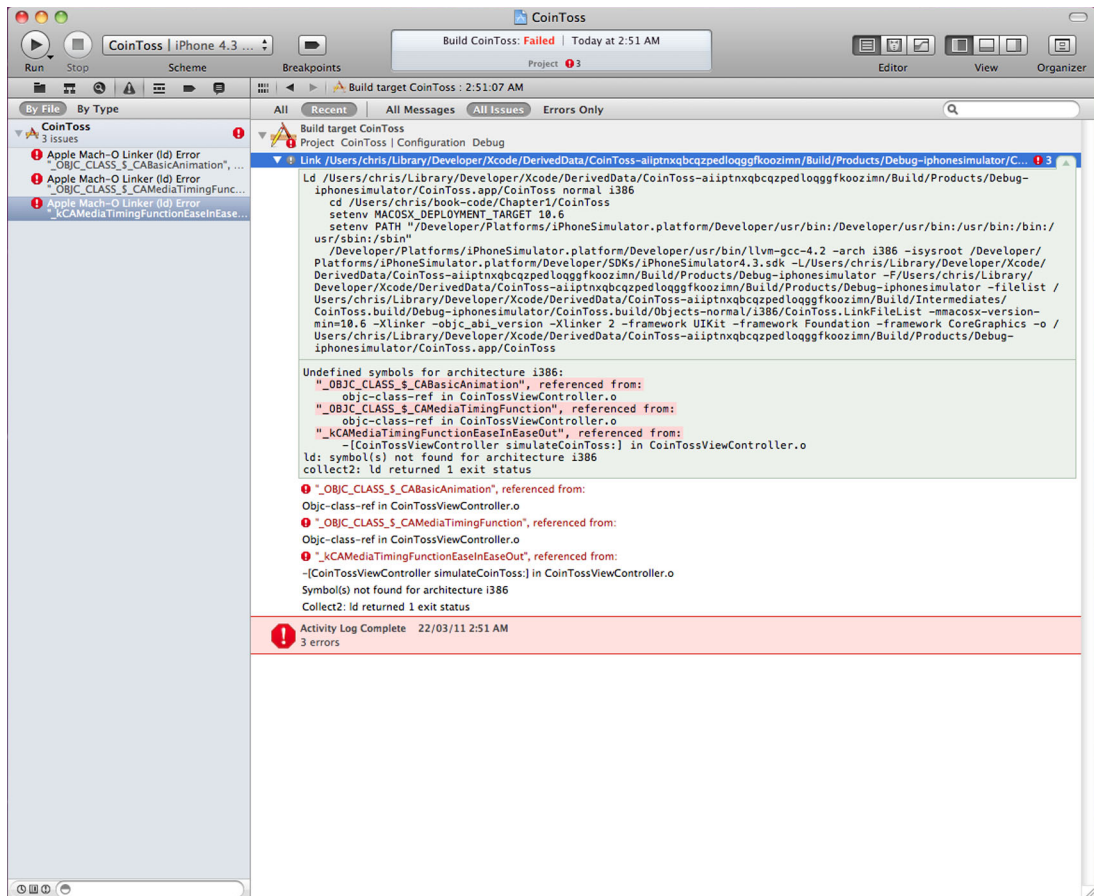


Figure 1.11 Xcode's text editor visually highlights lines of source code with compilation errors. After correcting any errors, building the project will indicate if you have successfully corrected the problem.

an integrated debugger that hooks into the execution of your application and allows you to temporarily pause it to observe the value of variables and step through source code line by line. But before you learn how to use it, we must take a slight detour.

1.6.1 Selecting a destination

Before testing your application, you must decide where you want to run it. During initial development, you'll commonly test your application via the iOS Simulator. The simulator is a pretend iPhone or iPad device that runs in a window on your desktop Mac OS X machine. Using the simulator can speed up application development because it's a lot quicker for Xcode to transfer and debug your application in the simulator than it is to work with a real iPhone.

Developers with experience in other mobile platforms may be familiar with the use of device emulators. The terms *simulator* and *emulator* aren't synonymous. Unlike an

Always test on a real iPhone, iPod Touch, or iPad device

The code samples in this book are designed to run in the iOS Simulator. This is a quick and easy way to iteratively develop your application without worrying about device connectivity or the delay involved in transferring the application to a real device.

Because the iOS Simulator isn't a perfect replica of an iPhone, it's possible for an application to work in the simulator but fail on an actual device. Never publish an application to the iTunes App Store that hasn't been tested on a real device, or better yet, try to test your application out on a few variants, such as the iPhone and iPod Touch.

emulator that attempts to emulate the device at the hardware level (and hence can run virtually identical firmware to a real device), a simulator only attempts to provide an environment that has a compatible set of APIs.

The iOS Simulator runs your application on the copy of Mac OS X used by your desktop, which means that differences between the simulator and a real iPhone occasionally creep in. A simple example of where the simulation “leaks” is filenames. In the iOS Simulator, filenames are typically case insensitive, whereas on a real iPhone, they're case sensitive.

By default, most project templates are configured to deploy your application to the iOS Simulator. To deploy your application to a real iPhone, you must change the destination from iPhone Simulator to iOS Device. The easiest way to achieve this is to select the desired target in the drop-down menu found toward the left of the toolbar in the main Xcode window, as shown in figure 1.12.

Changing the destination to iOS Device ensures that Xcode attempts to deploy the application to your real iPhone, but an additional change is needed before this will succeed.

1.6.2 Using breakpoints to inspect the state of a running application

While testing an application, it's common to want to investigate the behavior of a specific section of source code. Before you launch the application, it can be handy to configure the debugger to automatically pause execution whenever these points are reached. You can achieve this through the use of a feature called *breakpoints*.

A breakpoint indicates to the debugger a point in the source code where the user would like to automatically “break into” the debugger to explore the current value of variables, and so on.

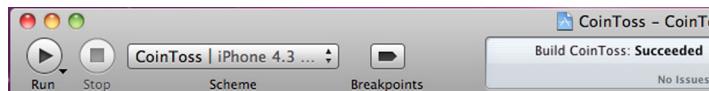


Figure 1.12 The top-left corner of the main Xcode window. Selecting the CoinToss | iPhone 4.3 Simulator drop-down menu allows you to switch between iPhone Simulator and iOS Device.

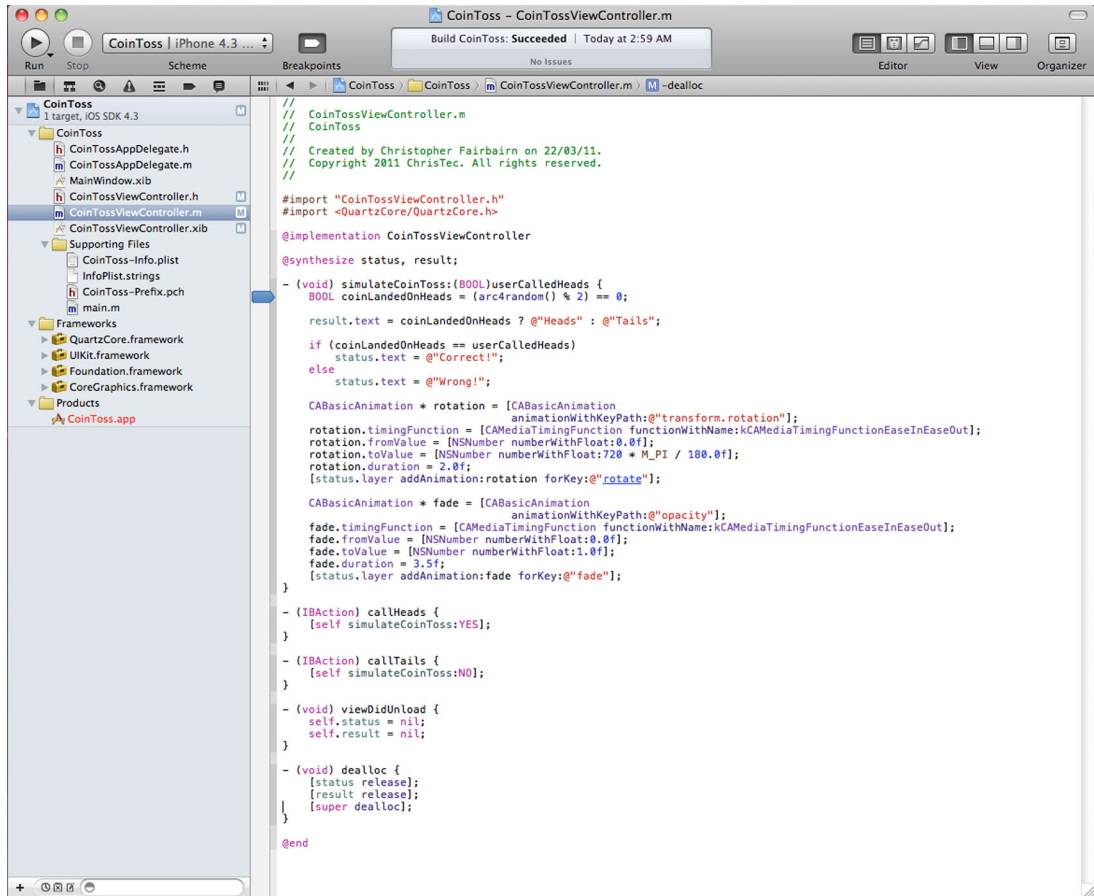


Figure 1.13 Setting a breakpoint to break into the debugger whenever the first line of the `simulateCoinToss:` method is called. Notice the arrow in the margin indicating an active breakpoint.

For the Coin Toss game, let's add a breakpoint to the start of the `simulateCoinToss:` method. Open the `CoinTossViewContoller.m` file and scroll down to the source code implementing the `simulateCoinToss:` method. If you then click the left margin beside the first line, you should see a little blue arrow appear, as shown in figure 1.13.

The blue arrow indicates that this line has an enabled breakpoint. If you click the breakpoint, it becomes a lighter shade of blue, indicating a disabled breakpoint, which causes the debugger to ignore it until it's clicked again to re-enable it. To permanently remove a breakpoint, click and drag the breakpoint away from the margin. Releasing the mouse will show a little "poof" animation, and the breakpoint will be removed.

1.6.3 Running the CoinToss game in the iPhone simulator

With the breakpoint in place, you're finally ready to run the application and see it in action. Select Run from the Product menu (Cmd-R). After a few seconds, the application

will appear on your iPhone. All that hard work has finally paid off. Congratulations—you're now officially an iPhone developer!

If you want to run the game but don't want any of your breakpoints to be enabled, you can click each one to disable them individually, but this would take a while, and you would need to manually re-enable all the breakpoints if you wanted to use them again. As a handy alternative, you can temporarily disable all breakpoints by selecting Product > Debug > Deactivate Breakpoints (Cmd-Y).

1.6.4 Controlling the debugger

Now that you've seen your first iPhone application running, you'll have undoubtedly felt the urge and tapped one of the buttons labeled Heads or Tails. When you tap a button, notice that the Xcode window jumps to the foreground. This is because the debugger has detected that execution of the application has reached the point where you inserted a breakpoint.

The Xcode window that appears should look similar to the one in figure 1.14. Notice that the main pane of the Xcode window displays the source code of the currently executing method. Hovering the mouse over a variable in the source code displays a data tip showing the variable's current value. The line of source code that's about to be executed is highlighted, and a green arrow in the right margin points at it.

While the debugger is running, you'll notice the left pane of the Xcode window switches to display the call stack of each thread in the application. The call stack lists the order in which currently executing methods have been called, with the current method listed at the top. Many of the methods listed will be gray, indicating that source code isn't available for them, in this case because most are internal details of the Cocoa Touch framework.

A new pane at the bottom of the screen is also displayed; it shows the current values of any variables and arguments relevant to the current position of the debugger as well as any textual output from the debugger (see figure 1.14).

Along the top of the bottom debug pane, you may notice a series of small toolbar buttons similar to those shown in figure 1.15.

These toolbar options enable you to control the debugger and become important when the debugger pauses the application or stops at a breakpoint. These toolbar buttons (which may not all be present at all points in time) allow you to perform the following actions:

- *Hide*—Hide the debugger's console window and variables pane to maximize the screen real estate offered to the text editor.
- *Pause*—Immediately pause the iPhone application and enter the debugger.
- *Continue*—Run the application until another breakpoint is hit.
- *Step Over*—Execute the next line of code and return to the debugger.
- *Step Into*—Execute the next line of code and return to the debugger. If the line calls any methods, step through their code as well.
- *Step Out*—Continue executing code until the current method returns.

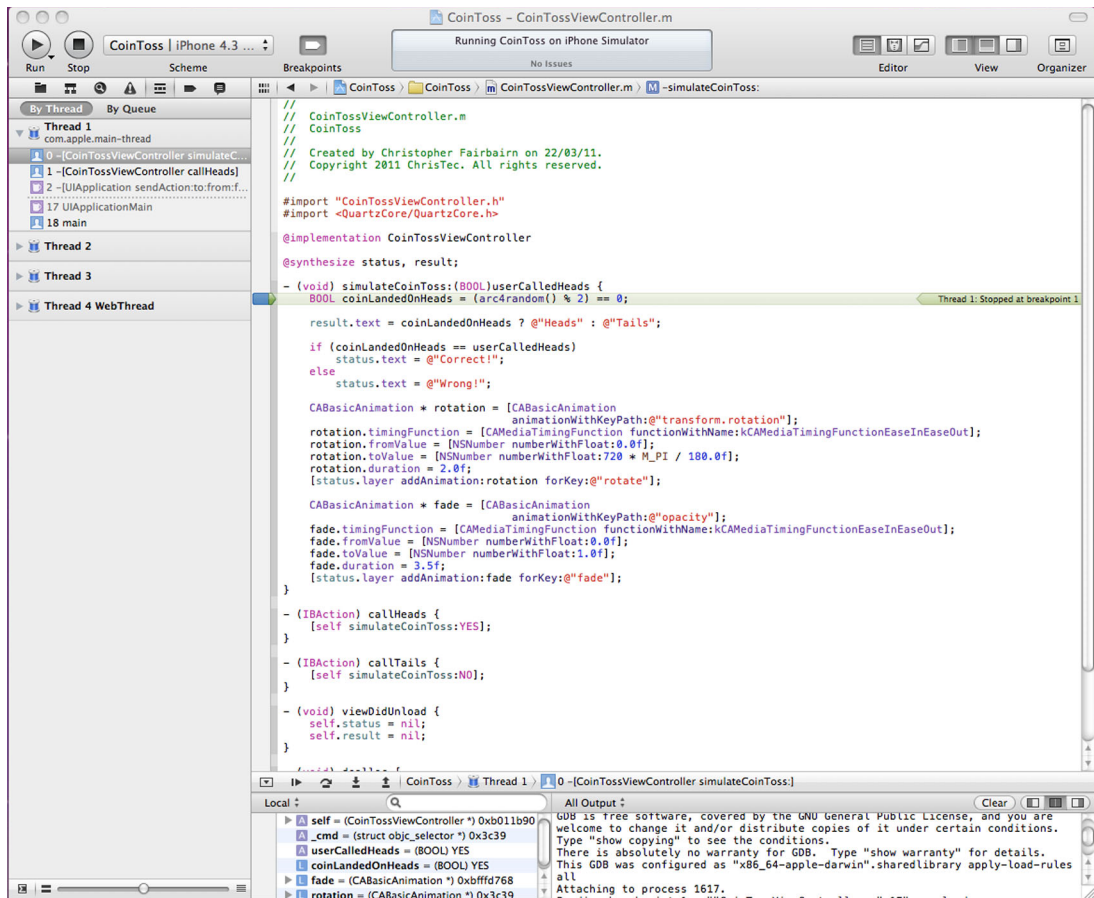


Figure 1.14 The Xcode debugger window after execution has reached a breakpoint

Your breakpoint caused the debugger to pause execution at the beginning of a simulated coin toss. If you view the variables pane or hover the mouse over the `userCalledHeads` argument, you can determine if the user has called heads (YES) or tails (NO).

The first line of the `simulateCoinToss:` method simulates flipping a coin (by selecting a random number, 0 or 1). Currently, the debugger is sitting on this line (indicated by the red arrow in the margin), and the statements on this line haven't been executed.

To request that the debugger execute a single line of source code and then return to the debugger, you can click the Step Over button to “step over” the next line of source code. This causes the coin toss to be simulated, and the red arrow should jump down to the next line that contains source code. At this stage, you can determine the

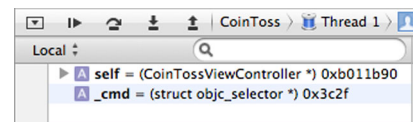


Figure 1.15 The toolbar options in Xcode for controlling the debugger

result of the coin toss by hovering the mouse over the `coinLandedOnHeads` variable name; once again, `YES` means heads and `NO` means tails.

Using the step-over feature a couple more times, you can step over the two `if` statements, which update the result and status `UILabels` in the user interface. Unlike what you may expect, however, if you check the iPhone device at this stage, the labels won't have updated! This is because of how the internals of Cocoa Touch operate: the screen will only update once you release the debugger and allow this method to return to the operating system.

To allow the iPhone to update the user interface and to see the fancy animations that herald in a new coin toss result, you can click Continue (or press `Cmd-Option-P`) to allow the application to continue execution until it hits another breakpoint or you explicitly pause it again. Taking a look at the iPhone, you should see that the results of the coin toss are finally displayed onscreen.

1.7 Summary

Congratulations, you've developed your first iPhone application! Show your friends and family. It may not be the next iTunes App Store blockbuster release, but while putting together this application, you've mastered many of the important features of the Xcode IDE, so you're well on your way to achieving success.

Although Objective-C is a powerful language with many capabilities, you'll find using visual tools such as Xcode can lead to a productivity boost, especially during initial prototyping of your application. The decoupling of application logic from how it's presented to the user is a powerful mechanism that shouldn't be underestimated. It's doubtful the first user interface you design for your application will be perfect, and being able to alter it without having to modify a single line of code is a powerful advantage.

By the same token, you were able to rely on the Cocoa Touch framework to handle the minutiae of how to implement many of the features of your game. For example, the animations were implemented in a fairly declarative manner: you specified starting and stopping points for the rotations and fading operations and left the Quartz Core framework to worry about the specifics of redrawing the screen, transitioning the animation, and speeding up or slowing down as the effect completed.

As you'll continue to see, there's great power in the Cocoa Touch frameworks. If you find yourself writing a vast amount of code for a particular feature, chances are you aren't taking maximum advantage of what Cocoa has to offer.

In chapter 2, we dive into data types, variables, and constants and are introduced to the Rental Manager application that you'll build throughout this book.

Data types, variables, and constants

This chapter covers

- Storing numeric-, logic-, and text-based data
- Creating your own data types
- Converting values between different data types
- Formatting values for presentation
- Introducing the Rental Manager sample application

Virtually every application needs to store, represent, or process data of some kind, whether a list of calendar appointments, the current weather conditions in New York, or the high scores of a game.

Because Objective-C is a statically typed language, whenever you declare a variable, you must also specify the type of data you expect to store in it. As an example, the following variable declaration declares a variable named `zombieCount`, which is of type `int`:

```
int zombieCount;
```

Short for *integer*, `int` is a data type capable of storing a whole number between `-2,147,483,648` and `+2,147,483,647`. In this chapter, you'll discover many data

types that can be used to store a wide range of real-world values. But before we dive in too far, let's introduce the Rental Manager application that we build over the course of this book.

2.1 Introducing the Rental Manager application

Each chapter in this book reinforces what you learned previously by applying the theory to a larger sample application. This application is designed to manage rental properties in a rental portfolio and to display details such as location, property type, weekly rent, and a list of current tenants. Figure 2.1 shows what the application looks like by the time you reach the end of this book.

Although you might not manage a rental portfolio, hopefully you'll see how the concepts in the Rental Manager application could be structured to develop an application of interest to you. For example, it could be turned into an application to manage details of a sports team or of participants in a running race.

2.1.1 Laying the foundations

To start developing the Rental Manager application, open Xcode and select the File > New > New Project option. In the dialog that appears, select the Navigation-based Application template and name the project "RentalManager." You should end up with Xcode showing a window similar to the one in figure 2.2.

Pressing Cmd-R to run the application at this stage displays an almost empty iPhone screen—but not quite as empty as the display that results from running the



Figure 2.1 Screenshots demonstrating various aspects of the Rental Manager application we build in this book. As well as smaller examples, each chapter reinforces concepts by adding functionality to this larger sample project.

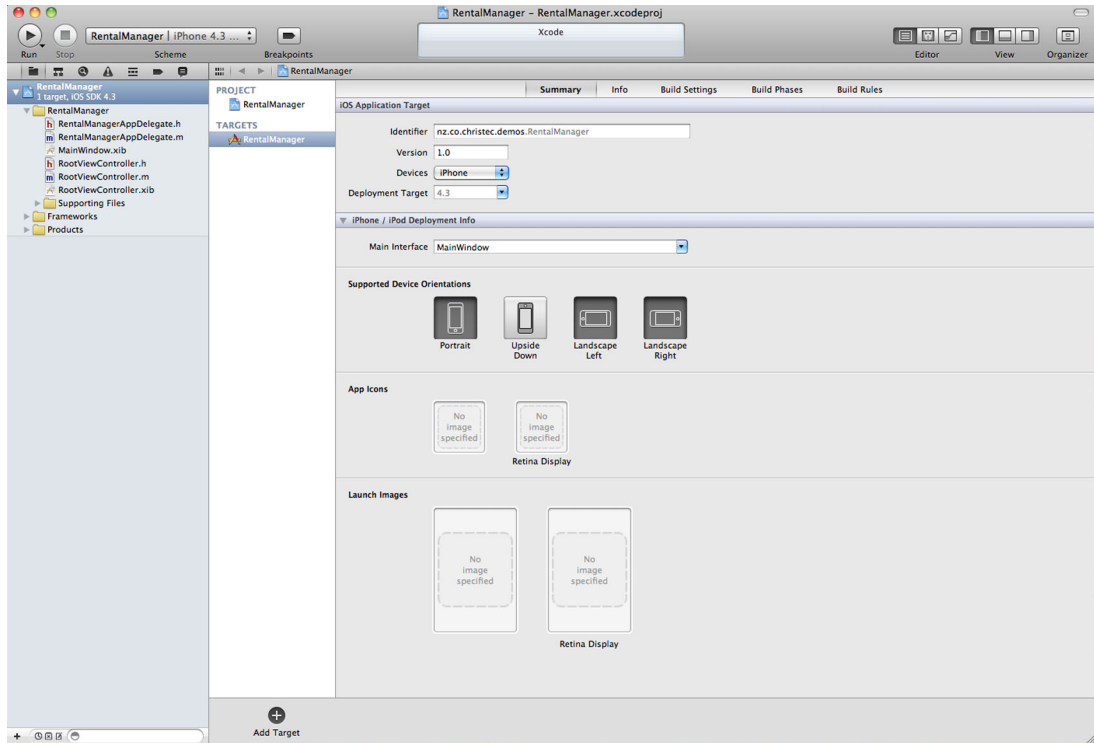


Figure 2.2 The main project window in Xcode immediately after creating the Rental Manager application using the Navigation-based Application template

View-based Application (as in the Coin Toss project from chapter 1). The Navigation-based Application template inserts a few user interface elements, such as a distinctive blue navigation bar at the top. As the Rental Manager application develops, you'll extend the various features provided, but for now, let's concentrate on adding the ability to display a list of rental properties.

When you run the application, you may notice that the white background is divided into a number of rows by light gray lines, and by using your finger, you can slide the list up and down. This control was added by the project template and is called a `UITableView`. To display data in this control, you must write a small amount of code to specify how many rows it should add and what each row contains.

Select the file named `RootViewController.m` and open it for editing. The project template has inserted a fair amount of source code into this file for your convenience (although, at present, most of it's commented out).

Locate the two methods named `tableView:numberOfRowsInSection:` and `tableView:cellForRowAtIndexPath:` and replace them with the code in the following listing.

Listing 2.1 Handling UITableView requests for the contents to display in the table

```

- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section {
  {
  return 25;
  }

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {
  {
  static NSString *CellIdentifier = @"Cell";
  UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[[UITableViewCell alloc]
      initWithStyle:UITableViewCellStyleDefault
      reuseIdentifier:CellIdentifier] autorelease];
  }

  cell.textLabel.text = [NSString
    stringWithFormat:@"Rental Property %d", indexPath.row];

  NSLog(@"Rental Property %d", indexPath.row);

  return cell;
  }
}

```

1 Determine number of rows

2 Create table view cell

3 Set the line of text

When the Rental Manager application runs, the UITableView control calls your `tableView:numberOfRowsInSection:` method to determine how many rows you want to display **1**. It then calls `tableView:cellForRowAtIndexPath:` a number of times as the user slides up and down the list to obtain the content for each visible row.

Your implementation of `tableView:cellForRowAtIndexPath:` is made up of two steps. The first **2** creates a new table view cell with the `UITableViewCellStyleDefault` style. This style displays a single line of large, bold text (other built-in styles replicate the layouts found in the settings or iPod music player applications). The second step **3** sets that line of text to the string "Rental Property %d", where %d is replaced with the index position of the current row.

Press Cmd-R to rerun the application and you should see the main view displaying a list of 25 rental properties. Your challenge in this chapter is to learn how to store data in your application before expanding your `tableView:cellForRowAtIndexPath:` method to display some practical information about each rental property.

Before moving on, take a look at `tableView:cellForRowAtIndexPath:` and, in particular, the last line that calls a function named `NSLog`. Notice that this method takes arguments similar to those of `NSString`'s `stringWithFormat:` method, which generated the string that was displayed in the table view cells onscreen.

`NSLog` is a handy function to learn and use. It formats a string but also sends the result to the Xcode debugger console (see figure 2.3). `NSLog` can be a useful way to diagnose the inner workings of your application without relying on breakpoints.

While the Rental Manager application is running, you can view the output from calls to `NSLog` by viewing the debugger console (Shift-Cmd-Y), as shown in figure 2.3.

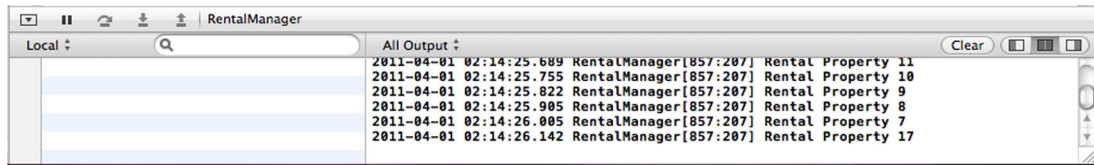


Figure 2.3 The Debugger Console window can be handy place to view diagnostic messages from the internal workings of your application as it's running.

As you scroll up and down the list of rental properties, you should see the console window logging which rows the `UITableView` has requested details of.

Now that the Rental Manager application has been introduced, and you have the initial shell up and running, let's get back to the subject at hand: how to store data in your applications. Throughout the rest of the chapter, feel free to insert the various code snippets shown into `tableView:cellForRowAtIndexPath:` and experiment. At the end of the chapter, we come back to this application and flesh it out for real.

2.2 The basic data types

The Objective-C language defines a set of standard data types that are provided as simple building blocks. These built-in data types are often called primitive data types, because each type can store a single value and can't be broken down into smaller units.

Once you master the use of primitive data types, it's also possible to combine multiple primitive data types to produce larger, more complex composite data types. These complex data types have names such as `enum`, `union`, and `struct`.

The primitive data types we cover in this chapter include `int`, `float`, `double`, `char`, and `bool`. Let's get started and discuss what kinds of data types you can use to store numerical data in your Rental Manager application.

2.2.1 Counting on your fingers—integral numbers

Integers are an integral part of any programming language. An integer is a whole number that can be negative or positive. The values 27, -5, and 0 are all valid integer values, but 0.82 isn't because it contains a decimal point.

To declare an integer variable, use the data type `int` (which is shorthand for integer), as demonstrated here:

```
int a;
```

By default, variables of type `int` are signed and can represent both positive and negative values. Sometimes you may want to restrict an integer variable to store only positive numbers. You can do this by adding the special qualifier `unsigned` before the data type, as shown in the following variable declaration:

```
unsigned int a;
```

This means the variable `a` will be allowed to store only positive numbers. Conversely, you can explicitly create a signed integer variable by using the `signed` qualifier:

```
signed int a;
```

Because variables of type `int` are signed by default, it's uncommon to see the signed qualifier used in most applications—its use is somewhat redundant.

Once you declare a variable, you can assign it a value with the assignment operator, which is represented by an equals sign. For example, the following statements declare a new variable called `a` and then assign it the value 15.

```
int a;
a = 15;
```

Because it's common to declare a variable and then assign it an initial value, both statements can be combined:

```
int a = 15;
```

The value 15 used in this assignment statement is called a *constant*. A constant is any value that can never change its value while the application is running. A constant doesn't have to be a single number; for example, the following variable declaration also makes use of a constant value.

```
int a = 5 + 3;
```

The value calculated by the expression `5 + 3` can never result in a number other than 8. The Objective-C compiler calculates the value of this expression during compilation and replaces it with a single constant value.

By default, integer constants are specified in decimal, or base 10, which is the most familiar notation for most people. It's also possible to specify integer constants in a number of other bases by using a special prefix in front of the number, as detailed in table 2.1.

Table 2.1 Different ways to express the value 15 as an integer constant. Each format is identified by a special prefix that precedes the number.

Name	Base	Prefix	Example constant
Octal	8	0	017
Decimal	10	—	15
Hexadecimal	16	0x	0x0F

One trap new developers occasionally make is to include a leading zero at the start of a decimal number. As far as Objective-C is concerned, `017` isn't the same value as 17. The leading zero in front of the first constant means the number is interpreted as an octal (base 8) number and hence equals the decimal value 15.

FACING THE LESS-THAN-IDEAL REAL WORLD

The iPhone is a vast improvement over the hardware and memory constraints of a traditional cell phone, but it's still constrained by real-world realities such as a fixed amount of memory being available to applications. In an ideal world, developers

wouldn't have any constraint on the value of integers—they could be infinitely high or low—but, unfortunately, constraints do exist. Declaring a variable of type `int` allocates a set amount of memory in which to store the value, and hence it can represent only a limited range of values. If you wanted to store infinitely large values, the variable would also require an infinite amount of memory, and that's clearly not possible.

This is one reason for having `unsigned` and `signed` qualifiers. Using the `unsigned` qualifier trades off the ability to store negative numbers with the ability to double the range of positive values you can store in the same amount of memory. Other qualifiers include `short` and `long`, which can be added to an `int` data type to expand or contract the size of the variable. The most common sizes are listed in table 2.2.

Table 2.2 Common integer data types. Various modifiers can be used to alter the size of a variable and hence the valid range of values that can safely be stored in them.

Data type	Size (bits)	Unsigned range	Signed range
<code>short int</code>	16	0–65,535	–32,768–32,767
<code>int</code>	32	0–4,294,967,295	–2,147,483,648–2,147,483,647
<code>long int</code>	32	0–4,294,967,295	–2,147,483,648–2,147,483,647
<code>long long int</code>	64	0–($2^{64} - 1$)	– 2^{63} –($2^{63} - 1$)

In Objective-C, `int` is the default data type for variables and parameters. This means you can remove the keyword `int` from your variable declaration statement and, in most cases, it will still compile. Therefore, the following two variable declarations are equivalent:

```
unsigned a;
unsigned int a;
```

The first variable declaration implicitly implies the presence of the data type `int` by the presence of the `unsigned` qualifier. In the second declaration, the `int` data type is said to be explicitly specified because it's present in the declaration.

THE LEGACY OF CONTINUAL PROGRESS—`NSInteger`, `NSUInteger`, AND THEIR ILK

As you explore Cocoa Touch, you'll find that most APIs use data types with names such as `NSInteger` or `NSUInteger` instead of `int` and `unsigned int`. These additional types are part of Apple's progress toward 64-bit computing.

Currently, all iOS-powered devices (and older versions of Mac OS X) use a programming model called ILP32, which supports a 32-bit address space. Since Mac OS X 10.4, the desktop has been moving toward a different programming model, LP64, which supports a 64-bit address space. Under the LP64 model, variables of type `long int` and memory addresses are increased to 64 bits in size (compared to the 32 bits shown in table 2.2), whereas the size of all other primitive types, such as `int`, remain the same.

As part of the effort to take full advantage of 64-bit platforms, Cocoa introduced the `NSInteger` data type to provide a data type that was a 32-bit integer on 32-bit platforms

while growing to a 64-bit integer when compiling the same source code on a 64-bit platform. This allows code to take advantage of the increased range provided by 64-bit integers while not using excessive memory when targeting 32-bit systems.

Eagled-eyed developers may wonder why new data types such as `NSInteger` were introduced when existing data types such as `long int` would appear to already fit the desired role. `NSInteger` exists for use in APIs that should be 64-bit integers on 64-bit platforms but for one reason or another must be typed as `int` instead of `long int` on 32-bit platforms.

Good habits learned now mean less hassle in the future

Declaring your variables using data types such as `NSInteger` and `NSUInteger` can be considered a form of future-proofing your source code. Although they're identical to `int` and `unsigned int` when compiling for the iPhone today, who knows what's around the corner? Perhaps a future iPhone or iPad will be a 64-bit device, or you'll want to reuse some of your source code in a matching desktop application.

It's much easier to establish the habit of using portable data types such as `NSInteger` or `NSUInteger` now (even if you don't get much immediate benefit from it) than to have to correct such portability issues in the future.

2.2.2 Filling in the gaps—floating-point numbers

When modeling the real world, it's common to come across numbers that contain a fractional part, such as 0.25 or 1234.56. An integer variable can't store such values, so Objective-C provides alternative data types called `float` and `double` for storing this kind of data. As an example, you can declare a variable `f` of type `float` and assign it the value 1.4 as follows:

```
float f = 1.4;
```

Floating-point constants can also be expressed in scientific or exponential notation by using the character `e` to separate the mantissa from the exponent. The following two variable declarations are both initialized to the same value:

```
float a = 0.0001;  
float b = 1e-4;
```

The first variable is assigned the value 0.0001 via a familiar decimal constant. The second variable is assigned the same value but this time via a constant expressed in scientific notation. The constant `1e-4` is shorthand for 1×10^{-4} , which, once calculated, produces the result 0.0001. The `e` can be thought of as representing “times 10 to the power of.”

In Objective-C, floating-point variables are available via two main data types, which trade off the range of possible values they can represent and the amount of memory they use, as shown in table 2.3.

You may wonder how a variable of type `float` or `double` can store such a wide range of values when a similar sized `int` can only store a much smaller range of values. The answer lies in how floating-point variables store their values.

Table 2.3 Common floating-point data types. A `double` takes twice the amount of memory as a `float` but can store numbers in a significantly larger range.

Data type	Size (bits)	Range	Significant digits (approx.)
<code>float</code>	32	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7
<code>double</code>	64	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15

On the iPhone, as with most modern platforms, floating-point values are stored in a format called the IEEE 754 Standard (<http://grouper.ieee.org/groups/754>). This format is similar in concept to scientific notation. With a lot of hand waving to gloss over more complex details, you can imagine that the 32 or 64 bits that make up a `float` or `double` value are divided into two smaller fields representing a mantissa and an exponent.

By using an exponent-based format, you can represent very large or very small numbers. But as with most things, this doesn't come completely for free. With only a restricted set of values you can use for the mantissa (due to the limited number of bits assigned to store it), you can't represent every value in the extended range enabled by the exponent. This leads to the interesting discovery that certain decimal values can't be stored precisely. As an example, the output of the following code snippet may surprise you:

```
float f = 0.6f;
NSLog(@"0.6 = %.10f", f);
```

The `%.10f` bit in the call to `NSLog` requests that the value be printed to 10 decimal places, but instead of the value `0.6000000000`, the value `0.6000000238` gets printed. The reason for this inaccuracy is that when the decimal value `0.6` is converted to a binary, or base 2, number, it produces an infinitely repeating sequence (similar in concept to how the value `0.33` behaves in decimal). Because a `float` variable has only a certain number of bits in which to store the number, a cut-off has to be made at some stage, leading to the observed error.

Many calculations with `floats` produce results that require rounding in order to fit in 32-bits. In general, a variable of type `float` should be relied on to be accurate to only about 7 significant digits, but a `double` extends this to about 15.

Careful consideration should be given when using floating-point numbers. Unlike integer values that can represent all values in their specified ranges, calculations on floating-point numbers are at worst an approximation of the result. This means you should rarely perform an equality comparison between two floating-point numbers because any intermediate calculations may introduce subtly different rounding type errors. Instead, floating-point numbers are traditionally compared by subtracting one value from another and checking that the difference is less than a suitably small epsilon value.

What does floating point mean?

In computing, *floating point* means that the equivalent of the decimal (or radix) point in the number can “float”: the decimal point can be placed anywhere between the significant digits that make up the number, on a number-by-number basis.

By contrast, in a fixed-point number, the decimal point is always positioned with a fixed number of digits after it.

Objective-C doesn’t provide any standard fixed-point data types; you can typically implement them yourself by using the existing integer data types. As an example, storing monetary values as a number of cents could be considered a fixed-point format with an implied (or fixed) decimal point positioned before the last two decimal digits. For instance, the integer value 12345 could be considered to represent the value \$123.45.

2.2.3 Characters and strings

In addition to data types that enable the storage of numerical data are a number of other types that allow storage of other categories of data. For example, the `char` data type can be used as the basis for storing textual data in your application.

IN THE BEGINNING THERE WAS TYPE CHAR

A variable of type `char` can store a single character, such as the letter `a`, the digit `6`, or a symbol such as an asterisk. Because some of these characters (such as a semicolon or curly brace) already have special meaning in Objective-C, special care must be taken when defining character constants. In general, a character constant is formed by enclosing the desired character in a pair of single quotation marks. As an example, the letter `a` can be assigned to a variable as shown here:

```
char c = 'a';
```

The `char` data type can be considered to be a tiny 8-bit integer, so it’s also possible to manually consult an ASCII character chart and assign a numeric value directly:

```
char d = 97;
```

If you refer to an ASCII character chart, you’ll notice that the value 97 represents a lowercase `a`, so variables `c` and `d` in the preceding examples will both store identical values.

Enclosing character constants in single quotation marks helps specify most characters that are printable. But a few, such as the carriage return and newline characters, are impossible to enter into your program’s source code in this manner. Objective-C therefore recognizes several escape sequences, which allow these special characters to be placed in character constants. See table 2.4 for a list of common backslash escape sequences.

By default, `char` is an unsigned value that can store a value between 0 and 255. Using the `signed` qualifier allows storage of a value between -128 and 127. In most cases, however, you should probably stick with the `int` data type if you want to store numbers.

Table 2.4 Common backslash escape sequences used to specify special characters in a character constant. Most characters in this list require special handling because they have no visible representation on a printed page.

Escape sequence	Description	Escape sequence	Description
\r	Carriage return	\"	Double quotation marks
\n	Newline	\'	Single quotation marks
\t	Horizontal tab	\\	Backslash

After declaring a variable that can store a single character, you'll undoubtedly want to expand on this to store sequences of characters to form entire words, sentences, or paragraphs. Objective-C calls such a sequence of characters a *string*.

STRINGING THINGS ALONG

A string is a sequence of characters placed one after another. Objective-C supports two types of strings: a traditional C-style string picked up via the C-based heritage of Objective-C and a new object-oriented type called `NSString`.

To declare a C-style string variable, you use the data type `char *`. A string constant is represented by a set of characters enclosed in double quotation marks and can also use character escape sequences such as those listed in table 2.4. For example, the following code snippet stores the string "Hello, World!" in a variable called `myString`:

```
char *myString = "Hello, World!";
```

The standard C runtime library provides various functions that work with C-style strings. For example, `strlen` can be used to determine the length of a particular string:

```
int length = strlen(myString);
```

When using C-style strings, you're responsible for ensuring enough memory is allocated to store any resultant string that might be generated by an operation. This is a critically important fact to remember, especially when using functions such as `strcat` or `strcpy`, which are used to build or append to existing strings. To append the text "are awesome!" to the end of an existing string stored in a variable called `msg`, you could use the following statement, which uses `strcat` to perform a string concatenation:

```
char msg[32] = "iPhones";
strcat(msg, " are awesome!");
```

Although this code snippet is correct, it has a problem if the resultant string could ever become larger than 31 characters. The square brackets after `msg` cause the compiler to allocate space for 31 characters (plus a so-called NULL character, which indicates the end of the string). If the string concatenation result ever becomes larger than the allocated space, it overwrites whatever lies next in memory, even if that stores an unrelated variable. This situation, called a *buffer overrun*, leads to subtle and hard-to-detect bugs such as variables randomly changing values or application crashes depending on how the user interacts with your application.

Objective-C therefore defines a much more practical data type, called `NSString`, which we discuss in depth in chapter 3.

Hello, hola, bonjour, ciao, 餓, привет

The `char` data type was created at a time when internationalization of software wasn't much of a concern. Therefore, most iPhone applications using C-style strings will probably make use of other closely related data types such as `unichar`. `unichar` is a 16-bit character data type that stores character data in UTF-16 format.

2.2.4 Boolean truths

Many languages have a Boolean data type capable of storing two states, commonly referred to as `true` and `false`. Objective-C is no exception: it provides the `BOOL` data type, which by convention uses the predefined values `YES` and `NO` (although `TRUE` and `FALSE` are also defined). You can assign a value via a simple constant as follows:

```
BOOL result = YES;
```

But it's more common to calculate Boolean values by performing a comparison between one or more values of another data type, as demonstrated here:

```
int a = 10;
int b = 45;
BOOL result = a > b;
```

As in most languages, the `>` operator compares the value on the left against the value on the right and returns `true` if the left side is greater than the right side. Table 2.5 lists the most common comparison and logical operators available in Objective-C.

Table 2.5 Common comparison and logical operators available in Objective-C for use in Boolean expressions. Don't confuse `&&` and `||` with the `&` and `|` operators, which perform a different task.

Operator	Description	Example expression
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal	<code>x >= y</code>
<code><=</code>	Less than or equal	<code>x <= y</code>
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>!</code>	Not (logical negation)	<code>!x</code>
<code>&&</code>	Logical And	<code>x && y</code>
<code> </code>	Logical Or	<code>x y</code>

Unlike some languages with a true Boolean data type, Objective-C doesn't restrict variables of type `BOOL` to storing only the two values `YES` and `NO`. Internally, the framework defines `BOOL` as another name for the signed char data type discussed previously. When Objective-C evaluates a Boolean expression, it assumes any nonzero value indicates true and a value of zero indicates false. This choice of storage formats can lead to some interesting quirks. For example, the following code snippet tries to suggest that a `BOOL` variable can store values of both `true` and `false` at the same time:

```
BOOL result = 45;

if (result) {
    NSLog(@"the value is true");
}
if (result != YES) {
    NSLog(@"the value is false");
}
```

This quirk occurs because the result variable stores a nonzero value (hence indicating truth) but doesn't store exactly the same value that `YES` is defined as (1). In general, it's best to compare `BOOL` values against the value `NO`. Because only one value can ever indicate false, you can be assured hard-to-spot errors such as the one demonstrated here don't occur in your source code.

This completes our initial coverage of how to declare and store values of built-in primitive data types. You still have much to learn about them, though, and we continue to discuss them throughout the rest of the book.

2.3 *Displaying and converting values*

Using variables to store values is useful, but at some stage, you'll want to display them to the user. Usually this will necessitate converting a raw value into a nicer, more formatted form. For example, a floating-point calculation may result in the value 1.0000000000000276, but a user may not be interested in an answer to this level of precision (if indeed the result was ever that precise given the potential inaccuracies discussed in section 2.2.2). It may be more suitable to present this value only to two decimal places, as 1.00.

In the following section, we discuss in detail how you can alter the display of arguments provided to a call to `NSLog` by altering what is called the *format string*. We also take a brief look at how numeric values can be converted between the various representations and how this too can affect the results of a calculation.

2.3.1 *NSLog and Format Specifiers*

The first argument provided in a call to `NSLog` specifies what is called the *format string*. `NSLog` processes this string and displays it in the Xcode debugger console. In most cases, the format string contains one or more placeholders that are indicated by a `%` character. If placeholders are present, `NSLog` expects to be passed a matching number of additional arguments. As `NSLog` emits its message, it substitutes each placeholder

with the value of the next argument. As an example, when the following code snippet is executed, NSLog replaces the first instance of %d with the value of variable a and the second instance with the value of variable b, resulting in the string "Current values are 10 and 25" being emitted to the debug console:

```
int a = 10;
int b = 25;
NSLog(@"Current values are %d and %d", a, b);
```

In a placeholder definition, the character immediately following the % character specifies the expected data type of the next argument and how that value should be formatted into a text string. Table 2.6 contains some common data types you may use and their associated format specifiers.

Table 2.6 Common format specifiers used in NSLog format strings. Notice some data types have multiple format specifiers to control the way their particular values are presented. For example, integers can be displayed in decimal, octal, or hexadecimal form.

Data type	Format specifier(s)
char	%c (or %C for unichar)
char * (C-style string)	%s (or %S for unichar *)
Signed int	%d, %o, %x
Unsigned int	%u, %o, %x
float (or double)	%e, %f, %g
Object	%@

A couple of entries in table 2.6 deserve additional comment. For integers, you can choose to display values with %d (or %u) for decimal, %o for octal, or %x for hexadecimal. Because there are multiple sizes of integer variables, you may also need to prefix these specifiers with additional letters that indicate the size of the argument. You can use h for a short, l for a long, or ll for a long long. To format a long long int into hexadecimal, for example, you utilize the format specifier %llx.

In a similar fashion, float and double types have three options: %e for scientific notation, %f for decimal notation, or %g to have NSLog determine the most suitable format based on the particular value to be displayed.

The % character has special meaning in the format string, so special care must be taken if you want to include a percentage sign in the generated text. Whenever you want a % sign, you must provide two in a row to signify that you aren't specifying a new placeholder but inserting the percent symbol (%). This can be handy when displaying values expressed as percentages. The following code snippet emits the text "Satisfaction is currently at 10%":

```
int a = 10;
NSLog(@"Satisfaction is currently at %d%%", a);
```


Portability is never as simple as it first seems

If you followed our previous advice and made use of data types such as `NSInteger` and `NSUInteger`, you must be extra careful when using functions, such as `NSLog`, which accept format specifiers. On 32-bit targets, such as the iPhone, it's common to use the `%d` specifier to format integer values, as demonstrated in the following code sample:

```
NSInteger i = 45;
NSLog(@"My value is %d", i);
```

If you reuse this code in a 64-bit-based desktop application, however, it could result in incorrect behavior. In a 64-bit environment, a variable declared of type `int` stays at 32 bits in size, whereas a variable declared of type `NSInteger` is redefined to be equivalent to `long`, which makes it now 64 bits in size. Hence, the correct format specifier for `NSInteger` in a 64-bit environment is `%ld`:

```
NSLog(@"My value is %ld", i);
```

To avoid altering source code like this, Apple's 64-bit Transition Guide for Cocoa recommends always casting such values to `long` or `unsigned long` (as appropriate). For example:

```
NSLog(@"My value is %ld", (long)i);
```

The typecast to `long` ensures that `NSLog` is always provided a 64-bit argument even if the current platform is 32 bit. This means `%ld` will always be the correct specifier.

To further control the presentation of values, you can place a number between the `%` and the field data type. This acts as a minimum field width and will right-align the value by padding out the string with spaces if it's not already long enough. For example:

```
int a = 92;
int b = 145;
NSLog(@"Numbers:\nA: %6d\nB: %6d", a, b);
```

This should result in the console window displaying output similar to the following:

```
Numbers:
A:    92
B:   145
```

Placing a negative sign in front of the number causes `NSLog` to instead left-align the field, whereas prefixing the number with a zero pads with zeros instead of spaces. When formatting floating-point numbers, it's also possible to specify the desired number of decimal places by preceding the minimum field-width specifier with a decimal point and the desired number of decimal places. As an example, the following code snippet emits the string "Current temperature is 40.54 Fahrenheit":

```
float temp = 40.53914;
NSLog(@"Current temperature is %0.2f Fahrenheit", temp);
```

2.3.2 Type casts and type conversions

It's common to perform calculations using expressions that contain variables or constants of different data types. For example, a calculation may involve a variable of type `float` and another of type `int`. A CPU can typically perform calculations only on similarly typed values, so when such an expression is encountered, the compiler must temporarily convert at least one of the values into an alternative format.

These conversions fall under two categories: explicit type conversions, which you must manually specify in code, and implicit type conversions, which are performed automatically by the compiler under specific circumstances. Let's investigate the following expression:

```
int a = 2, b = 4;
int c = a / b;
NSLog(@"%d / %d = %d", a, b, c);
```

This code snippet performs a division between variables `a` and `b` and stores the result in `c`. At first blush, you may believe that the value of variable `c` will be `0.5`, because this is the result a pocket calculator would get when dividing 2 by 4. But if you execute the code, it will report that the answer is 0. Variable `c` can't store the value `0.5` because it's typed as an integer, which can store only whole numbers.

You may think the answer is to store the result of the division in a variable of type `float`, as demonstrated here:

```
int a = 2, b = 4;
float c = a / b;
NSLog(@"%d / %d = %f", a, b, c);
```

Unfortunately, this snippet will still print the incorrect answer. Although the result of the calculation is stored in a floating-point variable, the division operation still sees that both operands (variables `a` and `b`) are of type `int`. This causes the division to be performed as an integer division (you may remember from your school days that 2 divided by 4 equals 0, remainder 2). Once the result of the integer division is calculated, the compiler notices you want to store the integer value in a variable of type `float` and performs an implicit type conversion between the two number formats.

In order for the division to be performed as a floating-point division, you must force at least one of its operands to be of type `float`. Although you could modify the data types of variables `a` or `b` to achieve this, doing so may not be practical in all scenarios. As an alternative, you can modify the expression by placing `(float)` in front of one of the operands.

```
int a = 2, b = 4;
float c = (float)a / b;
NSLog(@"%d / %d = %f", a, b, c);
```

Placing the name of a data type inside parentheses is a request to the compiler to convert the current value (or expression) into the specified data type. This operation is called an *explicit type conversion*, or *typecast*, because you must explicitly provide the hint to the compiler.

In the code snippet given, you typecast variable `a` to force it to be converted into a `float` value before the rest of the expression is evaluated. This means the division operation now sees one operand of type `float` and another of type `int`. This causes the compiler to implicitly convert the other operand to type `float` to make them compatible with each other and then perform a floating-point division. This calculation results in the desired result of `0.5`.

It's important to note that not all typecasts result in a perfect conversion. It's possible for data to become "lost" or truncated in the process. For example, when the following statements are executed, the explicit typecast between a floating-point constant and the `int` data type causes data loss.

```
int result1 = (int)29.55 + (int)21.99;
int result2 = 29 + 21;

NSLog(@"Totals are %d and %d", result1, result2);
```

Notice that the first expression involving floating-point constants is evaluated to have the same result as the calculation using the values `29 + 21`. Typecasting a floating-point number into an integer removes anything after the decimal point: it performs a truncation instead of a rounding operation (which would have resulted in the total `52` for `result1`).

This concludes our look at the primitive data types available to Objective-C developers. Using these types, you can express most forms of data, numerical in the form of `int` and `float`, Boolean in the form of `BOOL`, and textual via various data types such as `char`, `char*`, and `NSString`.

As nice as this is, however, you'll still come across situations where these data types are lacking. As an example, you may need a variable that can store a limited subset of values, or you may want to group multiple related values. In such scenarios, Objective-C allows you to create your own custom data types, which is the topic we discuss next.

2.4 **Creating your own data types**

Objective-C provides a number of language constructs that allow you to define your own custom data types. These can be as simple as providing an additional name for an existing type or as complex as creating new types that can store multiple elements of information.

The first custom data type we investigate is called an *enumeration* and enables us to restrict the valid set of values an integer variable can store.

2.4.1 **Enumerations**

When analyzing the real world, it's common to summarize data into a small set of possible values. Rather than saying it was `52` degrees yesterday, you're perhaps more likely to state it was hot, warm, cold, or freezing. Likewise, a loan may have a state of approved, pending, or rejected, and a simple digital compass could point North, South, East, or West.

With what we've covered so far, the best approach to store such values would be to use an integer variable and to assign each state a unique value. For example, a value of 1 could indicate the compass was pointing North, whereas a value of 2 could indicate it was pointing East. As a human, however, it can be rather tricky to manage this because you don't tend to think in terms of numbers. Seeing a variable set to the value 2, you don't immediately think "East." Objective-C has a custom data type called an enumeration that's designed to resolve this.

To define a new enumerated data type, you use the special keyword `enum`. This is followed by a name and a list of possible values in curly braces. For example, you could declare an enumerated data type to store compass directions as follows:

```
enum direction { North, South, East, West };
```

Once the new enumeration data type is declared, you can create a variable of type `enum direction` and assign it the value `North`, as shown here:

```
enum direction currentHeading = North;
```

You may wonder what integer value `North` represents because you didn't explicitly specify this during the declaration of the enumerated data type. By default, the first enumerated value (`North`) is given the integer value 0, and each value thereafter is given a value one greater than the name that preceded it. This convention can be overridden when the enumeration is declared. As an example, with the following declaration:

```
enum direction { North, South = 10, East, West };
```

`North` has the value 0 because it's the first enumerated value. `South` has the value 10 because it's explicitly specified via an initializer, and `East` and `West` have the values 11 and 12 respectively because they immediately follow `South`. It's even possible for more than one name to map to the same integer value, although in this case there's no way to tell the two values apart.

In theory, an enumerated data type should be used to store only one of the values specified in the enumeration. Unfortunately, Objective-C won't generate a warning if this rule is violated. As an example, the following is perfectly valid in an Objective-C program and in many cases won't produce a compilation error or warning:

```
enum direction currentHeading = 99;
```

Even if this is possible, you shouldn't rely on it. Try to restrict yourself to storing and comparing only the symbolic names you specified in the type declaration. Doing so allows you to easily change the values in a single place and have confidence that all logic in your application is correctly updated. If you make assumptions based on the value of an enumerated data type (or store random integer values), you defeat one of their main benefits, which is the association of a symbolic name to a specific value.

The association of integer value and symbolic name has a number of benefits, including an improved debugging experience, as can be seen in figure 2.4.

```

// Customize the number of sections in the table view.
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return 25;
}

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier] autorelease];
    }

    enum direction { North, South = 10, East, West };
    enum direction currentHeading = South;
    int currentHeading2 = South;
    int currentHeading2 = 10;

    // Configure the cell.
    cell.textLabel.text = [NSString stringWithFormat:@"Rental Property %d", indexPath.row];
    NSLog(@"Rental Property %d", indexPath.row);
    return cell;
}

/*
// Override to support conditional editing of the table view.
*/

```

Debugger Variable Pane:

- self = (RootViewController *) 0x4d0d170
- _cmd = (struct objc_selector *) 0x48285c
- tableView = (UITableView *) 0x581bc00
- indexPath = (NSIndexPath *) 0x4d24e40
- North = (enum direction) North
- South = (enum direction) South
- East = (enum direction) East
- West = (enum direction) West
- cell = (UITableViewCell *) 0x4d25100
- CellIdentifier = (NSString *) 0x3580 Cell
- currentHeading2 = (int) 10
- currentHeading = (enum direction) South
- East = (enum direction) East
- North = (enum direction) North
- South = (enum direction) South
- West = (enum direction) West

All Output:

```

GNU gdb 6.3.50-20050815 (Apple version gdb-1510) (Sat Feb 12 02:52:12 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".sharedlibrary apply-load-rules all
Attaching to process 900.
Pending breakpoint 1 ~ ""RootViewController.m":73" resolved
Current language: auto; currently objective-c
(gdb)

```

Figure 2.4 The benefit of using enumerations extends into the debugger. The variable pane displays the name associated with the current value instead of its raw integer value. Even though the variable `currentHeading2` is assigned the value `South` (like `currentHeading1`), the debugger displays the value `10` due to its data type being `int`.

In the code snippet shown, the variables `currentHeading1` and `currentHeading2` are both set to the enumerated value `South`; the difference is that one variable is of type `enum direction`, whereas the other is of type `int`. As you can see, the debugger is able to determine if a variable is of an enumerated data type and will display the name associated with the variable's current value rather than displaying the raw integer value. This can be an immense benefit during long debugging sessions when the enumeration values represent the state of the application.

2.4.2 Structures

In more complex applications, another common scenario is to have a set of variables that are related in concept. For example, you may want to record the width, height, and depth of a box. You could use individual variables to represent this information:

```

int width;
int height;
int depth;

```

And if you wanted to store details of multiple boxes, you could duplicate this set of variables to come up with something similar to the following:

```
int width_box1, width_box2, width_box3;
int height_box1, height_box2, height_box3;
int depth_box1, depth_box2, depth_box3;
```

This process could become a laborious one if you decided in the future to add additional details you wanted to record for each box, such as its color. You would need to find all occurrences of box details and manually update the variable declarations.

Using individual variables also makes passing details of a specific box more difficult. Rather than passing a single variable, you must pass multiple values, and there's nothing stopping you from accidentally transposing the variables representing the width and height of a box. These kinds of bugs can be subtle and hard to detect.

It would be better if you could declare a variable of data type `box` and have the compiler automatically know that it must provide space for individual width, height, and depth values. By doing this, you'd have a single variable that can be easily passed and a single definition of what a box consists of.

Objective-C calls such custom data types *structures*. Structures are declared with the `struct` keyword and consist of a name followed by a semicolon-separated list of field declarations. You could define a box structure as follows:

```
struct box {
    int width;
    int height;
    int depth;
};
```

This statement declares that a box is a data structure that contains three integer fields, called `width`, `height`, and `depth`. Once a structure is declared, you can create variables of type `struct box`, and they'll be allocated space to store three integer values. For example, you could create enough variables to store information about five boxes as follows:

```
struct box a, b, c, d, e;
```

Each variable would have a unique width, height, and depth value associated with it. When using a structure-based variable, you must specify which field you want a particular statement to access. You do this by placing a period after the variable name followed by the desired field name. For example, to set the width of the second box, you could use the following statement:

```
b.width = 99;
```

As another example, you could calculate the volume of box `b` as follows:

```
int volume = b.width * b.height * b.depth;
```

Like variables of primitive data types, structure variables can be initialized with an initial value. You can achieve this through two forms of syntax. The first is to provide a value for each field in a set of curly braces:

```
struct box a = { 10, 20, 30 };
```

This line specifies a box 10 wide \times 20 high \times 30 deep. The values in the curly braces are specified in the same order as the fields were defined in the structure declaration. If you don't specify enough values, any unspecified fields are left uninitialized. An alternative syntax explicitly specifies the name of each field being initialized, as follows:

```
struct box a = { .width=10, .height=20, .depth=30 };
```

This alternative syntax can be clearer in intent and will also cope if the order of the fields in the structure declaration changes. It's even possible to initialize only select fields in the structure when using this syntax:

```
struct box a = { .depth=30 };
```

This last example sets the depth field to 30 and leaves the width and height fields uninitialized. In this sense, *uninitialized* means the field will have the value 0 or its equivalent, such as `nil`, although this depends on the storage class of the variable.

Structures are a handy way to group related sets of variables into easier-to-manage chunks, but they don't solve all of the problems you may encounter with storing details about multiple highly similar objects. For example, you may need to calculate the total width of all the boxes in your application. To do this, you could come up with an expression similar to the following:

```
struct box a, b, c, d, e, f;  
int totalWidth = a.width + b.width + c.width + d.width + e.width;
```

Although this is manageable with only five boxes, imagine how torturous the expression would be to write if the application instead needed to store details of 150 different boxes. There would be a lot of repetitive typing involved, and it would be easy to miss a box or include a box multiple times. Each time you change the number of boxes your application needs to store, you'd also need to update this calculation. That sounds like a lot of pain! What you want the expression to say is “add the width of every box” and have the statement work without modification no matter the number of boxes your application currently keeps track of. Not surprisingly, Objective-C has a data type to help you out.

2.4.3 Arrays

Another common scenario is the need to store multiple values of the same data type. C provides a handy data structure called an *array* to make this task easier. An array is a data structure that stores a list of items. Each item (or element) is stored consecutively, one after the other, in memory and can only be accessed by its relative position. This position is often called the item's *index*. To declare an array capable of storing details of 150 boxes, you could write a variable declaration similar to this:

```
struct box boxes[150];
```

With this statement, you've declared enough space to store 150 boxes but used only one variable name, `boxes`, to identify them all. The number specified in square brackets indicates the number of items that can be stored in the array.

To access the details of each box, you must provide the name of the array along with the index value of the desired element. By convention, the first element in an array has an index of 0, and each element thereafter has an increasing index value. This means that the last entry in a 150-element array will be accessible via index 149. The following statement will access and print the width of the sixth box:

```
NSLog(@"The 6th box has a width of %d inches", boxes[5].width);
```

Notice that to access the sixth element, you specified index 5. That's because indexes start at 0 and not the more "natural" value of 1. When accessing array elements, the array index need not be specified as a simple constant. You can use any valid expression that results in an integer value. This allows you to write code that accesses different array items based on the state of the application. For example, consider the following code sample, which adds the widths of all 150 boxes:

```
int totalWidth = 0;
for (int i = 0; i < 150; i++) {
    totalWidth = totalWidth + boxes[i].width;
}
```

This example uses a `for` loop to repeat the statement `totalWidth = totalWidth + boxes[i].width` with the value of variable `i` incrementing between 0 and 149. Contrasting this expression with the one mentioned toward the end of section 2.4.2, you can see that this version can easily be updated to cope with different numbers of boxes by changing the value 150 in the second line. Using an array in this scenario makes for an easier and vastly more maintainable solution.

INITIALIZING ARRAYS

When declaring array variables, it can be helpful to provide each element in the array with an initial value. To do so, place a comma-separated list of initial values in a set of curly braces, similar to how structures are initialized:

```
int count1[5] = { 10, 20, 30, 40, 50 };
int count2[] = { 10, 20, 30, 40, 50 };
```

This code snippet creates two integer arrays with five elements. In each array, the first element stores the value 10, the second element stores 20, and so on.

In the first array, the array is explicitly sized to store five elements. It's an error to provide more initialization values than the size of the array, but if fewer are provided, any uninitialized elements at the end of the array will be set to zero.

The second array demonstrates that when initializing an array, it's possible to omit its size. In this scenario, the C compiler infers the size from the number of initialization values provided.

HOW DO ARRAYS DIFFER FROM SIMPLE TYPES?

Arrays behave a little bit differently from the data types we've discussed up to this point. For example, you may expect the following code snippet to print the result A=1, B=2, C=3 because of the assignment statement `array2 = array1` setting the second array to the contents of the first:

```
int array1[3] = { 1, 2, 3 };
int array2[3] = { 4, 5, 6 };

array2 = array1;
NSLog(@"A=%d, B=%d, C=%d", array2[0], array2[1], array2[2]);
```

But if you attempt to build this code sample, you should notice a compiler error complaining cryptically about “incompatible types in assignment.” What this error is attempting to convey is that the variable `array2` can't be used on the left-hand side of the assignment (=) operator: it can't be assigned a new value, at least not in the manner shown here.

In chapter 3, as we discuss the object-oriented features of Objective-C, we explore concepts such as pointers and the difference between value and reference types, which will help explain why Objective-C refuses to accept what on initial glance looks like a perfectly acceptable request.

2.4.4 The importance of descriptive names

Objective-C provides a way to declare an alternative name for an existing data type via a statement known as a *type definition*, or typedef for short. Typedefs can be useful when the built-in data type names aren't descriptive enough or if you want to differentiate the purpose or intent of a variable from its physical data type (which you may decide to change at some point in time).

You've already seen typedefs, although they weren't pointed out as such: the `NSInteger` and `NSUInteger` data types discussed earlier are typedefs that map back to the built-in datatypes `int` and `long int` as required. This isn't Apple-specific magic, simply a couple of lines of prespecified source code that automatically get included in every Cocoa Touch application.

To declare your own typedefs, you use the `typedef` keyword followed by the name of an existing data type (or even the specification of an entirely new one) followed by the new name you want to associate with it.

As an example, you could assign the alternative name `cube` to the `struct box` data type you previously declared by adding the following type definition:

```
typedef struct box cube;
```

Or you could merge the declaration of `struct box` into the typedef statement:

```
typedef struct box {
    int width;
    int height;
    int depth;
} cube;
```

Both type definitions say that struct `box` can also be referred to by the name `cube`. This allows you to declare variables in your application as follows:

```
struct box a;
cube b;
```

The new data type `cube` created via the `typedef` statement is purely an element of syntactic sugar. As far as the Objective-C compiler is concerned, `struct box` and `cube` both mean the same thing: you've specified alternative names for the developer's convenience.

Although you can use typedefs to rename enums, structs, and unions, they can also be beneficial to provide alternative names for primitive data types such as `int` or `double`. One problem with the basic numeric data types is that they can sometimes seem meaningless in isolation. If you were presented with the following statement

```
double x = 42;
```

you wouldn't be able to determine what the value 42 represents. Is it a temperature, weight, price, or count? Giving an existing data type a new name can make such a statement more self-documenting. As an example, listing 2.2 contains a type definition that can be found in the Core Location API, which is responsible for GPS positioning.

Listing 2.2 Example typedef from Core Location's `CLLocation.h` header file

```
// CLLocationDegrees
// Type used to represent a latitude or longitude coordinate
// in degrees under the WGS 84 reference frame. The degree can
// be positive (North and East) or negative (South and West).
typedef double CLLocationDegrees;
```

Although the type definition provides an alternative name for `double`, it enables us to declare a variable using a statement similar to the following:

```
CLLocationDegrees x = 42;
```

This new statement, although essentially identical to the previous, makes the meaning of the value 42 much more apparent.

Good variable names are equally as important

Although the use of the `typedef` statement can provide more descriptive names to existing data types, this feature should not be relied on in isolation. For example, look at the given variable declaration:

```
CLLocationDegrees x = 42;
```

This could have arguably been made even more descriptive by using a better variable name instead of the `typedef`. For instance:

```
double currentHeadingOfCar = 42;
```

In general, try to be as descriptive as possible with any form of identifier, be that a variable, datatype, method, or argument name. The more self-documenting your code, the easier it is to maintain and come back to after periods of inactivity.

Let's put some of the concepts you've learned in this chapter into practice by completing the remaining tasks required to get the Rental Manager application to display details about the set of rental properties in your portfolio.

2.5 **Completing Rental Manager v1.0, App Store here we come!**

Now that you have a well-rounded understanding of how data can be stored in an Objective-C application and of the different data types involved, you're ready to get back to the Rental Manager application.

You may remember that when you left it earlier in the chapter, it was displaying a list of 25 rental properties, but each property was labeled "Rental Property x." This didn't provide you with much detail about each property! You now have the knowledge and skills required to resolve this problem.

The first step is to define the information you would like to associate with each rental property. Some good details to start with could be

- The physical address of the property
- The cost to rent the property per week
- The type of property (townhouse, unit, or mansion)

To store this information, you can define a custom data type based on a structure. Open up the `RootViewController.h` header file for editing and insert the definitions found in the following listing at the bottom of the file's existing content.

Listing 2.3 `RootViewController.h`

```
typedef enum PropertyType {
    Unit,
    TownHouse,
    Mansion
} PropertyType;

typedef struct {
    NSString *address;
    PropertyType type;
    double weeklyRentalPrice;
} RentalProperty;
```

The first addition is the definition of an enumeration called `PropertyType`. It's used to group the rental properties you manage into three distinct categories: units in a larger property, townhouses, or mansions.

The second addition is a custom data type called `RentalProperty` that nicely encapsulates all of the details you want to store about a rental property. This typedef statement declares that the `RentalProperty` data type is a structure containing individual address, property type, and weekly rental price fields. If you pay close attention, you'll notice there's no name specified after the `struct` keyword. When using a typedef, it isn't strictly necessary to name the `struct` because you usually don't intend people to refer to the data type in this manner but via the name assigned to it with the typedef.

Having modified `RootViewController.h` to specify the new data types related to storing rental property details, you're ready to declare details about an initial set of rental properties. Open up `RootViewController.m` and insert the contents of the following listing directly below the line `#import "RootViewController.h"`.

Listing 2.4 RootViewController.m

```
#define ARRAY_SIZE(x) (sizeof(x) / sizeof(x[0]))

RentalProperty properties[] = {
    { @"13 Waverly Crescent, Sumner", TownHouse, 420.0f },
    { @"74 Roberson Lane, Christchurch", Unit, 365.0f },
    { @"17 Kipling Street, Riccarton", Unit, 275.9f },
    { @"4 Everglade Ridge, Sumner", Mansion, 1500.0f },
    { @"19 Islington Road, Clifton", Mansion, 2000.0f }
};
```

The main portion of listing 2.4 declares a variable called `properties`. This is an array of `RentalProperty` structures. The array is initialized with details of five example properties in your portfolio by using a combination of the array and structure initialization syntaxes discussed earlier in this chapter.

Now all that's left to do is to provide suitable `tableView:numberOfRowsInSection:` and `tableView:cellForRowAtIndexPath:` replacements that refer back to the data in the `properties` array to satisfy their requests. These can be found in the following listing.

Listing 2.5 RootViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    {
        return ARRAY_SIZE(properties);
    }
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    {
        static NSString *CellIdentifier = @"Cell";

        UITableView *cell =
            [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
        if (cell == nil) {
            cell = [[[UITableViewCell alloc]
                    initWithStyle:UITableViewCellStyleSubtitle
                    reuseIdentifier:CellIdentifier] autorelease];
        }

        cell.textLabel.text = properties[indexPath.row].address;
        cell.detailTextLabel.text =
            [NSString stringWithFormat:@"Rents for $%0.2f per week",
             properties[indexPath.row].weeklyRentalPrice];

        return cell;
    }
}
```

The `tableView:numberOfRowsInSection:` implementation isn't notable. It returns the number of items present in the `properties` array (5). It makes use of a C preprocessor macro defined in listing 2.4 to determine this number, but more about that later.

The `tableView:cellForRowAtIndexPath:` implementation has a couple of changes. The first is a change in table view cell styles. You're now requesting `UITableViewCellStyleSubtitle`, which provides an iPod application-style cell with two horizontal lines of text: a main one that's black followed by a gray line designed to show additional details.

In `tableView:cellForRowAtIndexPath:` you're provided with the index of the row the `UITableView` wants data for via the `indexPath.row` property. You can use this expression to index into the `properties` array to access the details, such as the address, of the associated property. Likewise, you can format a similar string detailing the rental price of the property to two decimal places for use as the details line.

Build and run the application (Cmd-R), and you should be rewarded with a much better list of rental properties. Your first practical version of the Rental Manager application is completed!

2.6 **Summary**

All software applications are ultimately about data and how to interpret, process, and present it to the user. Even games are required to store maps, enemy positions, and scoring information, for example. It's important to have a strong grasp of how to represent and store data in your applications.

In this chapter, you met some of the most basic data types available to Objective-C developers, including `int`, `float`, `char`, and `bool`. We also highlighted some of the potential issues these data types could throw up, such as the inability of floating-point numbers to accurately represent all values in their stated ranges.

As programs start to develop and grow in complexity, managing a large number of individual variables becomes untenable, so we investigated a number of Objective-C features such as enumeration, structures, and arrays that allow you to group multiple fields and constants.

In chapter 3, we complete our coverage of Objective-C data types by discussing the concept of objects. Objects are another form of data type, but clearly, because Objective-C begins with the word *object*, understanding them is fairly critical to the success use of Objective-C.

An introduction to objects

This chapter covers

- The concepts of objects, classes, and instances
- Class hierarchies, inheritance, and polymorphism
- Foundation Kit
- The `NSString` class

In Objective-C the use of object-oriented programming is optional. Because Objective-C is based on a C foundation, it's possible to use C-style functions (as evidenced by calls to `NSLog` in the previous chapter), but Objective-C's full power is unlocked only if you make full use of its object-oriented extensions.

In this chapter we reveal some of the benefits of object-oriented development by covering how the `NSString` class, provided by Foundation Kit, can improve your productivity while increasing the robustness of any code that interacts with text.

Before we go too far in depth on that particular topic, let's first take a look at the most basic concepts of object-oriented programming.

3.1 **A whirlwind tour of object-oriented programming concepts**

In this chapter we can't do justice to every concept associated with object-oriented programming (also termed OOP, for short). Instead, the goal of this chapter is to make sure you have a solid understanding of the fundamental concepts and benefits of object-oriented programming and a working knowledge of the terminology. Throughout this book, we discuss object-oriented programming concepts as they apply to Objective-C and expand on what's covered in this chapter. Let's get started by learning what's so wrong with C.

3.1.1 **What's wrong with procedural-based languages such as C?**

In very broad brush strokes, a procedural language requires greater concentration and observance of manually enforced or informal rules than does an object-oriented language.

One reason for this is that procedural languages focus on dividing the application's source code into individual functions, but generally their ability to control access to data is less fine-grained. Data will generally belong to a particular function or be globally accessible by any function. This causes problems when several functions need to access the same data. To be available to more than one function, such variables must be global, but global data can be accessed (or worse yet, modified in inappropriate ways) by any function in the application.

Object-oriented programming, on the other hand, attempts to merge these two concepts. When developing a new application, you first think of the different kinds of things represented in it, then the type of data it's required to store, and then the actions each thing should be able to perform. These "things" are commonly called *objects*.

3.1.2 **What are objects?**

When developing an object-oriented application, you're creating a miniature model of a system. That model is constructed from one or more building blocks called *objects*.

As an example, in a drawing application, a user may create three objects representing a circle and two rectangles. Each object has associated data that is relevant to itself. The circle object's data may describe a radius, while the rectangle objects, to be recognized as rectangles, will probably require a width and height to be stored as their data.

Objects in an application can usually be grouped into similar types. For example, all circles will require that similar attributes be specified to describe their size, shape, and color. But the attributes required to describe circles will most likely be different from the attributes required for all of the objects of type rectangle.

Individual objects in an application are created from a cookie-cutter-type template called a *class*. A class describes what type of data an object must store and the type of actions or operations it can perform.

3.1.3 What are classes?

A class is a specification, or blueprint, designed to describe the structure of one or more objects in a system that share a similar purpose. Classes are the most basic form of encapsulation in Objective-C: they combine a small amount of data with a set of relevant functions to manipulate or interact with that data.

Once a class is defined, its name becomes a new data type, meaning that you can declare a variable to be of that type. The class is like a factory line, rolling out cookie after cookie on demand. When instantiated, each newly created object has its own copy of the data and methods defined by the class. By convention, class names begin with a capital letter to differentiate them from method and instance variable names, which typically start with a lowercase letter.

3.1.4 Inheritance and polymorphism

An advantage of object-oriented programming is the ability to reuse existing classes time and time again. Not only can you create additional objects with little effort, but one class can build on the foundations of another. This technique, called *inheritance*, is similar to a family tree. One class in the system inherits, or derives, from another.

Inheritance has two major benefits:

- *Code reuse*—A subclass inherits all the data and logic defined by its superclass (ancestor). This avoids duplicating identical code in the definition of similar classes.
- *Specialization*—A subclass can append additional data or logic and/or override existing behavior provided by the superclass.

Class clusters

While researching Objective-C you may come across the concept of class clusters. These are an example of inheritance and polymorphism. In a class cluster, a superclass is documented, while a number of subclasses are purposely left undocumented (as private, implementation-specific details).

As an example, virtually all Objective-C tutorials, including this book, discuss using the `NSString` class to store text. It may surprise you that in most cases there will never be an object of type `NSString` created in your application.

Instead, when you request a new `NSString` object, one of a number of subclasses with names such as `NSCFString` is created in its place. Because these subclasses inherit from `NSString`, they can be used in its place. As the old saying goes, “If it walks like a duck, quacks like a duck, and swims like a duck, it’s probably a duck.”

`NSString` uses these “hidden” subclasses to allow itself to optimize memory and resource usage based on the specifics of each string generated.

The next time you’re in the Xcode debugger, hover the mouse over a variable of type `NSString`. In the data tip that appears, you’ll probably see the object’s data type listed as `NSCFString`. This is a class cluster in action.

The core concept of inheritance is that a subclass becomes a more specialized version of its superclass. Rather than describing all of the logic contained in the subclass from scratch, only the differences in behavior from the superclass need to be specified.

Polymorphism is a related concept that enables you to treat any object, no matter its class, as if it were typed as one of its superclasses. This works because subclasses can only extend or modify the behavior of a class; they can't remove functionality.

3.2 *The missing data type: id*

In chapter 2 we made one glaringly large omission when discussing the data types available to represent data in an application. We didn't cover how to store an object—something rather important for a language starting with the word *Objective!*

In Objective-C a variable, which can represent an object, makes use of a data type called `id`. For example:

```
id name = @"Christopher Fairbairn";
```

`id` is a special kind of data type in that it can store a reference to an object of any type. Objective-C also enables you to be more specific about the type of object you expect to store in a variable. For example, you would more commonly see the previous variable declaration declared as follows:

```
NSString *name = @"Christopher Fairbairn";
```

Here the `id` data type is replaced by `NSString *`. Being more explicit about the type of object you expect to store in the variable enables a number of helpful compile-time checks. For starters, the compiler will produce an error if you attempt to assign or store an object of another type to the variable, and a warning will be produced if you attempt to send the object a message that, to the best of the compiler's knowledge, it can't handle.

There's no magic with `id`

You may think there's some kind of magic occurring with the `id` data type. How can a variable of type `id` store an object of any data type, and why don't you need to specify a `*` character after the data type, as you do with other class names such as `NSString`? To answer these questions, declare a variable of type `id` and then double-click it while holding down the `Cmd` key. You should see something similar to the following:

```
typedef struct objc_object { Class isa; } * id;
```

This is the declaration of the `id` data type. It's another name (a type definition) for a pointer to a struct called `objc_object`. The `objc_object` struct can be considered as low-level "plumbing" in the Objective-C runtime. With a slight amount of hand waving, you can consider it to be the same as `NSObject`. Because all objects in Objective-C ultimately derive from `NSObject`, a variable of type `id` can store a pointer to any object, no matter its type. Because the declaration of the `id` data type contains a `*`, one isn't required when it's utilized in an application.

You may have noticed that the `NSString` example contained a `*` character before the variable name. This character has special significance (you'll notice it's also present in the declaration of the `id` data type discussed in the sidebar). This additional character indicates that the data type is a pointer. But what exactly does this mean?

3.3 Pointers and the difference between reference and value types

A variable in an application consists of four components:

- Name
- Location (where it's stored in memory)
- Type (what kind of data it can store)
- Current value

Until now, we haven't covered where variables are located or whether they are accessible by means other than their names. Understanding these concepts is closely related to the concept of pointers.

3.3.1 Memory maps

You can consider the memory of the iPhone as being made up of a large pile of bytes, each stacked one on top of another. Each byte has a number associated with it, called an *address*, similar to houses having an associated street number. Figure 3.1 represents several bytes of the iPhone's memory, starting at address 924 and extending through address 940.

When you allocate a variable in your application, the compiler reserves an explicit amount of memory for it. For example, a statement such as `int x = 45` causes the compiler to reserve 4 bytes of memory to store the current value of `x`. This is represented in figure 3.1 by the 4 bytes starting at address 928.

3.3.2 Obtaining the address of a variable

Referring to the name of a variable in an expression will access or update its current value. By placing the address-of (`&`) operator in front of a variable name, you can learn the address at which the variable is currently stored.

A variable that can store the address of another variable is called a *pointer* because it's said to "point to" the location of another value. The following code snippet demonstrates how you can use the address-of operator.

```
int x = 45;
int *y = &x;
```

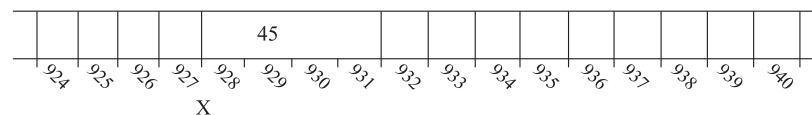


Figure 3.1 Representation of a region of the iPhone's memory showing the location of variable `x`

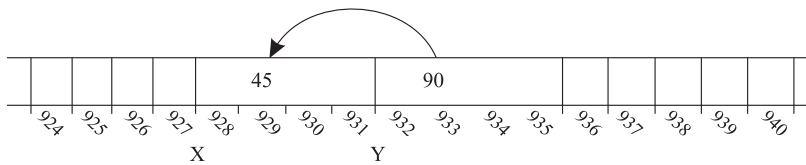


Figure 3.2 An updated memory map showing how variable `y` stores the address of variable `x`

This code snippet declares an integer variable `x` that's initialized to the value 45. It also declares variable `y` with a data type of `int *`. The `*` at the end of the data type indicates a pointer and means you don't want to store an actual integer value but rather the memory address at which one can be found. This pointer is then initialized with the address of variable `x` via use of the address-of operator. If variable `x` had been stored at address 928 (as previously mentioned), you could graphically represent the result of executing this code snippet by updating the memory map to be similar to that shown in figure 3.2.

Notice how the 4 bytes allocated to store variable `y` now store the number 928. When interpreted as an address, this indicates the location of variable `x`, as indicated by the arrow. The expression `y = &x` can be read as "place the address of variable `x` into variable `y`."

3.3.3 Following a pointer

Once you have an address stored in a pointer variable, it's only natural to want to determine the value of whatever it points to. This operation is called *dereferencing* the pointer and is also achieved by using the `*` symbol:

```
int x = 45;
int *y = &x;
NSLog(@"The value was %d", *y);
```

The statement on the last line prints out the message "The value was 45" because the `*` in front of variable `y` causes the compiler to follow the pointer and access the value it currently points to. In addition to reading the value, it's possible to replace it, as demonstrated next. Confusingly, this operation also makes use of the `*` operator:

```
int x = 45;
int *y = &x;

*y = 92;
```

The statement on the last line stores the value 92 at the address located in variable `y`. Referring to figure 3.2, you'll see that variable `y` stores (or points to) address 928, so executing this statement updates the value of variable `x` even though `x` is never explicitly referred to in the statement.

Arrays are pointers in disguise

A variable identifier for a C-style array can at some level be thought of as a simple pointer. This pointer always points to the first element in the array. As an example, the following is perfectly valid Objective-C source code:

```
int ages[50];
int *p = ages;
NSLog(@"Age of 10th person is %d", p[9]);
```

Notice you can assign the array variable to a pointer variable directly without using the `&` operator, and you can use the familiar `[]` syntax to calculate an offset from the pointer. A statement such as `p[9]` is another way to express `*(p + 9)`: it's a shorthand way to say "add 9 to the pointer's current value and then dereference it."

When working with pointers to structure-based data types, a special dereferencing syntax is available that allows you to dereference the pointer and access a specific field in the structure in a single step, using the `->` operator:

```
struct box *p = ...;
p->width = 20;
```

The `->` operator on the second line dereferences the pointer `p` and then accesses the `width` field in the structure. While following a pointer to read or alter the value it points at, it's sometimes helpful to compare two pointers to check if they point to identical values.

3.3.4 Comparing the values of pointers

When comparing the values of two pointers, it's important to ensure you're performing the intended comparison. Consider the following code snippet:

```
int data[2] = { 99, 99 };
int *x = &data[0];
int *y = &data[1];

if (x == y) { NSLog(@"The two values are the same"); }
```

You might expect this code to emit the message "The two values are the same" but it doesn't. The statement `x == y` compares the address of each pointer, and because `x` and `y` both point to different elements in the `data` array, the statement returns NO.

If you want to determine if the values pointed to by each pointer are identical, you dereference both pointers:

```
if (*x == *y) { NSLog(@"The two values are the same"); }
```

Now that you understand the concept of pointers and how multiple pointers can reference the same object, you're ready to communicate with the object. Communicating with an object enables you to interrogate it for details it stores or to request that the object perform a specific task using the information and resources at its disposal.

Indicating the absence of a value

Sometimes you want to detect if a pointer variable is currently pointing at anything of relevance. For this purpose, you'll most likely initialize the pointer to one of the special constants `NULL` or `nil`:

```
int *x = NULL;
NSString *y = nil;
```

Both constants are equivalent to the value 0 and are used to indicate that the pointer isn't currently pointing to anything. The Objective-C convention is to use `nil` when referring to an object and relegating `NULL` for use with older C-style data types.

Initializing the pointer to one of these special values enables an `if (y != nil)` check to determine if the pointer is currently pointing to anything. Because `nil` is equivalent to the value 0, you may also see this condition written as `if (!y)`.

Also be careful not to dereference a `NULL` pointer. Trying to read or write from a pointer that points to nothing causes an access violation error, which immediately exits your application.

3.4 Communicating with objects

In most C-inspired languages such as C++, Java, and C#, developers call a method implemented by an object. Developers using Objective-C, on the other hand, don't "call" a method directly; instead, they "send" a message to an object. The object "receives" the message and decides if it wants to process it, usually by invoking a method with the same name. This is a fundamentally different approach and is one of the many features that make Objective-C a more dynamic language because it enables the object to have finer-grained control over how method dispatch occurs.

3.4.1 Sending a message to an object

Figure 3.3 outlines the basic Objective-C syntax for sending a message to an object. In source code, a message send is represented by a set of square brackets; immediately after the opening bracket is the target, followed by the name of the message itself. The target can be any expression that evaluates to an object that should receive the message.

You can consider a basic message send such as the one shown in figure 3.3 as analogous to addressing an envelope with a person's name and address. It sets up a container that can then be delivered to its intended recipient.

Sometimes, for an object to make sense of a message, additional information must be sent along with it, similar to placing a letter or invoice in an envelope. Additional information provided in a message send is represented by one or more values called *arguments*. When sending a message, arguments are provided by placing a colon character after the method name, as shown in figure 3.4.

```
[ myObject myMessage ]:
```

Figure 3.3 The basic syntax for sending a message to an object in Objective-C. In a pair of square brackets, the target (or object that receives the message) is specified, followed by the name of the message itself.

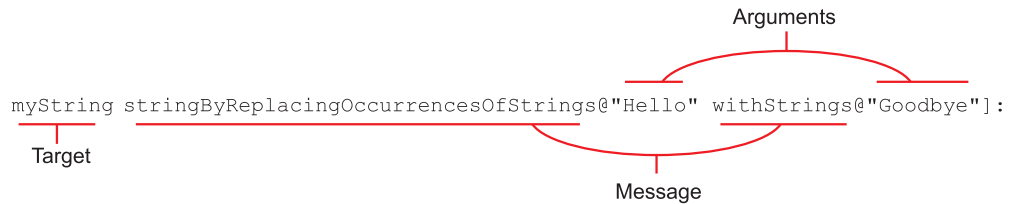


Figure 3.4 A slight change in syntax allows you to send a message that passes along one or more arguments for the object to use as it attempts to process the message.

When `myString` receives the message sent in figure 3.4, it also gains access to the values `@"Hello"` and `@"Goodbye"`. One interesting thing to note from figure 3.4 is how messages that expect multiple arguments are handled. The message sent in the figure is named `stringByReplacingOccurrencesOfString:withString:`; each colon character represents where an argument can be placed. It's common for Objective-C message names to be descriptive and verbose, almost making small sentences when combined with their arguments: figure 3.4 could be read as “string by replacing occurrences of string hello with string goodbye.”

3.4.2 Sending a message to a class

Say you have two objects of type `dog`, one called `Puggsie` and another called `Mr Puggington`. If you wanted both dogs to sit, you could individually send each object a message called `sit`. Both dogs would receive this message and (assuming they don't have minds of their own) sit down on request.

Sometimes you may want to request information or provide instructions that aren't specific to a particular instance of a class. For example, you may want to inquire about the number of dog breeds available. It doesn't make sense to ask either `Mr Puggington` or `Puggsie` (instances of the `Dog` class) this question, but the question is clearly related to the concept of `Dog`.

In such cases, it's possible to send a message directly to the class. The syntax for this is similar to sending a message to an object, but because you don't have a particular object to target the message to, you use the name of the class as the target:

```
int numberOfDogBreeds = [Dog numberOfDogBreeds];
```

Many requests can be made of `Puggsie` and `Mr Puggington`, but there are limitations to what you can request. For example, sending a `Dog` object (`Mr Puggington`, perhaps) a message called `makeThreeCourseDinnerForHumanOwner` isn't likely to result in a meal. Compilers for many languages can detect such infeasible method calls and halt compilation as soon as one is found. Objective-C, however, allows such infeasible requests to be made. Who knows—the dog may be the next Einstein.

3.4.3 *Sending nonexistent messages*

With the message send concept, it's possible to send an object a message that it may not be able to handle. Unlike in a language such as C++ where a method must be declared in order for it to be called, requesting that a message be sent is just that: a request.

Because of Objective-C's dynamic typing, there's no guarantee that the recipient of a message will understand how to process it, and no check is made at compile time to ensure it exists. The following code snippet will successfully compile even though the `NSString` class doesn't implement a method called `mainBundle`:

```
NSString *myString = @"Hello World!";
[myString mainBundle];
```

The best that the Objective-C compiler can do in a situation like this is produce the following warning during compilation:

```
'NSString' may not respond to '-mainBundle'
```

You may be wondering why Objective-C produces only a warning when other languages are more heavy handed and produce a fatal error during compilation. The answer is that Objective-C allows objects to dynamically add and remove methods and also gives them an opportunity to process any unhandled messages. Consequently, the compiler can never really be sure that a particular object won't respond to an arbitrary message without giving it a go at runtime.

If you want to get rid of the compile-time warning, the easiest way is to change the variable's data type to `id`:

```
id myString = @"Hello World!";
[myString mainBundle];
```

Alternatively, you could use a typecast:

```
NSString *myString = @"Hello World";
[(id)myString mainBundle];
```

Because the `id` data type can reference an object of any type, the compiler is more lenient with its warnings. If it didn't make this concession, it would have to warn about every message sent to an object typed as `id` because this type doesn't explicitly declare the existence of any methods to the compiler.

It's important to note that although all the code snippets in this section successfully compile, they will all fail at runtime because `NSString` doesn't handle a message called `mainBundle`. If the target object doesn't define a suitable method, the message will eventually be rejected, just as addressing an envelope to a nonexistent address will cause it to become undeliverable. You'll most likely come across this situation when you see the following fatal error in the debugger:

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException',
    reason: '*** -[NSString mainBundle]: unrecognized selector sent to
    instance 0x3b0a1e0'
```

This is the ultimate sign that Objective-C looked for a suitable method to handle this message and came up short. We return to this topic in chapter 8 where we discuss how a class can be modified to intercept unknown messages and perform tricks such as redirecting them to another method.

Although sending a message you know will result in a fatal exception isn't wise, this technique has some practical uses. It's heavily used by the Core Data framework discussed in chapter 12, for example.

Another unusual quirk of the message send implementation, one that's widely used, is sending a message to `nil`, which is similar in concept to mailing an envelope without an address.

3.4.4 Sending messages to `nil`

An interesting feature of Objective-C is the behavior of sending a message when the target is `nil`—when the message is addressed to no one. In many languages, doing so would produce a fatal error at runtime, typically called a NULL Reference exception, and to guard against such errors, developers commonly use code snippets similar to the following:

```
NSString *str = ...expression which returns a string or nil...;
NSUInteger length;

if (str != nil)
    length = [str length];
else
    length = 0;
```

The `if` statement checks to see if the `str` variable is currently pointing to a valid object and sends the `length` message (which returns the length of the string) only if a valid object is available. If `str` currently contains the value `nil`, indicating no object is available to send the message to, a default length of 0 is assumed. In Objective-C these checks are commonly not required, meaning you can simplify the code snippet as follows:

```
NSString *str = ...expression which returns a string or nil...;
NSUInteger length = [str length];
```

Both code snippets are identical in behavior. Looking at the simplified code snippet, you may wonder how it's safe to execute the statement `[str length]` when the target is `nil`. The answer is that, as part of the message send process, Objective-C internally checks if the target is `nil` and avoids sending the message. If the Objective-C runtime decides not to send the message, the return value will typically be 0 (or its equivalent, such as 0.0, or NO).

Now that you have a firm grasp of the concepts of object-oriented programming, the difference between objects and classes, and how to communicate with objects, let's investigate a common class undoubtedly used by almost every application: `NSString`, which represents a sequence of characters.

The `NSString` class provides a good example of the benefits of object-oriented programming. For starters, you don't need to write the code for this class: it's provided

for you out of the box, and you can reuse existing classes without needing to reinvent the wheel with each application. Reuse makes you more productive, allowing you to focus on differentiating and perfecting your application instead of spending days building a basic foundation.

3.5 Strings

The `NSString` class provides a nice object-oriented interface to represent, interact with, and modify a string of characters. Unlike in a C-style `char *` null-terminated string or `char[]` array, all aspects of memory allocation, text encoding, and string manipulation are hidden from the application developer as internal implementation details of the `NSString` class. This allows you to worry about the more important and unique aspects of your application logic rather than the nitty-gritty details of how strings are stored in a computer or how operations such as string concatenation operate.

This abstraction also means that common sources of error, such as attempting to store a 250-character string in a variable that has allocated space for only 200 characters, are easily avoided in Objective-C. Let's start our discussion of strings by learning how to create new strings in your source code.

3.5.1 Constructing strings

The easiest way to create a new `NSString` instance is via the convenient `@"..."` language syntax introduced in previous chapters. For example, the statement

```
NSString *myString = @"Hello, World!";
```

creates a new `NSString` object and initializes it with the value `"Hello, World!"`. This form of syntax, however, is a special case for the `NSString` class. It's impossible to use this syntax to create instances of any other class. The use of text strings in applications is so pervasive that the Objective-C language designers decided it was beneficial to have dedicated (and concise) syntax to create them. A more generic technique to construct an object of a particular class is to send that class an `alloc` message to allocate memory for a new object, followed by some kind of initialization message. For example, here's another way to create a string:

```
NSString *myString = [NSString alloc];  
myString = [myString initWithString:@"Hello, World!"];
```

This code sample explicitly demonstrates the two stages of constructing a new object in Objective-C: allocation and initialization. The first line sends the `alloc` (short for allocate) message to the `NSString` class. This causes `NSString` to assign enough memory to store a new string and returns a C-style pointer to that memory. At this stage, however, the object is rather blank, so the next step is to call an initialization method to initialize the object with some kind of sensible value.

Many classes provide a number of initialization messages. In this example, you sent the `initWithString:` message, which initializes the new string object with the contents

of a string constant. It's much more common to see both of these statements written on a single line by nesting one message send inside of another:

```
NSString *myString = [[NSString alloc] initWithString:@"Hello, World!"];
```

As an alternative to the alloc- and init-based object creation process, most classes also provide *factory methods* that allow you to perform both steps at the same time. The following snippet shows another way to create a new string:

```
NSString *myString = [NSString stringWithString:@"Hello, World!"];
```

This statement is identical in behavior to calling alloc, followed by initWithString:, but it's slightly easier to read and quicker to type. In general, many initialization messages named in the form initWithXYZ: have a matching classNameWithXYZ: factory method available that performs an implicit alloc behind the scenes. A subtle but important difference in memory management between the two techniques is discussed in depth in chapter 9. For now, let's stick to using factory methods.

Using your newfound knowledge of object construction techniques, you can look at some of the previous code samples in a new light. For example, in the current version of the Rental Manager application is the following line of source code:

```
cell.detailTextLabel.text =
    [NSString stringWithFormat:@"Rents for $%0.2f per week",
     properties[indexPath.row].weeklyRentalPrice];
```

With your knowledge of message-naming conventions, you can see that this statement creates a new string by sending the NSString class the stringWithFormat: message. This message constructs a new string by interpreting an NSLog-style format string. Now that you know how to create new string objects, let's look at how to work with them.

3.5.2 Extracting characters from strings

Once you create a string object, you're ready to interact with it by sending the object a message. Perhaps the simplest message you can send to a string is the length message, which returns the total number of characters in the string:

```
int len = [myString length];
NSLog(@"'%@' contains %d characters", myString, len);
```

Because a string is made up of a sequence of individual characters, another message that is useful is characterAtIndex:, which allows you to obtain the character at a particular index within a string:

```
NSString *myString = @"Hello, World!";
unichar ch = [myString characterAtIndex:7];
NSLog(@"The 8th character in the string '%@' is '%C'", myString, ch);
```

Notice that the characterAtIndex: message works in a fashion similar to the array indexing operator [], available for C-style arrays. If instead of a single character you want to obtain a particular range of characters within a string, you can use the stringWithRange: message, as demonstrated by the following listing.

Listing 3.1 Using `substringWithRange:` to obtain the last word in a string

```
NSString *str1 = @"Hello, World!";

NSRange range;
range.location = 7;           <----- or call NSRangeMake(7, 5);
range.length = 5;

NSString *str2 = [str1 substringWithRange:range];
NSLog(@"The last word in the string '%@' is '%@'", str1, str2);
```

The `substringWithRange:` message returns a new string made up of a sequence of characters obtained from the original string. The range is specified by providing an `NSRange`. This is a typedef for a small C-style structure that consists of a `location` and `length` field. The `location` field is the index of the first character in the string that should be returned; the `length` field indicates how many characters from that position should be included.

While extracting individual characters and substrings of a larger string is handy, how do you go about modifying the original string instead?

3.5.3 *Modifying strings*

`NSString` provides many messages that enable you to modify the contents of an existing string. For example, the following statement converts the contents of `str1` into lowercase form and stores the resultant string in the variable `str2`:

```
NSString *str1 = @"I am in MiXeD CaSe!";
NSString *str2 = [str1 lowercaseString];
```

Likewise, if you wanted to convert the string "Hello, World!" into "Hello, Chris!", you could use `stringByReplacingOccurrencesOfString:withString:` to replace any occurrence of "World" with "Chris":

```
NSString *str1 = @"Hello, World!";
NSString *str2 = [str1 stringByReplacingOccurrencesOfString:@"World"
                withString:@"Chris"];
```

Immutable vs. mutable objects

You may notice that statements such as `NSString *str2 = [str1 lowercase]` return a new string in lowercase form rather than modifying the existing contents of `str1` itself. After executing the statement, you end up with two strings, the original string in `str1` and a lowercase form in `str2`.

A string is an example of an immutable object. An object that is immutable can't be modified in any way after it has been initially created. The only way to modify an immutable object is to create an entirely new one and initialize it with the desired value.

The opposite of an immutable object is a mutable one, an object that can "mutate" or change its value. In many cases, Foundation Kit gives you the choice of immutable or mutable variants of a class. With strings, you have `NSString` and `NSMutableString`.

To append one string onto the end of another, you can use the `stringByAppendingString:` message:

```
NSString *str2 = [str1 stringByAppendingString:@"Cool!"];
```

This statement appends "Cool!" to the end of `str1` and stores the result in `str2`. You can't state `str2 = str1 + @"Cool!"` for reasons that will soon become clear.

Now that you know how to create and manipulate string objects, it's only natural to want to compare one against another to determine if they're identical.

3.5.4 Comparing strings

To compare two string variables against each other, you may come up with a code snippet such as the following based on what you've learned about pointers in this chapter:

```
NSString *str1 = [NSString stringWithFormat:@"Hello %@", @"World"];
NSString *str2 = [NSString stringWithFormat:@"Hello %@", @"World"];

if (str1 == str2) {
    NSLog(@"The two strings are identical");
}
```

Surprisingly, when this code is executed, you'll notice it indicates that the two freshly created strings aren't equal! To understand why, you need to realize that the two variables `str1` and `str2` are both simple C-style pointers. With its C-based roots, the `==` operator checks for equality by determining if both variables point to the same piece of memory. Because you create two separate string instances, the two pointers point to distinct locations, and the expression evaluates to `false`. That both strings contain an identical sequence of characters is irrelevant because the `==` operator only considers the string's memory location.

To work around this problem, Objective-C, like most object-oriented languages with pointers, provides a message to compare the contents of an object instead of its location in memory. In Objective-C this message is called `isEqual:`:

```
if ([str1 isEqual:str2]) {
    NSLog(@"The two strings are identical");
}
```

It's for a similar reason that the `+` operator can't be used for string concatenation: it's already used with C-style pointers for pointer arithmetic.

3.6 Sample application

To put the concepts you've learned in this chapter into practice, let's expand the Rental Manager application to make greater use of the services provided by `NSString` to search and modify string content. Figure 3.5 shows how the application will look once you finish this round of modifications. Each rental property has an image beside it that categorizes the property's location, such as near the sea, in the city, or in an alpine setting.

To display images in your iPhone application, the obvious first step is to include some image resources in your project. To add images to the Xcode project, drag and

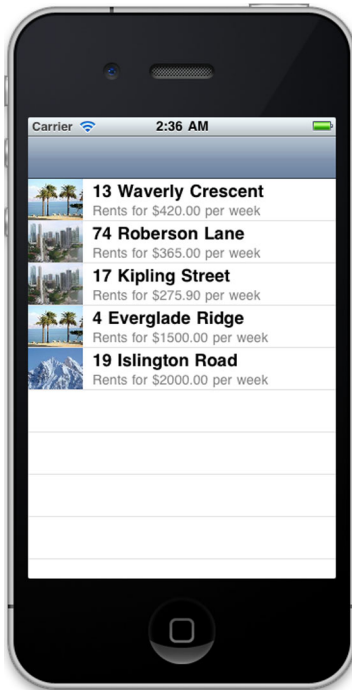





Figure 3.5 An updated version of the Rental Manager application. Notice each property has an image beside it indicating the type of geographical location of the property.

drop the image files from a Finder window into the Supporting Files group located in Xcode’s project navigator pane. A sheet will appear from which you can select a button labeled Add. For this task, you need to source three images. Details of the images we utilized are listed in table 3.1. You can use a service such as Google Image Search to source your own.

Table 3.1 The images used to represent different property locations. Selection of the image for a particular property is based on the city found in its associated address. Some city names that map to each image are provided as examples.

Image	Filename	Example cities
	sea.png	Sumner
	mountain.png	Clifton
	city.png	Riccarton, Christchurch

Now that you have the required image resources included in your project, you're ready to modify the application's source code to use them. Open the `RootViewController.m` file in the editor and replace the existing version of the `tableView:cellForRowAtIndexPath:` method with that in the following listing.

Listing 3.2 Replacement `tableView:cellForRowAtIndexPath:` method implementation

```

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:CellIdentifier] autorelease];
    }

    RentalProperty *details = &properties[indexPath.row];

    int indexOfComma = [details->address rangeOfString:@","].location;
    NSString *address = [details->address
        substringToIndex:indexOfComma];
    NSString *city = [details->address
        substringFromIndex:indexOfComma + 2];

    cell.textLabel.text = address;

    if ([city isEqual:@"Clifton"])
        cell.imageView.image = [UIImage imageNamed:@"mountain.png"];
    else if ([city isEqual:@"Sumner"])
        cell.imageView.image = [UIImage imageNamed:@"sea.png"];
    else
        cell.imageView.image = [UIImage imageNamed:@"city.png"];

    cell.detailTextLabel.text =
        [NSString stringWithFormat:@"Rents for $%0.2f per week",
            details->weeklyRentalPrice];

    return cell;
}

```

1 Determine semicolon location

2 Separate street address and city

3 Display proper image

Most of the code in listing 3.2 is similar to its previous incarnation, but some of the features discussed in this chapter are added.

To start with, pointers are used to avoid having to constantly retype the expression `properties[indexPath.row]` whenever you want to access details about the property for which you are currently generating a cell. Instead, calculate the expression once and store the memory address at which the resultant property can be found in the `details` variable. This pointer is then used throughout the rest of the method and dereferenced to access the various fields of rental property information you store. If you ever need to change or update how you determine the current property, you now need only do it in one place, and you get to type less to boot!

The hardest challenge in providing images for each rental property in the list is determining in which city each property is located. At this stage, you'll be fairly naive and split the address field on the sole semicolon it contains. In the future, you may like to revisit this solution and find a more robust mechanism or use some of the iPhone SDK geocoding APIs.

Using the `rangeOfString:` message available on the `NSString` class, ❶ you can determine the location of the semicolon in the property's address, and by passing this index into additional `NSString` methods, `substringFromIndex:` and `substringToIndex:`, you can separate the street address and city into two separate strings ❷. In the call to `substringFromIndex:` you have to add 2 to the index returned by `rangeOfString:` to skip over the semicolon and the space that follows it.

Now that you've extracted a rental property's city from its address, you're ready to determine which image to display and attach it to the table view cell ❸. You do this by comparing the contents of the `city` variable against a couple of hardcoded city names. Once you determine the type of location, use `UIImage`'s `imageNamed` functionality to load the images you previously included in the application.

One obvious problem with the technique demonstrated in this revision of the Rental Manager application is that the list of cities and their mappings to location types is hardcoded and specified in code. This makes it difficult and more time consuming (especially with the App Store submission process) to update the behavior of the application as your rental portfolio expands and covers properties in other cities. Ideally, you want to separate the logic from the data so you can easily update the city mappings without needing to recompile or to resubmit the application: more on this in the next chapter.

3.7 **Summary**

Enhancing the procedural C language to have object-oriented features is essentially what brought Objective-C to life. The benefits of developing applications in an object-oriented manner generally far outweigh the extra effort required to learn the additional terminology and techniques that object-orientation entails.

Chief among the advantages of object-oriented programming is an improved ability to separate a complex application into a number of smaller, discrete building blocks, or classes. Rather than considering a large, complex system, the developer's task becomes one of developing multiple smaller systems that combine and build on top of each other to perform tasks far more complex than any one part can do alone.

The ability to package data plus logic into modules, called classes, also makes it easier to transplant an object designed and developed in one application into another. Application frameworks such as Cocoa Touch and Foundation Kit take this to another level. Their sole purpose is to provide developers with a large number of classes out of the box and ready to be put to work in their applications. Developers need not spend time ironing out bugs and quirks in the 900th string concatenation implementation the world has seen. Application developers can instead focus on the distinct features of their own applications.

In chapter 4, we continue our discussion of how various classes in Foundation Kit improve upon the basic data types provided by the procedural-based C language. We look at the collection classes such as `NSArray` and `NSDictionary`, which are designed to replace and improve upon C-style arrays.

Storing data in collections

4

This chapter covers

- NSArray
- NSDictionary
- Storing `nil` and other special values in collections
- Boxing and unboxing non-object-based data

Chapter 3 introduced the concept of object-oriented programming and demonstrated some of its advantages by using the services of the prebuilt `NSString` class, which provides common text manipulation and query-related functionality.

A large part of Foundation Kit is concentrated on the storage of data in collection data structures with names such as arrays, dictionaries, sets, and hashmaps. These data structures allow you to efficiently collect, group, sort, and filter data in a way that's convenient for the application at hand. Discussing these commonly used classes is what this chapter is all about.

Let's start by discussing how Foundation Kit improves upon a data structure you're currently familiar with: a simple array.

4.1 Arrays

The Rental Manager application, as it currently stands, stores a list of rental property details in a C-style array called *properties*. Inherently, nothing's wrong with this technique, but it has some limitations. For example, when you declare a C-style array, you create it with a fixed number of elements, and it's not possible to add or remove additional elements from the array without recompiling the application. Your rental management business may become more successful and require the ability to add new properties to the list at runtime.

The Foundation Kit provides a convenient array-like data structure called `NSArray` that overcomes this and other limitations. An `NSArray` is an ordered collection of objects, just like a C-style array, except it also has the ability to grow or shrink as required.

Like strings, arrays can be immutable or mutable. Immutable arrays are handled by the `NSArray` class, and mutable ones are handled by the `NSMutableArray` subclass. Let's learn how creating an `NSArray` instance is different from creating a C-style array.

4.1.1 Constructing an array

You can create a new `NSArray` in various ways, depending on your needs. If you want to create an array consisting of a single element, the easiest way is to use the `arrayWithObject:` factory message:

```
NSArray *array = [NSArray arrayWithObject:@"Hi"];
```

This statement creates an array consisting of a single element, in this case the string "Hi", although it could be any other object, even another `NSArray` instance, for example. A similarly named factory message, `arrayWithObjects:`, can be used for the more typical scenario of initializing an array with more than one element:

```
NSArray *array = [NSArray arrayWithObjects:@"Cat", @"Dog", @"Mouse", nil];
```

`arrayWithObjects:` is an example of a *varadic* method. A varadic method is a method which expects a variable number of arguments. Vardic methods must have some way to know when they've reached the end of the list of arguments. `arrayWithObjects:` detects this condition by looking for the special value `nil`. An array can't store a `nil` value, so its presence indicates the end of the list. It's an error to call `arrayWithObjects:` without the last argument being `nil`. If `arrayWithObjects:` is causing your application to crash, make sure you've got a `nil` at the end.

When bridging between C-based code and Objective-C, you may come across the need to convert a C-style array into an `NSArray` instance. `NSArray` provides a factory method to make this task easy:

```
NSString *cArray[] = {"Cat", "Dog", "Mouse"};
NSArray *array = [NSArray arrayWithObjects:cArray count:3];
```

The `arrayWithObjects:count:` message creates a new `NSArray` instance and initializes it with a copy of the first `count` elements from the C-style array. This trick isn't the only one available in `NSArray`'s toolbox. For example, another factory message called

`arrayWithContentsOfURL:` allows you to populate an array with the contents of a file fetched from the internet.

```
NSArray *array = [NSArray arrayWithContentsOfURL:
    [NSURL URLWithString:@"http://www.christec.co.nz/example.plist"]];
```

This code snippet fetches the file located at `http://www.christec.co.nz/example.plist` and expects it to be an XML file that conforms to the Property List (plist) schema. An example plist file containing an array is displayed in the following listing.

Listing 4.1 A property list XML file describing an array of three elements

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
    DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <array>
    <string>Cat</string>
    <string>Dog</string>
    <integer>42</integer>
  </array>
</plist>
```

Another important thing demonstrated by this plist file is that `NSArray`-based arrays don't have to be homogeneous. It's possible for each element in an `NSArray` to have a different data type. This is unlike a C-style array in which each array element must be of the same data type. The plist in listing 4.1 creates an array that consists of two strings and an integer.

All the arrays demonstrated so far are immutable (once they're created, it's impossible to add or remove elements or even replace existing ones) because you used `NSArray`, which represents an immutable array. To create an array that can have elements added, removed, or updated, you use the `NSMutableArray` class. As an example, each of the following two statements creates a new array consisting of one element.

```
NSArray *array1 = [NSArray arrayWithObject:@"Hi"];
NSMutableArray *array2 = [NSMutableArray arrayWithObject:@"Hi"];
```

The only difference between the two arrays is their immutability. `array1` is effectively read-only and allows no modifications, while `array2` allows you to add and remove elements from the array to your heart's content. Let's leave the topic of array creation and learn how to interact with the various elements contained in them.

4.1.2 Accessing array elements

With your existing knowledge of C, it's not easy to determine the number of elements in a C-style array, but the `NSArray` and `NSMutableArray` classes provide a straightforward way to determine the number of elements stored in them. You just send the instance a count message:

```
int numberOfItems = [myArray count];
```

On return, the variable `numberOfItems` tells you how many elements are in the array. To access each element in an `NSArray`, you can use another message called `objectAtIndex:`, which behaves similarly to the `[]` indexing operator used with C-style arrays:

```
id item = [myArray objectAtIndex:5];
```

You could imagine this last statement as being equivalent to `myArray[5]` if `myArray` had instead been declared as a C-style array. The intent and behavior is identical. Using the techniques you learned in this section, you could access the last element of an array with the following code snippet:

```
int indexOfLastItem = [myArray count] - 1;
id item = [myArray objectAtIndex:indexOfLastItem];
```

You could even condense this into a single statement by nesting the two method calls:

```
id item = [myArray objectAtIndex:[myArray count] - 1];
```

The `-1` in the index calculation accounts for the fact that the last element in the array will have an index of 1 less than the number of items in the array. Because this is a common code pattern and Foundation Kit is all about developer efficiency, `NSArray` provides a `lastObject` message that performs the same task in a cleaner manner:

```
id item = [myArray lastObject];
```

Using this concept of simplifying common coding tasks, let's investigate some of the other aspects of the `NSArray` class.

4.1.3 Searching for array elements

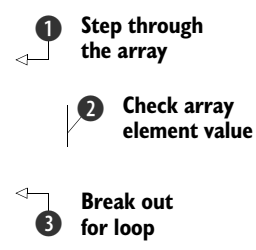
You now have the building blocks for determining if a particular value is present in an array. To achieve this goal, you can use a `for` loop to step through each element in the array and use an `if` statement to compare the current element with the value you're looking for. This process is demonstrated in the following listing.

Listing 4.2 Determining if an NSArray contains the word "Fish" using C-style code

```
NSArray *pets = [NSArray arrayWithObjects:@"Cat", @"Dog", @"Rat", nil];
NSString *valueWeAreLookingFor = @"Fish";

int i;
BOOL found = NO;
for (i = 0; i < [pets count]; i++)
{
    if ([[pets objectAtIndex:i]
        isEqual:valueWeAreLookingFor])
    {
        found = YES;
        break;
    }
}

if (found)
{
```



```

        NSLog(@"We found '%@' within the array",
              valueWeAreLookingFor);
    } else {
        NSLog(@"We didn't find '%@' within the array",
              valueWeAreLookingFor);
    }
}

```

In this listing you set up a loop ❶ to step through each element in the array. The loop sets the loop counter variable `i` to the values 0 through `[pets count] - 1`. In the loop you fetch the array element at index `i` via the `objectAtIndex:` method and check to see if it's equal to the value you are looking for ❷. If a match is found, you set the found variable to YES ❸ and break out of the `for` loop immediately, because there's no need to check any remaining array elements.

We think that's a lot of code for such a simple task! It also allows ample opportunity for subtle bugs to creep in. Luckily, Foundation Kit provides a much more convenient technique to check if an array contains a specific value in the form of `NSArray`'s `containsObject:` message:

```

BOOL found = [pets containsObject:@"Fish"];
if (found) {
    NSLog(@"We found 'Fish' within the array");
}

```

Internally, `containsObject:` performs a similar search through the array, comparing each element as it goes; however, this action is hidden from you, and more important, there's no opportunity for you to introduce errors. This technique demonstrates another advantage of using objects: the ability to easily reuse methods instead of continually writing them from scratch.

Sometimes you're interested in determining not only if a value exists in an array but also its position. The `indexOfObject:` message performs a similar task to `containsObject:`, but instead of returning a Boolean flag to indicate if the object is found, it returns the index at which the item is found or a special value `NSNotFound` if it's not found:

```

int indexOfItem = [pets indexOfObject:@"Dog"];
if (indexOfItem != NSNotFound) {
    NSLog(@"We found the value 'Dog' at index %d", indexOfItem);
}

```

Although messages such as `indexOfObject:` and `containsObject:` allow you to remove, or at least hide, logic that loops over each element in an array, some cases may require such logic. For example, you may want to convert the name of each pet into uppercase form, and `NSArray` has no built-in method to achieve this task. But Objective-C and Foundation Kit provide mechanisms to perform iteration over an array more efficiently and safely than was done in listing 4.2.

4.1.4 Iterating through arrays

Using the `count` and `objectAtIndex:` messages of the `NSArray` class, you can loop over each element in an array with a code snippet similar to the following:

```
int i;
for (i = 0; i < [pets count]; i++) {
    NSString *pet = [pets objectAtIndex:i];

    NSLog(@"Pet: %@", pet);
}
```

This code may not be the most efficient or cleanest of techniques. Each iteration through the loop, when it's time to evaluate the condition `i < [pets count]`, you are asking the array to recalculate its length; and depending on how the array is internally stored (a dynamic array or linked list), repeatedly calling `objectAtIndex:` may not be efficient because of repeated walks through the data structure to reach the required element. One solution Objective-C provides for more efficient iteration through a data structure is the enumerator.

NSENUMERATOR

An *enumerator* is an object that allows you to step through a sequence of related items in an efficient manner. In Objective-C an enumerator is represented by the `NSEnumerator` class. Because an enumerator must step through a set of data, you typically ask a data structure such as an array to create a suitable enumerator on your behalf rather than create an `NSEnumerator` instance directly.

As an example, the following code snippet uses `NSArray`'s `objectEnumerator` message to obtain an `NSEnumerator` instance that enables you to step through each type of pet stored in the `pets` array:

```
NSArray *pets = [NSArray arrayWithObjects:@"Cat", @"Dog", @"Rat", nil];
NSEnumerator *enumerator = [pets objectEnumerator];
NSString *pet;

while (pet = [enumerator nextObject]) {
    NSLog(@"Pet: %@", pet);
}
```

The `objectEnumerator` method returns an instance of the `NSEnumerator` class that's suitable for enumeration of each element in the specified array.

`NSEnumerator` is a simple class that provides a single method called `nextObject`. This method, as its name suggests, returns the next object in the sequence the enumerator is enumerating over. By placing it in a `while` loop, you'll eventually be provided with the value of each element in the array. Once you reach the end of the sequence, `nextObject` returns `nil` to indicate the end of the sequence (another reason `NSArray` can't store the value `nil`). Executing the code snippet on the array of pet types results in output similar to the following:

```
Pet: Cat
Pet: Dog
Pet: Rat
```

Notice that at no point do you specify in which order you want to step through the array. This logic is inherently built into the particular `NSEnumerator` instance you obtain. Different versions of `NSEnumerator` may step through the sequence in different orders.

As an example, try replacing the existing call to `objectEnumerator` with a call to `reverseObjectEnumerator`. Both `NSArray` messages provide `NSEnumerator` instances, but the enumerator provided by `reverseObjectEnumerator` steps through the array in reverse order, starting at the last element and working toward the first.

Using an enumerator can be efficient because the `NSEnumerator` instance can keep track of additional state about the internal structure of the object it's enumerating, but it's not the only weapon available in Objective-C's toolbox of performance tricks.

FAST ENUMERATION

As the name suggests, fast enumeration is a feature designed to make the process of enumeration even faster and more efficient. It's one part Objective-C language syntax and one part runtime library support. For fast enumeration, you can use a special form of the `for` statement:

```
NSEnumerator enumerator = [pets reverseObjectEnumerator];
for (NSString *pet in enumerator) {
    NSLog(@"Next Pet: %@", pet);
}
```

The syntax is cleaner and more concise, but how does it work? In the brackets of the `for` statement, you declare a new variable followed by the keyword `in` and the object to which you want to apply fast enumeration. Each time through the loop, the `for` statement assigns the next element in the enumeration to the specified variable.

Fast enumeration can be used with any `NSEnumerator`, and it can also be used directly on an instance of the `NSArray` or `NSMutableArray` class. For example, the following snippet also works and leads to some very clean code:

```
for (NSString *pet in pets) {
    NSLog(@"Next Pet: %@", pet);
}
```

Fast enumeration can't be used with every object, however. It requires the object that is after the `in` keyword to implement the `NSFastEnumeration` protocol. Protocols are discussed in detail in chapter 7.

4.1.5 Adding items to an array

If you create an instance of the `NSArray` class, the array is said to be *immutable* because you can't modify the structure or contents of the array after it's been initially constructed. If you attempt to modify an `NSArray` instance, you'll get an exception message such as the following listed to the Xcode debugger console, and your application will crash:

```
*** Terminating app due to uncaught exception
    'NSInternalInconsistencyException', reason: '*** - [NSCFArray
    replaceObjectAtIndex:withObject:]: mutating method sent to immutable
    object'
```

If you need to modify an array after it's been created, you must make sure you create a mutable array. A good way to get started is by creating an empty array using the array factory method on the `NSMutableArray` class instead of the `NSArray` class. This creates an array that's initially empty, but because the array is mutable, you'll be able to append new elements to it at runtime as desired.

```
NSMutableArray *array = [NSMutableArray array];
```

An `NSMutableArray` grows in size whenever you add elements to it. You can conceptually think of the array as allocating additional memory to the array to store each element as it's added. But this isn't the most efficient way to use memory. If you know ahead of time how many elements you intend to add to the array, it's more efficient for the array to allocate enough memory to store all the elements in one go. As you may expect, `NSMutableArray` is smart: it provides another factory method called `arrayWithCapacity:` for this very case:

```
NSMutableArray *pets = [NSMutableArray arrayWithCapacity:50];
```

This code snippet creates a new `NSMutableArray` and internally allocates enough memory to store a minimum of 50 elements. It's important to note, however, that this is only a hint as to the number of items expected to eventually be added to the array. An array created via such a technique is perfectly able to store more than 50 elements and will allocate additional memory once the specified capacity is exceeded.

It's also important not to confuse the capacity of an array with its length or count. Although the array just created has memory set aside to store at least 50 elements, if you ask for its current size via the `count` message, it will indicate 0 because you still haven't physically added anything to it. The capacity is only a hint that allows the `NSMutableArray` class to avoid excessive memory allocations. Specifying a capacity of 50 means there won't be any additional memory allocations until at least the 51st element is added to the array. Doing this may lead to potential performance gains and less memory fragmentation.

Now that you know how to create an `NSMutableArray` instance that allows you to dynamically modify its contents, how do you add additional elements to an array? One answer is to use the `addObject:` message, which allows you to add a new element to the end of an existing array:

```
[pets addObject:@"Pony"];
```

This snippet expands the size of the array by one element and stores the string "Pony" in the newly added element. It's also possible to insert an element into the middle of an array. For this, you can use `insertObject:atIndex:`, which expects the array index at which to insert the new element:

```
[pets insertObject:@"Hamster" atIndex:2];
```

When this statement is executed, every element starting from array index 2 is shifted one place higher: the object at index 2 becomes the element at index 3, and so on. The vacated space at index 2 is then filled by inserting the string "Hamster". Instead of

inserting a new array element, you can replace an existing one by using the `replaceObjectAtIndex:withObject:` message:

```
[myArray replaceObjectAtIndex:0 withObject:@"Snake"];
```

This statement replaces the first element in the array with the string "Snake". One final operation worth noting is how to reduce the size of an array by removing existing elements. You can do this by providing the `removeObjectAtIndex:` message with the index of the element you want to remove:

```
[myArray removeObjectAtIndex:5];
```

This statement causes the element at index 5 to be removed from the array, and all elements after it are moved one index position to reclaim the “hole” left in the array. The array is hence reduced in length by one, as can be verified by a call to the `count` method.

Arrays are useful data structures with many practical applications, but they’re not the most flexible of data structures. A more flexible data structure is the dictionary.

4.2 Dictionaries

An array is only one form of data structure provided by Foundation Kit. Another useful data structure is the dictionary or map. A dictionary is a collection of key/value pairs. You use one value, termed a *key*, to look up a related value of interest.

In Foundation Kit, a dictionary is represented by the `NSDictionary` and `NSMutableDictionary` classes. Each entry you place in a dictionary consists of a key and its matching value—much like a physical dictionary uses a word as a key and a brief definition as a value, or a phone book matches names to numbers.

In a dictionary each key must be unique; otherwise confusion may occur when a key is provided and multiple matching values are found. If you must store multiple values against a given key, you can always store an `NSArray` instance as your single value. A key can be any object: one dictionary might use numbers, while another might use strings.

If you have a computer science background, you may be more familiar with the concept of a hash table and hashing functions. A hash table is one way in which the abstract concept of a dictionary can be implemented, but `NSDictionary` and `NSMutableDictionary` insulate you from such implementation minutiae and allow you to concentrate on their practical benefits to your application instead of on how they’re physically implemented.

Let’s start our investigation of dictionaries by learning how to create a new one and populate it with some initial entries.

4.2.1 Constructing a dictionary

The distinction between the immutable `NSDictionary` class and mutable `NSMutableDictionary` classes is similar to that between `NSArray` and `NSMutableArray`. An `NSDictionary` can’t be modified, whereas an `NSMutableDictionary` can freely have

new entries added, removed, or updated. To create an empty dictionary, you use the dictionary factory message:

```
NSMutableDictionary *myDetails = [NSMutableDictionary dictionary];
```

This message is primarily of use only with the `NSMutableDictionary` subclass. Otherwise, your empty dictionary will forever stay empty because it's immutable!

A message called `dictionaryWithObject:forKey:` enables you to create a dictionary that initially consists of a single key/value pair:

```
NSMutableDictionary *myDetails = [NSMutableDictionary dictionaryWithObject:@"Christopher"
                                                                    forKey:@"Name"];
```

This code creates a new dictionary containing a single entry consisting of the key "Name" with a value of "Christopher", both of which are strings. More than likely, however, you'll want to initialize a dictionary with multiple key/value pairs. A similarly named `dictionaryWithObjects:forKeys:` message allows for doing so:

```
NSArray *keys = [NSArray arrayWithObjects:@"Name", @"Cell", @"City", nil];
NSArray *values = [NSArray arrayWithObjects:@"Christopher", @"+643123456",
                                           @"Christchurch", nil];
NSMutableDictionary *myDetails = [NSMutableDictionary dictionaryWithObjects:values
                                                                    forKeys:keys];
```

In this example, you create a new dictionary with details about a particular person. The `dictionaryWithObjects:forKeys:` message expects to be provided with two arrays of equal length. The first value from the keys array is matched up with the first value from the values array, and so on, to create the key/value pairs that will populate the dictionary.

The creation of temporary arrays can be tiresome, especially if your only intent is to populate a dictionary and you don't need the arrays for other purposes. Naturally, the designers of the `NSMutableDictionary` class considered this scenario, and they provided a more convenient factory method to allow you to specify multiple key/value pairs without creating temporary arrays:

```
NSMutableDictionary *myDetails = [NSMutableDictionary dictionaryWithObjectsAndKeys:
    @"Christopher", @"Name",
    @"+643123456", @"Cell",
    @"Christchurch", @"City",
    nil];
```

This message expects a variable number of parameters to be provided to it. The parameters alternate between being interpreted as a value or a key and are matched up into pairs until a `nil` value is detected to indicate the end of the list. Other than providing an alternative way to specify the list of key/value pairs, `dictionaryWithObjectsAndKeys:` and `dictionaryWithObjects:forKeys:` perform identical functionality.

A number of other factory methods are also available on the `NSMutableDictionary` and `NSMutableDictionary` classes. For example, `dictionaryWithContentsOfURL:` performs a function similar to `NSArray's arrayWithContentsOfURL:` and enables a dictionary to easily be populated with contents of a file located on a website.

Due to the use of the `NSDictionary` class, all of the dictionaries constructed by the code samples in this section result in immutable dictionaries, which are read-only. If you want to create a dictionary that can be modified after creation, you need to replace the class name `NSDictionary` with `NSMutableDictionary`.

Now that you can construct dictionary instances, let's proceed to determine how to query a dictionary for details about the key/value pairs it contains.

4.2.2 *Accessing dictionary entries*

The methods for interacting with a dictionary are similar to those for an array. For example, you can send the `count` message to determine how many entries are currently contained in the dictionary:

```
int count = [myDetails count];
NSLog(@"There are %d details in the dictionary", count);
```

Rather than an `objectAtIndex:` message, which accesses an element by index position, you're provided with a similar message called `objectForKey:`, which enables you to obtain the value associated with a given key:

```
NSString *value = [myDetails objectForKey:@"Name"];
NSLog(@"My name is %@", value);
```

This statement searches the dictionary to determine if a key named "Name" exists and then returns the value associated with it. Unlike `NSArray`'s `objectAtIndex:` message, it's not an error to provide a key to `objectForKey:` that doesn't exist in the dictionary. In this case, the special value `nil` will be returned, indicating that the key was not found in the dictionary.

One common use of dictionaries is to store related but flexible data about a given item. For example, the dictionaries you constructed over the previous couple of pages store various bits of information about a particular person, such as name, cellphone number, and location. When interacting with such dictionaries, you'll likely want to query the value of multiple keys in quick succession. For example, if you were printing address labels, you would probably want the person's name and location details. Although you could make multiple calls to `objectForKey:`, you can also perform multiple lookups in a single statement via the `objectsForKeys:notFoundMarker:` message:

```
NSArray *keys = [NSArray arrayWithObjects:@"Name", @"City", @"Age", nil];
NSArray *values = [myDetails objectsForKeys:keys notFoundMarker:@"???"];

NSLog(@"%@ is located in %@ and is %@ years old",
      [values objectAtIndex:0],
      [values objectAtIndex:1],
      [values objectAtIndex:2]);
```

`objectsForKeys:notFoundMarker:` expects to be provided with an array consisting of the keys of the dictionary entries you want to query. It then returns an array with the values of those keys. If a particular key you request isn't present in the dictionary (such

as the person's age in this example), the value you provide via the `notFoundMarker` argument is placed in the array. The returned array has a one-for-one correspondence to the keys array that you pass in, so the value for the key specified in array element 0 of the keys array will be found in array element 0 of the returned values array.

Now that you have a strong handle on how to access and query the existing contents of a dictionary, let's look at how you can manipulate the contents of a dictionary by adding and removing additional key/value pairs.

4.2.3 Adding key/value pairs

Assuming you've created a mutable dictionary of type `NSMutableDictionary`, you can store additional key/value pairs in the dictionary by using the `setObject:forKey:` message:

```
[myDetails setObject:@"Wellington" forKey:@"City"];
```

If the key already exists in the dictionary, the previous value is discarded and the specified value takes its place. It's an error for the value or key arguments to be `nil` because, as previously discussed, messages such as `objectForKey:` use `nil` as a special value to indicate the absence of an entry.

If you have an existing dictionary of key/value pairs, it's possible to merge those entries into another dictionary by way of the `addEntriesFromDictionary:` message, as follows:

```
NSDictionary *otherDict = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Auckland", @"City",
    @"New Zealander", @"Nationality",
    @"Software Developer", @"Occupation",
    nil];

[myDetails addEntriesFromDictionary:otherDict];
```

This code adds each key/value pair found in `otherDict` into `myDict`, replacing the value of any key/value pairs that were already present. Instead of adding new key/value pairs, you can remove one by passing its associated key to the `removeObjectForKey:` message.

```
[myDetails removeObjectForKey:@"Name"];
```

This code deletes the entry associated with the key "Name". If you have a number of entries you want to delete, you can store the keys in an array and use the handy `removeObjectsForKeys:` message to delete them in one step. As an example, you could remove the details identifying the person's location by removing the key/value pairs identified by the strings "City" and "Nationality" using the following statement:

```
NSArray *locationRelatedKeys =
    [NSArray arrayWithObjects:@"City", @"Nationality", nil];

[myDetails removeObjectsForKeys:locationRelatedKeys];
```

Finally, it's possible to empty the dictionary completely by calling the `removeAllObjects` method, which, as its name suggests, deletes every key/value pair from the dictionary:

```
[myDetails removeAllObjects];
NSLog(@"There are %d details in the dictionary", [myDetails count]);
```

This code snippet results in the string "There are 0 details in the dictionary" because the call to `removeAllObjects` emptied it completely.

One message that can easily be confused with `setObject:forKey:` is the message `setValue:forKey:`. Although these two messages have similar names, their behavior is subtly different with regard to how they handle the special value `nil`, which, as you may recall, typically represents the absence of a value.

As the following code snippet demonstrates, both `setObject:forKey:` and `setValue:forKey:` can be used to update the value stored in a dictionary for a particular key:

```
[myDetails setObject:@"Australian" forKey:@"Nationality"];
[myDetails setValue:@"Melbourne" forKey:@"City"];
```

These two messages diverge in how they handle a `nil` value being specified for the key's value. Sending `setObject:forKey:` with the object parameter specified as `nil` results in an exception that crashes your application because `nil` isn't a valid value to store in a dictionary. Passing `nil` to `setValue:forKey:` is acceptable, however, because it is interpreted as though you had instead called

```
[myDetails removeObjectForKey:@"Nationality"];
```

Although it hardly makes a difference in the example code snippets (and arguably makes the code harder to understand), the benefits of using `setValue:forKey:` to remove a key/value pair really come to light when cleaning up a code snippet such as the following:

```
NSString *myNewValue = ...get new value from somewhere...
if (myNewValue == nil)
    [myDetails removeObjectForKey:@"Nationality"];
else
    [myDetails setObject: myNewValue forKey:@"Nationality"];
```

With `setValue:forKey:`, this entire code snippet can be replaced with a single line of source code. `setValue:forKey:` will be discussed in greater detail in chapter 11, which introduces the concept of Key-Value Coding. Throughout the rest of the book, we come back to the subject of dictionaries, but let's round off this discussion with a look at how to list all entries in a dictionary, similar to an index or table of contents in a book.

4.2.4 *Enumerating all keys and values*

Like arrays, dictionaries can be enumerated to list all of the key/value pairs contained in them. But because dictionaries don't retain any ordering, the order in which key/value pairs are enumerated may not match the sequence in which they were added to the dictionary.

Because a dictionary is made up of key/value pairs, `NSDictionary` provides two convenient messages to obtain enumerators to iterate over each entry in the dictionary. The `keyEnumerator` message provides an enumerator that iterates over all keys in the dictionary, while `objectEnumerator` performs a similar task but iterates over all the values instead:

```
NSEnumerator *enumerator = [myDetails keyEnumerator];
id key;

while (key = [enumerator nextObject]) {
    NSLog(@"Entry has key: %@", key);
}
```

Using the `keyEnumerator` message, this code snippet lists the name of each key currently stored in the dictionary. Try using the `objectEnumerator` message on the first line, and you'll see the values listed instead. Fast enumeration is also possible, and if you use the dictionary object directly, it iterates over all of the keys:

```
for (id key in myDetails) {
    id value = [myDetails objectForKey:key];

    NSLog(@"Entry with key '%@' has value '%@'", key, value);
}
```

This code sample also demonstrates that in the iteration loop it's possible to use `objectForKey:` to obtain the value that matches the current key. When placing code in an enumeration loop, such as the one just shown, you must be careful that you don't mutate the data structure. As an example, you may think that one way to remove all the entries in a dictionary (other than the more logical `removeAllObjects`) would be the following code snippet:

```
for (id key in dictionary) {
    [dictionary removeObjectForKey:key];
}
```

Although this appears conceptually correct, the code snippet has a fundamental flaw that will result in an exception similar to the following and crash the application:

```
*** Terminating app due to uncaught exception 'NSGenericException', reason:
    '*** Collection <NSCFDictionary: 0x3b11900> was mutated while being
    enumerated.'
```

If you enumerate over a data structure by using fast enumeration or an `NSEnumerator`, your code shouldn't modify the associated data structure until your enumeration has completed. Modifying the contents of an array or dictionary that's currently being enumerated will alter its internal data structure and cause any associated enumerators to become invalid. If you need to enumerate through a dictionary and potentially add or remove entries before enumeration has completed, you should first make a copy of all the keys by calling a method such as `allKeys`. This method creates an array and copies a list of all keys currently in the dictionary into that array. This allows you to

create a snapshot of the dictionary's keys and then enumerate through the snapshot while modifying the original dictionary. Here's an example of this process:

```
NSArray *myKeys = [myDetails allKeys];
for (NSString *key in myKeys) {
    [myDetails removeObjectForKey:key];
}
```

This works because the `allKeys` message creates a copy of the keys in the `myDetails` dictionary. The enumerator then loops through the contents of the array (and not the dictionary). Because the array is never modified, it's safe to enumerate its contents. Each time through the loop, you modify the dictionary, indirectly causing its internal data structures to change. This doesn't matter, however, because technically you're not currently enumerating its contents.

While discussing arrays and dictionaries, we casually mentioned that various messages won't accept `nil` or that they use `nil` to indicate special conditions. We also mentioned that these data structures are only capable of storing objects, not primitive values such as integer or float. But most applications need to store a list of numbers, so how can you force an `NSArray` or `NSDictionary` instance to store a set of primitive values? The answer is a technique called *boxing*.

4.3 **Boxing**

Undoubtedly in your application you'll come across the need to store a number such as 3, 4.86, or even a Boolean YES or NO value in an `NSArray`- or `NSDictionary`-based data structure. You may think you could accomplish this task with a code snippet such as the following:

```
[myArray addObject:5];
```

But when you attempt to compile this statement, the compiler will warn you that "passing argument 1 of `addObject:` makes pointer from integer without a cast," hinting that something isn't quite right. This example is a classic demonstration of the procedural C-based side of Objective-C running up against the newer object-oriented additions to the language.

Classes such as `NSArray` and `NSDictionary` expect their keys and values to be objects of some kind. The integer number 5 isn't an object: it's a simple primitive data type. As such, it's not possible to directly store an integer in an array.

Many languages such as Java 5 and C# take care of this problem automatically through a concept called *autoboxing*. Behind the scenes, the primitive (non-object-oriented) value is wrapped inside a container object (a box), and this container is passed around instead of the raw value. Likewise, when you attempt to access the array and extract the value, the compiler detects a boxed value and automatically extracts the primitive payload for you. You as a developer are none the wiser to this process occurring. This process can be conceptualized in much the same way as a gift being placed inside a FedEx box and removed once it reaches its destination. FedEx can only deal with boxes of certain sizes, not your oddly shaped teapot for Aunt Betty, but

that doesn't stop you from shipping it by temporarily placing it inside a box that meets FedEx's requirements.

Unfortunately, Objective-C doesn't provide for the automatic boxing and unboxing of primitive data types. Therefore, to store an integer or other primitive value in an array or dictionary, you must perform the boxing and unboxing yourself. This is the purpose of the `NSNumber` class.

4.3.1 The `NSNumber` class

`NSNumber` is an Objective-C class that can be used to wrap a value of a primitive data type such as `int`, `char`, or `BOOL` into an object and then allow that value to be extracted at a later stage. It's most useful for allowing you to place values of primitive data types into data structures such as `NSArray` or `NSDictionary` that can only store objects.

Manually boxing a primitive value into an `NSNumber` instance is fairly straightforward: you call one of `NSNumber`'s factory methods, such as `numberWithInt:`:

```
NSNumber *myNumber = [NSNumber numberWithInt:5];
```

There are similar factory messages called `numberWithFloat:`, `numberWithBool:`, and so on, to allow boxing of other common primitive data types.

Now that the integer value is boxed inside an `NSNumber` (which is an object), you can store it in your array, as you originally intended:

```
[myArray addObject:myNumber];
```

Boxing the integer inside an `NSNumber` also means that when you go to fetch the value from the array, you need to perform the reverse operation to extract the primitive value from the `NSNumber` instance. You can do this with a code snippet similar to the following that uses `NSNumber`'s `intValue` message:

```
NSNumber *myNumber = [myArray objectAtIndex:0];  
int i = [myNumber intValue];
```

The `NSNumber` class has various other methods that conform to the naming convention `xxxValue`, where `xxx` is replaced with the name of a primitive data type. It's not an error to box a value via `numberWithInt:` and then retrieve it via a method such as `floatValue`. Although you stored an integer and fetched a float, this is acceptable. In this scenario, the `NSNumber` class performs a typecast operation similar to those discussed in chapter 2 to convert the value into your desired data type.

A little bit of fancy footwork was required, but in the end, boxing and unboxing primitive values so you can use them as if they were objects wasn't too bad. What happens, though, if you want to store one or more `RentalProperty` structures in an `NSArray`? These structures are also not objects, but it's doubtful that the `NSNumber` class has a `numberWithRentalPropertyDetail` method available for you to box them. The answer to this conundrum is another closely related class called `NSNumber`.

4.3.2 *The NSValue class*

NSNumber is a special subclass of NSValue. While NSNumber provides a convenient and clean interface for boxing and unboxing numeric-based primitive types, NSValue allows you to box and unbox any C-style value at the expense of having a slightly more complex interface to program against.

To box the value of an arbitrary C structure or union, you can take advantage of NSValue's `valueWithBytes:objCType:` message. For example, the following code sample boxes a RentalProperty structure similar to those created in chapter 3:

```
RentalProperty myRentalProperty =
    {270.0f, @"13 Adamson Crescent", TownHouse};

NSValue *value = [NSValue valueWithBytes:&myRentalProperty
                    objCType:@encode(RentalProperty)];
```

This code snippet creates a copy of the rental property structure and places it inside the NSValue object created by the call to `valueWithBytes:objCType:`. The `valueWithBytes` argument is the address of the value you want to store in the NSValue instance, while `objCType` and the strange `@encode` statement allow you to tell NSValue what kind of data you want it to store.

The process of unboxing an NSValue is different but still relatively straightforward. Rather than returning the unboxed value directly, the `getValue` message expects you to pass in a pointer to a variable it should populate with the value it contains, as demonstrated here:

```
RentalProperty myRentalProperty;
[value getValue:&myRentalProperty];
```

On return, `getValue:` will have populated the `myRentalProperty` variable with the value you previously stored. You might be asking yourself, if NSNumber and NSValue allow you to get around the restriction of only being able to store objects in an array or dictionary, will a similar technique allow you to store the equivalent of a nil value? The answer is, yes.

4.3.3 *nil vs. NULL vs. NSNull*

In this chapter you learned that nil can't be stored in an array or dictionary and it's used to indicate the absence of a value or entry. But what exactly is nil?

With a standard C-style pointer, the special value NULL indicates the absence of a value. nil represents the very same concept, except applied to a variable that's designed to point to an Objective-C object.

Because nil represents the absence of a value, it can't be stored in an array or dictionary. But if you really want to store an empty or nil value, you can use a special class called NSNull. You can conceptualize NSNull as being another kind of boxing, similar to NSNumber or NSValue but specialized for storing nil.

To store a nil value in an array or dictionary, you can use the null factory method located on the `NSNull` class:

```
NSNull myValue = [NSNull null];  
[myArray addObject:myValue];
```

An instance of the `NSNull` class is an object, so it can be stored in an `NSArray` or `NSDictionary` instance or used anywhere else an object is expected. Because an `NSNull` instance represents the absence of a value, the class doesn't provide any way in which to extract a value, so when it comes time to pull your `NSNull` value out of an array, you can compare it against itself:

```
id value = [myArray objectAtIndex:5];  
if (value == [NSNull null]) {  
    NSLog(@"The 6th element within the array was empty");  
} else {  
    NSLog(@"The 6th element within the array had the value %@", value);  
}
```

Unlike the problems you had comparing `NSString` instances with the `==` operator, this technique works for `NSNull` instances. The statement `[NSNull null]` doesn't create a new object each time it's called. Instead, `[NSNull null]` always returns the memory location of a sole instance of the `NSNull` class. This means that two variables that are storing a pointer to an `NSNull` value will have an identical address, and hence the `==` operator will consider them the same. This is an example of the Singleton design pattern.

This completes our coverage of the basics of Foundation Kit's core classes and the data structure implementations they provide. With your newfound knowledge, you can also resolve the issue with the Rental Manager application highlighted at the end of chapter 3.

4.4 Making the Rental Manager application data driven

One outstanding problem with the Rental Manager application is that the mapping of city names to geographical locations is hardcoded, when what you ideally want is for it to be data driven and easily updateable without the need to recompile the entire application.

These concepts nicely match the functionality provided by an `NSDictionary` object (with the key being the city name and the value being its matching location type) that obtains its initial contents from a plist file.

Let's add a new plist file called `CityMappings.plist` to your project. As you may expect, the New File dialog (located in `File > New > New File...` and shown in figure 4.1) has a suitable template to get you started. Once the new file is added to your project, Xcode presents a graphical plist file editor that quickly allows you to edit its contents without worrying about balancing XML open and closing brackets, and so on. Using this editor, populate your new plist file with the contents shown in figure 4.2.

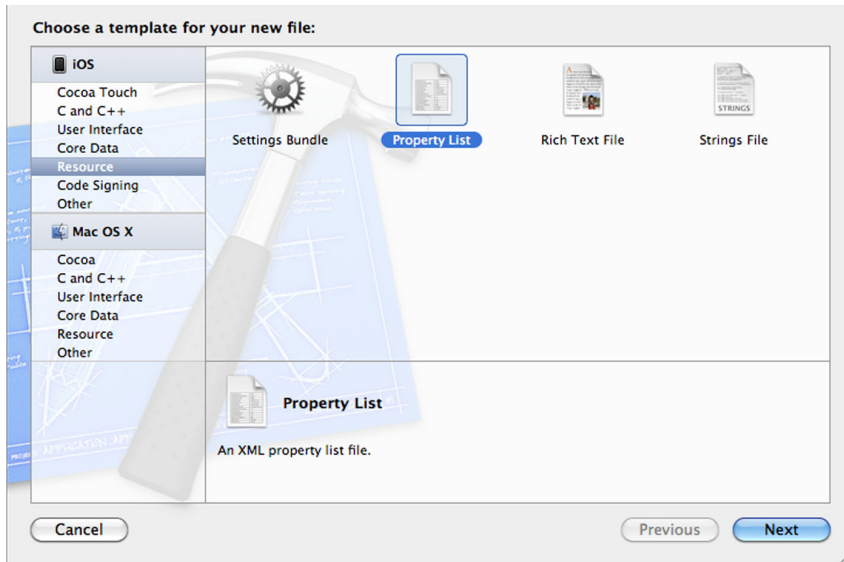


Figure 4.1 Adding a new Property List file to your project via the New File dialog. Watch out! The Property List file template is located in the Resource section, not in Cocoa Touch with the rest of the iOS-related templates.

With the easily editable plist file populated and added to your project, you can now make the few minor code changes required to make use of it.

The first change is to update the definition of the `RootViewController` class available in the `RootViewController.h` file to provide a place to store an `NSDictionary` containing the city-to-location mappings. You should end up with something like the following:

```
@interface RootViewController : UITableViewController {
    NSDictionary *cityMappings;
}
```

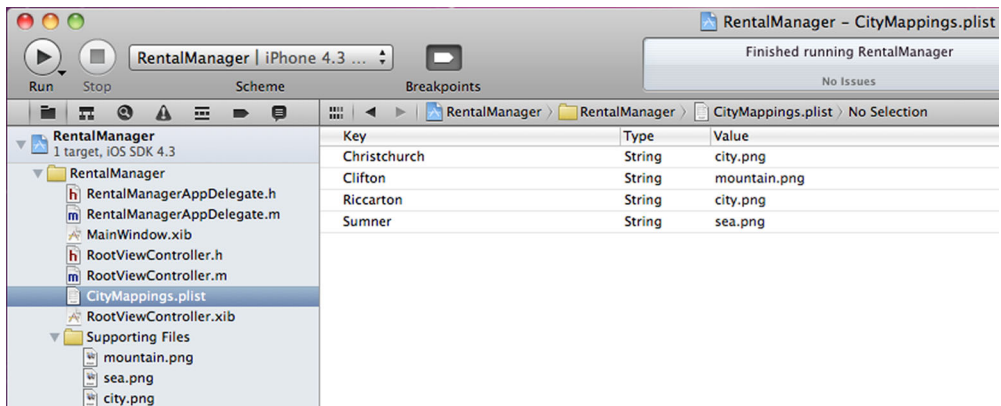


Figure 4.2 The graphical plist file editor available in Xcode whenever a plist file is selected

With the storage of the dictionary taken care of, you can now switch to the `RootViewController.m` file and make a couple of additional changes that'll create a new dictionary object and initialize it with the contents of the `CityMappings.plist` file.

Open your copy of `RootViewController.m` and replace the existing versions of the `viewDidLoad` and `dealloc` methods with those in the following listing.

Listing 4.3 Creating and destroying the city-to-geolocation mapping dictionary

```
- (void)viewDidLoad {
    [super viewDidLoad];

    NSString *path = [[NSBundle mainBundle]
                     pathForResource:@"CityMappings"
                     ofType:@"plist"];
    cityMappings = [[NSDictionary alloc] initWithContentsOfFile:path];
}

- (void)dealloc {
    [cityMappings release];
    [super dealloc];
}
```

The first additional line of source code determines the full path to the `CityMappings.plist` file in the application bundle. You could conceivably also replace this line with one that fetches the plist file from an internet site (as discussed previously in this chapter), allowing the mappings to be truly dynamic and updated almost instantaneously.

Once the location of the plist file is determined, you use `NSDictionary`'s `initWithContentsOfFile:` method to create a new immutable dictionary with the required city-to-geographical-location mappings.

Because you allocated memory, you must take care to return it to the operating system when you're done using it. This is the purpose of the `release` message seen in the `dealloc` method.

The only remaining modification required to get the Rental Manager application to use the `cityMapping` dictionary is to replace the `if` statement in the `tableView:cellForRowAtIndexPath:` method that loads a suitable image with the following two lines:

```
NSString *imageName = [cityMappings objectForKey:city];
cell.imageView.image = [UIImage imageNamed:imageName];
```

These two lines use `NSDictionary`'s `objectForKey:` message to look up a city name (the key) and find the matching image filename, which is then loaded and displayed to the user in the second line of source code.

Rebuild the application and run it in the debugger. If everything goes correctly, you should see no visible change from the previous version. But with the modified structure, no matter how many city-to-filename mappings are appended to the plist file, the previous two lines of source code will happily perform the task with no further attention or maintenance required. Score one for developer productivity!

4.5 Summary

Object-oriented programming, as implemented by Objective-C, has many advantages to the application developer over procedural-based languages such as C. The ability to combine data with logic means that you can encapsulate common functionality and avoid developing the same algorithms for enumeration, searching, and other actions in each application you develop. This enables you to concentrate on differentiating your product and to avoid common types of bugs. Nothing comes for free, however, and you've seen a number of limitations where the procedural world meets the object-oriented, such as the challenges of testing for object equality or storing values of primitive data types such as `int` in an object.

By using the reusable classes provided by Foundation Kit, you were quickly able to modify the Rental Manager application from being hardcoded to being a dynamic and rather easily modified application. Rather than requiring code changes to alter the city-to-image mappings, the updated version, which uses an `NSDictionary`, can easily be updated from a plist, a file on the internet, or any other source the developer's imagination dreams up.

In chapter 5, we cover some of the more technical aspects of Objective-C by learning how you can define and implement your own classes. You do this by turning the `RentalProperty` structure into an Objective-C class, so your sample application is truly object oriented. This is the last major C-style holdover left in your Rental Manager application.

Part 2

Building your own objects

If you're developing an Objective-C-based application, it won't be long before you're faced with the challenge of developing your own custom classes and objects. The classes provided by Foundation and UIKit can only take you so far. This part of the book begins with an explanation of how to create a class, and then expands into how to extend classes to specialize them, or adapt them for slightly different needs.

You'll also delve deeper into the process of message sending, and how the dynamic nature of Objective-C allows for classes to behave in dynamic ways. With custom objects, memory allocation and management is always an important topic to master, so we round out with a discussion of proper memory management techniques, and the five simple rules of object ownership.

5

Creating classes

This chapter covers

- Creating custom Objective-C classes
- Adding instance variables
- Adding class and instance methods
- Adding properties
- Implementing `init` and `dealloc`

The real power of object-oriented programming comes into play when you start to create your own custom objects that perform application-specific tasks, and this is what we explore in this chapter. By creating a custom class, you can encapsulate a common set of functionality into a reusable component that you can use repeatedly throughout an application (or even a set of applications).

To put the concepts into context, we concentrate on updating the Rental Manager application to use object-oriented design principles, replacing the `RentalProperty` structure with a class that provides equivalent functionality. The new class will store and maintain the details of a single rental property, which means you must specify what data each object should keep track of and add instance and class methods to allow safe and controlled access to that data.

We also cover the concept of properties and how they make calling common methods that set or get the value of a field stored by an object easier and quicker. Let's start by defining the shell of the new class.

5.1 *Building custom classes*

Creating a new class in Objective-C is relatively straightforward and typically consists of three steps:

- 1 Specifying the data you want associated with each object. This data is commonly called *instance variables* (or *ivars* for short).
- 2 Specifying the behavior or actions that an object can be expected to perform. This is commonly called *message* or *method declarations*.
- 3 Specifying the logic that implements the behavior declared in step 2. This is commonly called *providing an implementation of the class*.

By convention, the source code for a class tends to be spread over two files. A header file (*.h) provides a declaration of the instance variables and expected behavior, and a matching implementation, or method file (*.m), contains the source code to implement that functionality. You start by adding a header and method implementation file to the Rental Manager project to store the source code for a class called `CTRentalProperty`.

5.1.1 *Adding a new class to the project*

The easiest way to create a new class in Xcode is to select the New File menu option (Cmd-N). The New File dialog is displayed in figure 5.1.

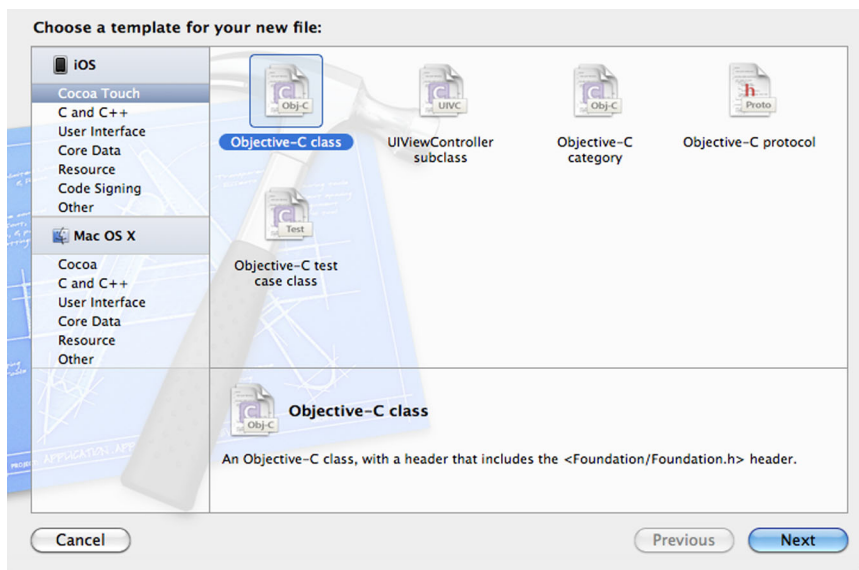


Figure 5.1 The New File dialog in Xcode enables you to add new files to your project. Selecting a different template allows you to alter the type of file generated and the default contents it'll contain.

The most suitable template for your purposes is Objective-C Class, which creates the basic structure of a class using Objective-C features without being designed for a specific purpose. This template is found in the Cocoa Touch Class subsection. Once you click Next, a pop-up menu is displayed that allows you to select the superclass of your new object (the default `NSObject` is the best option for you). Clicking Next brings up the last part of the New File dialog.

In this step you can specify the name and location of the files that are about to be created. Specify the name `CTRentalProperty.m` in the Save As box.

Clicking Save creates the method (*.m) and matching header (*.h) files and returns you to the main Xcode window with `CTRentalProperty.m` open in the editor.

Exploring the contents of the two newly created files, you'll notice that they aren't completely blank: the template has added a number of special directives such as `@interface` and `@implementation` to them. Let's look at how to create a custom class by opening the `CTRentalProperty.h` file and investigating the code it contains, starting with the `@interface` directive.

5.2 Declaring the interface of a class

The interface of a class defines the “public face” of a class—what the class looks like and how it can be interacted with. The interface declaration describes the kind of data the class stores and the various messages you can send it to perform tasks on your behalf. It's designed to provide enough details for other sections of an application to make use of the class without providing any details on how the class physically achieves its stated capabilities.

The declaration of a new class is signaled by the `@interface` compiler directive. Everything from this point on, until a matching `@end` directive, makes up the interface of the class. You may think the use of a directive named `@class` would make more sense. One way to remember this apparent discrepancy is to realize that your interface must eventually be matched with a suitable implementation, and indeed there is a matching `@implementation` directive that we cover later in this chapter.

The general form of a class definition is as follows:

```
@interface ClassName : SuperClass {
    ... instance variable declarations ...
}

... method declarations ...

@end
```

As you can see, immediately after the `@interface` directive is the name of the new class, followed by a colon and the name of the class it inherits from (its base class, or superclass).

After you specify the name of the class and outline its position in the class hierarchy, the body of the class interface declaration is split into two main sections, roughly divided by a pair of curly braces. The first section declares the list of instance variables

Sometimes subtle differences have the most profound effect

In Objective-C code it's common to see the superclass explicitly declared as `NSObject`. If you're a Java or C# developer, this may seem redundant, because in those languages omitting the superclass causes the compiler to automatically inherit your class from a predefined superclass (`java.lang.Object` or `System.Object`, respectively). Objective-C is similar to these languages in that it supports only single inheritance but, unlike those languages, it allows for multiple root classes. Not every class must ultimately inherit from `NSObject`.

If you don't explicitly provide a superclass, Objective-C declares your class as a new root class. In other words, the class will sit beside `NSObject` in the class hierarchy rather than below it. While this is possible, it generally isn't desirable because it means your new class won't inherit any of the standard functionality typically associated with an Objective-C object. Helpful methods such as `isEqual:` and memory management such as `alloc`, `init`, and `dealloc` are all implemented by `NSObject`.

the class associates with each object created, while the second section declares any methods and properties the class wishes to allow clients to access.

Before discussing how objects can provide logic to manipulate or work with the instance data they store, let's look at how you store data in a class.

5.2.1 Instance variables (*ivars*)

One of the first steps in defining a new class is to determine the type of data it needs to store. For the `CTRentalProperty` class, you want to store data similar to that which you previously stored in the `RentalProperty` structure. At a minimum, you'd like to store the following data for each rental property:

- Rental price per week (`float`)
- Address of property (`NSString *`)
- Property type (`enum`)

In the context of a class, such fields are called *instance variables* (or *ivars*, for short). Instance variables are specified inside a set of curly braces in a class's `@interface` declaration and appear just like ordinary variable declarations: you provide the data type and name of each variable and terminate each statement with a semicolon. For example, you could declare the instance variables of the `CTRentalProperty` class as follows:

```
float rentalPrice;
NSString *address;
PropertyType propertyType;
```

Each time you create a new object of the `CTRentalProperty` class, a unique set of these variables will be allocated to store the details of this particular instance. Therefore, if you create multiple `CTRentalProperty` instances to represent a group of rental properties, each will be able to store its own unique `rentalPrice`, `address`, and `propertyType` values.

Although declaring instance variables looks similar to declaring normal variables, there's a little more to their syntax than what we've discussed. Just as an employee may not want her boss to know she's searching for a new job, an object may want to restrict access to some of its internal data. The declaration of an instance variable can be preceded by an optional directive such as `@private` to alter who can access (or alter) its current value. Table 5.1 outlines the available directives. Once a visibility directive is specified, it applies to all additional instance variables declared until another accessibility directive is found. By default, instance variables are `@protected`.

Table 5.1 Standard visibility directives available for instance variables in Objective-C

Visibility level	Description
<code>@private</code>	Accessible only by the class that declared it
<code>@protected</code>	Accessible only by the class that declared it or by any subclass
<code>@public</code>	Accessible from anywhere
<code>@package</code>	Accessible from anywhere in the current code image (the application or static library in which the class is compiled)

To see why the concept of instance variable visibility is important, imagine that you had access to a class that represented a loan in a banking system. The bank wouldn't be happy if anyone with access to a loan object could reach inside and change the status instance variable from "rejected" to "accepted" or change the interest rate to -10% so the bank would pay the loan holder for the privilege of having a loan! Access and modification of this data would need to be tightly controlled, and this is exactly the intent of directives such as `@protected` and `@private`.

At the other end of the spectrum, you could mark an instance variable as `@public`, which means any piece of code can directly access the instance variable with no checks or balances to ensure it's done correctly. This practice is usually discouraged because it provides less control and flexibility over how instance variables are utilized. If direct access is discouraged, then how are you meant to access or change their values? The answer leads nicely onto the subject of how to declare methods in a class.

5.2.2 Method declarations

Because allowing direct access to instance variables is discouraged, you must find another way to allow users to query and alter their values. This alternative technique should allow you to tightly control access and optionally provide other services such as logging and validation. One solution is to provide a set of messages that the object can be sent in order to update or access the instance variables on your behalf. Because the code implementing the method is part of the class, it'll have access to the instance variables no matter their protection level.

Falling on the side of caution

Although it may be tempting to declare all instance variables as `@public`, in general, you should try to keep them as tightly restricted as possible. Once access to an instance variable is granted, it can be hard to remove or alter the purpose of the ivar as your application continues to grow and develop. This is especially true if your class is designed to be part of a widely used framework or support class.

With the default visibility of `@protected`, a subclass can access any instance variable declared by its superclass. Once it does, however, an updated version of the superclass can't easily remove that variable or alter its purpose. If it does, the subclass might break, as it now relies on a nonexistent instance variable or at least is potentially using it in an incorrect manner.

By making the instance variable `@private` and providing methods to indirectly access its value, you can isolate the storage of the value from its behavior. If an update to the superclass means you want to remove the instance variable and instead obtain its value via a calculation, you can do this in the accessor methods without affecting any users of the class. As the saying goes, most things in computer science can be solved by adding another layer of indirection.

For the `CTRentalProperty` class, you want clients to be able to

- Get or set the rental price to an absolute dollar amount
- Get or set the address of the property
- Get or set the property type

You can easily come up with additional helpful messages that improve the usability of the class, such as

- Increase the rental price per week by a fixed percentage
- Decrease the rental price per week by a fixed percentage

The messages understood by a class are declared in the `@implementation` section after the curly braces enclosing instance variables and before the `@end` directive. This section is like a list of steps in a recipe. It gives you a general idea of how the final product is put together, but it doesn't go into great detail about how each step is achieved.

The simplest form of method declaration is one that expects no parameters and returns a single value. For example, the following declares a method called `rentalPrice`, which returns a floating-point number:

```
- (float)rentalPrice;
```

The data type of the value returned by the method is enclosed in parentheses and can be any valid Objective-C type, such as those discussed in chapter 2. A special data type, `void`, can also be used to indicate that no value is returned.

Now that you have a method declared that allows you to query the current rental price, it's only natural to want to complement it with a method to allow you to change

By sticking with conventions, you won't rock the boat

In Objective-C it's traditional to name methods that return the value of instance variables the same as the instance variables themselves. This is unlike languages such as Java in which such methods would commonly use a "get" prefix, as in `getRentalPrice`.

You can name your methods using any convention you like, but by using the time-honored Objective-C conventions, you'll have a smoother voyage as you develop your application. Many features of Cocoa Touch, such as Core Data, Key-Value Coding, and Key-Value Observing, rely in part on language conventions—or at least work nicer out of the box if these conventions are respected. If you use other conventions, you may discover that you have additional work to do before being able to fully use all features of Objective-C or Cocoa Touch.

the rental price. To do so, you need to specify that the method expects a parameter, or argument:

```
- (void)setRentalPrice:(float)newPrice;
```

In a method declaration, a parameter is introduced by the colon (:) character, which is followed by the parameter's data type and name. In this case you've declared a method called `setRentalPrice:`, which has a single parameter called `newPrice` of type `float`.

It's possible to declare a method that expects multiple parameters. For example, you could declare a method to decrease the rental price by a fixed percentage, ensuring it doesn't drop below a predetermined minimum value, by declaring a method as follows:

```
- (void)decreaseRentalByPercent:(float)percentage withMinimum:(float)min;
```

This code declares a method named `decreaseRentalByPercent:withMinimum:` that accepts two parameters (`percentage` and `min`). Notice how the position of each colon character in the name indicates where a parameter is expected and that the method name is interspersed between the parameter declarations rather than being a single identifier at the start of the declaration.

It's important to realize that in a method signature, the colons aren't optional. As an example, the identifiers `rentalPrice` and `rentalPrice:` identify different methods. The first accepts no parameters, while the second expects a single parameter (as indicated by the colon). It's possible (although rather confusing) for a class to declare both methods.

It isn't essential to provide a part of the method name before each parameter. For example, the following declaration is equally valid even though no part of the method name is provided before the last argument named `min`:

```
- (void)decreaseRentalByPercent:(float)percentage :(float)min;
```

This alternative method is named `decreaseRentalByPercent::` and is generally considered bad form because the name of the method is less self-documenting than

Named parameters aren't the same as optional parameters

At first glance, it may appear that methods in Objective-C support named parameters. For example, a method such as `decreaseRentalByPercent:withMinimum:` would be called as follows:

```
[myProperty decreaseRentalByPercent:25 withMinimum:150];
```

where `decreaseRentalByPercent:` and `withMinimum:` could be considered names for the two parameters. This appearance is only skin deep, however. Unlike languages that support true named parameters, Objective-C requires when you call a method that you specify the parameters in the exact order they're declared in the method declaration. Likewise, it isn't possible to omit parameters in order to have default values assumed.

Naming parameters leads to more descriptive code. For example, to call a C function called `addWidget`, you may use the following statement:

```
addWidget(myOrder, 92, NO);
```

From the call alone, it's impossible to determine what this achieves. Comparing it to the following Objective-C–based code, the benefit of naming parameters becomes more obvious:

```
[myOrder addWidgetWithPartNumber:92 andRestockWarehouse:NO];
```

`decreaseRentalByPercent:withMinimum:`, where the purpose of the last argument is more obvious without needing to refer to additional documentation.

It's important to note that a method signature, such as `decreaseRentalByPercent:withMinimum:`, doesn't contain any details about the data types of expected arguments. This fact means that method overloading isn't possible in an Objective-C class.

CLASS VS. INSTANCE METHODS

You may have noticed that every method declaration so far has had a leading `-` character. This symbol indicates that the method is an instance method. When you invoke the method, you must specify a particular object (or instance) of the class for the method to operate on, and as a result, the method has access to all of its instance variables.

Another type of method is the class method, also commonly called a *static* method. Class methods are identified by a `+` character instead of a `-` character. The obvious advantage of class methods is that you don't need to create an instance of the class in order to be able to call them. But this can also be a disadvantage because it means class methods are unable to access any information (such as instance variables) specific to a particular object or instance.

Class methods are commonly used to provide simpler ways to create new objects (for example, `NSString`'s `stringWithFormat:`) or to provide access to shared data (for example, `UIColor`'s `greenColor`). These methods don't need any instance-specific data in order to perform their intended task, so they're ideal class method candidates.

As an example, you could define a method that takes a `PropertyType` enumeration value and returns a string containing a brief description of that value:

```
+ (NSString *)descriptionForPropertyType:(PropertyType)propertyType;
```

To invoke such a method, slightly different syntax is required. Because an instance or object isn't required to execute the method against, you instead specify the name of the class as the target, or receiver, of the message:

```
NSString *description = [CTRentalProperty
                        descriptionForPropertyType:TownHouse];
```

Well, that's enough of the abstract theory—let's put the concepts you've just learned into practice by completing the `CTRentalProperty.h` header file stubbed out by the Objective-C class file template.

5.2.3 *Fleshing out the header file for the CTRentalProperty class*

If you've been following along, you should have `CTRentalProperty.h` open in Xcode. Replace the current contents with that of the following listing.

Listing 5.1 Defining the interface of the `CTRentalProperty` class

```
#import <Foundation/Foundation.h>

typedef enum PropertyType {
    TownHouse, Unit, Mansion
} PropertyType;

@interface CTRentalProperty : NSObject {
    float rentalPrice;
    NSString *address;
    PropertyType propertyType;
}

- (void)increaseRentalByPercent:(float)percent
    withMaximum:(float)max;
- (void)decreaseRentalByPercent:(float)percent
    withMinimum:(float)min;

- (void)setRentalPrice:(float)newRentalPrice;
- (float)rentalPrice;

- (void)setAddress:(NSString *)newAddress;
- (NSString *)address;

- (void)setPropertyType:(PropertyType)newPropertyType;
- (PropertyType)propertyType;

@end
```

1 **Declare new class**

2 **Protected instance variables**

3 **Two methods**

4 **Setter and getter methods**

The contents of `CTRentalProperty.h` declare ❶ a new class called `CTRentalProperty` that derives (or inherits) from the `NSObject` base class. The class contains a number of instance variables ❷ which are all protected due to the absence of any directives such as `@public` or `@private`. The class is then completed by declaring a set of setter and getter methods ❸ to allow you to provide controlled access to the instance variables.

To demonstrate that classes can contain methods other than simple getters and setters, the `CTRentalProperty` class also declares ❸ two methods with the names `increaseRentalByPercent:withMaximum:` and `decreaseRentalByPercent:with-Minimum:`. These methods act as special setters that update the `rentalPrice` instance variable by performing a simple calculation based on the provided parameters.

This class interface declaration is enough for anybody to go off and develop code that makes use of the `CTRentalProperty` class. It has outlined to the compiler, and to any interested users of the class, what kind of data you store and what kind of behavior they can expect from you. With the classes interface specified, it's time to provide the compiler with the logic that implements the class.

5.3 *Providing an implementation for a class*

The `@interface` section of a class only declares what an object will look like. It doesn't provide details on how methods are implemented. Instead, this source code is typically contained in a separate implementation (or method) file that commonly has the file extension `*.m`.

Similar to the `@interface` section, the implementation of a class is started by an `@implementation` directive followed by the class name and ending with an `@end` directive:

```
@implementation CTRentalProperty
    ... method implementations ...
@end
```

The class name is required after the `@implementation` directive because it's possible for a single `*.m` file to contain the implementations of multiple classes. With the location of a particular class's implementation determined, let's look at how you specify the behavior of a particular method.

5.3.1 *Defining method implementations*

You define the logic and behavior of a method by repeating its declaration. But instead of ending the declaration with a semicolon, you use a set of curly braces and provide the required logic that should be executed whenever the method is invoked. The following is one possible implementation of the `setRentalPrice:` method:

```
- (void)setRentalPrice:(float)newRentalPrice {
    NSLog(@"TODO: Change the rental price to %f", newRentalPrice);
}
```

This implementation, however, simply logs a to-do message requesting you provide a more practical implementation. To flesh out the implementation, you must be able to access the instance variables associated with the current object. Luckily, this is easy to do.

5.3.2 *Accessing instance variables*

When the body of an instance method is declared, the method automatically has access to any instance variables associated with the current object. You can refer to

them by their name. The following is a possible implementation of `CTRentalProperty`'s `setRentalPrice:` method:

```
- (void)setRentalPrice:(float)newRentalPrice {
    rentalPrice = newRentalPrice;
}
```

Notice how the assignment statement can assign a new value to the `rentalPrice` instance variable without any fuss. You may wonder how this statement knows which object to update in the case that multiple `CTRentalProperty` objects have been created. You don't pass in any reference to an object of interest. The answer is that, behind the scenes, every instance method is passed two additional hidden parameters called `self` and `_cmd`. We leave discussion of `_cmd` for chapter 8, but `self` is the magic that enables the method to know which object it should work with. You could explicitly make this linkage more apparent by rewriting the `setRentalPrice:` method as follows:

```
- (void)setRentalPrice:(float)newRentalPrice {
    self->rentalPrice = newRentalPrice;
}
```

The `->` operator allows you to access the instance variables associated with the variable referenced on the left-hand side. Because the compiler can pass different objects into the method via the hidden `self` parameter, it can change which object the method works with.

Check out your debugger window

The next time you use the Xcode debugger and hit a break point, check out the Arguments section in the variables panel. You should be able to clearly see the `self` and `_cmd` parameters listed.

Congratulations! You're well on your way to becoming an Objective-C guru. Another slightly mysterious piece of the IDE suddenly makes a little more sense.

Knowing that `self` is a hidden parameter that always represents the current object is handy, but as demonstrated previously, it isn't typically necessary to use it to access instance variables. There is, however, another practical use of `self` that's unavoidable.

5.3.3 Sending messages to self

When implementing an instance method, you may wish for it to call on the services of other methods defined by the class. But to send a message, you must first have a reference to the object you want to send it to. This reference is usually provided in the form of a variable. When you want to refer to the current object, whatever it may be, the hidden `self` parameter neatly fills the role, as follows:

```
- (void)handleComplaint {
    NSLog(@"Send out formal complaint letter");
    numberOfComplaints = numberOfComplaints + 1;
}
```

```

    if (numberOfComplaints > 3) {
        [self increaseRentalByPercent:15 withMaximum:400];
    }
}

```

The `handleComplaint` method (something you might like to add as the Rental Property application continues to grow) increases the `numberOfComplaints` instance variable by 1. If a rental property has had more than three formal complaints, the method also invokes the `increaseRentalByPercent:withMaximum:` method on the same object, as indicated by the use of `self` as the target for the message.

5.3.4 *Fleshing out the method file for the CTrentalProperty class*

Using your newfound knowledge of how to implement methods, you're ready to complete the first revision of your `CTrentalProperty` class. Replace the contents of `CTrentalProperty.m` with the code in the following listing.

Listing 5.2 Providing an initial implementation of the `CTrentalProperty` class

```

#import "CTrentalProperty.h"

@implementation CTrentalProperty

- (void)increaseRentalByPercent:(float)percent
  withMaximum:(float)max {

    rentalPrice = rentalPrice * (100 + percent) / 100;
    rentalPrice = fmin(rentalPrice, max);
}

- (void)decreaseRentalByPercent:(float)percent
  withMinimum:(float)min {

    rentalPrice = rentalPrice * (100 - percent) / 100;
    rentalPrice = fmax(rentalPrice, min);
}

- (void)setRentalPrice:(float)newRentalPrice {
    rentalPrice = newRentalPrice;
}

- (float)rentalPrice {
    return rentalPrice;
}

- (void)setAddress:(NSString *)newAddress {
    [address autorelease];
    address = [newAddress copy];
}

- (NSString *)address {
    return address;
}

- (void)setPropertyType:(PropertyType)newPropertyType {
    propertyType = newPropertyType;
}

```

1 Update instance variable

2 Free previous address

3 Make a copy

```
- (PropertyType)propertyType {
    return propertyType;
}
@end
```

Most of the methods in `CTRentalProperty.h` are relatively straightforward (for example, `setRentalPrice:` ❶ updates the value of an instance variable). One method that deserves extra attention is `setAddress:` because it must deal with memory management issues. Rather than storing the new address in the instance variable directly, it makes a copy ❸ (taking care to free any previous address first ❷). This is done so code similar to the following will work as expected:

```
NSMutableString *anAddress =
    [NSMutableString stringWithString:@"13 Adamson Crescent"];
myRental.address = anAddress;
[anAddress replaceOccurrencesOfString:@"Crescent"
    withString:@"Street"
    options:NSCaseInsensitiveSearch
    range:NSMakeRange(0, [anAddress length])];
NSLog(@"The address is %@", myRental.address);
```

Most developers would expect this code snippet to indicate the address is 13 Adamson Crescent even though the string variable is updated to 13 Adamson Street. Making a copy of the string provided to `setAddress:` ensures this works as expected. Writing code to handle conditions like these can get arduous and error prone fairly quickly. It's just the kind of thing you let slip toward the end of a late-night, caffeine-aided coding session. Luckily, Objective-C provides special features to make writing such code easier.

5.4 Declared properties

In listings 5.1 and 5.2, much of the code is related to the declaration and implementation of getter and setter methods to provide safe access to associated instance variables and provide extensibility points for future logging or validation. Objective-C can automatically implement most of the code required to implement these methods using a feature called *Declared Properties*.

A property allows you to describe the intent of a setter and getter method pair while leaving it up to the compiler to provide the actual implementation. This is more convenient to write and ensures consistent code quality.

5.4.1 @property syntax

The first step in using a property is to declare its presence in the `@interface` section of a class. To do this, you use a special `@property` directive. As an example, you can declare a property called `rentalPrice` of type `float` as follows:

```
@property float rentalPrice;
```

You can consider this property declaration as being equivalent to manually declaring the following two methods:

```
- (float)rentalPrice;
- (void)setRentalPrice:(float)newRentalPrice;
```

By default, the `@property` statement declares a getter method with the specified name and a matching setter by prepending the prefix “set.”

As an example of using properties, update the contents of `CTRentalProperty.h` with the following listing. Notice you removed the manual setter and getter method declarations and replaced them with equivalent `@property` declarations.

Listing 5.3 Simplifying `CTRentalProperty.h` by using declared properties

```
#import <Foundation/Foundation.h>

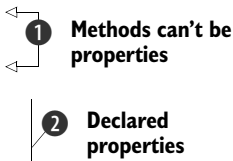
typedef enum PropertyType {
    TownHouse, Unit, Mansion
} PropertyType;

@interface CTRentalProperty : NSObject {
    float rentalPrice;
    NSString *address;
    PropertyType propertyType;
}

- (void)increaseRentalByPercent:(float)percent
    withMaximum:(float)max;
- (void)decreaseRentalByPercent:(float)percent
    withMinimum:(float)min;

@property(nonatomic) float rentalPrice;
@property(nonatomic, copy) NSString *address;
@property(nonatomic) PropertyType propertyType;

@end
```



1 Methods can't be properties

2 Declared properties

By using declared properties **2**, you've replaced six lines of source code with three. Notice, however, that methods such as `increaseRentalByPercent:withMaximum:` **1** and `decreaseRentalByPercent:withMaximum:` couldn't be specified as properties because they don't perform simple get- and set-style operations. Each property declaration can contain an optional list of attributes enclosed in parentheses (such as the `nonatomic` or `copy` attributes seen in listing 5.3). These attributes provide information about how the getter and setter methods generated by Objective-C should behave. Table 5.2 summarizes the available property attributes and their purposes.

Table 5.2 Common attributes that can affect the behavior of declared properties

Category	Example attributes	Description
Method naming	setter, getter	Allow the developer to override the name of the generated methods.
Writeability	readonly, readwrite	Allow the developer to specify that a property is read-only (it doesn't have a setter method).
Setter semantics	assign, retain, copy	Allow the developer to control how memory management of property values is handled.
Thread safety	nonatomic	Properties are safe to use in multithreaded code. This safety can be removed to gain a potential performance increase.

Let's briefly look at how to use the optional `@property` attributes listed in table 5.2 to alter the behavior of the declared properties.

METHOD NAMING

By default for a property named `foo`, the Objective-C compiler produces a getter method named `foo` and a setter method named `setFoo:`. You can override these names by explicitly specifying an alternative name via the optional getter and setter attributes.

As an example, the following property declaration provides a getter called `isSelected` and a setter called `setSelected:`

```
@property (getter=isSelected) BOOL selected;
```

This attribute is commonly used with properties with a `BOOL` data type to rename their getter methods into the form `isXYZ`, another Objective-C naming convention.

WRITEABILITY

A property typically indicates the presence of both a getter and setter method. By adding the `readonly` attribute to a property declaration, you can ensure that users of the class can only query the current value of the property and can't change its value. For example, the following declaration specifies that the `age` property has only a getter method:

```
@property (readonly) int age;
```

The default behavior of each property having a getter and setter method can also explicitly be specified using the `readwrite` attribute. Because a `readonly` property has no setter, code in the class would have to modify the associated instance variable directly.

SETTER SEMANTICS

These attributes enable you to specify how a setter method will deal with memory management. Following are the three mutually exclusive options:

- *Assign*—The setter uses a simple assignment statement (the default option).
- *Retain*—The setter calls `retain` on the new value and `release` on the old one.
- *Copy*—A copy of the new value is made and retained.

Memory management can become a complex topic (look out for chapter 9). For now, it's enough to understand that `assign` indicates that no additional memory management behavior is required (the property assignment is treated as if it were a simple variable). The `copy` attribute requests that a “carbon-copy” duplicate of the value being provided be made and the copy stored, and the `retain` attribute does something in between.

THREAD SAFETY

By default, properties are atomic. This is fancy-speak for indicating that in a multi-threaded environment, Objective-C ensures the value obtained via a getter or set via a setter is always consistent and not corrupted by concurrent access by other threads.

This protection, however, doesn't come for free and may have a performance cost associated with it due to the time taken to acquire the locks needed to protect against

concurrent access. If your property is unlikely to be called from multiple threads, you can specify the `nonatomic` attribute to opt out of this protection:

```
@property (nonatomic) int age;
```

Most iPhone tutorials and sample source code specify properties with the `nonatomic` attribute because it's unlikely that properties in a view controller or view subclass will find themselves used in a multithreaded scenario. Hence, the use of `nonatomic` may cause a slight performance improvement, especially if the property is heavily utilized.

This concludes our look at how to declare properties in the `@implementation` section of a class. Although you updated the `CTRentalProperty` class to use properties, you'll notice you still have the original method implementations in `CTRentalProperty.m`. This is perfectly acceptable (in fact, now would be a good time to compile the project to make sure no typos have crept in), but you can improve on this situation by allowing the compiler to provide automatic implementations for you.

5.4.2 *Synthesizing property getters and setters*

Using the `@property` directive allows you to simplify the declaration of getter and setter methods, but Objective-C can go even further and automatically write the getter and setter method implementations for you. To do this, you use a directive called `@synthesize` in the class's `@implementation` section.

As an example, you can simplify listing 5.2, removing all the setter and getter methods you manually wrote and replacing them with an additional directive called `@synthesize`, as shown in the following listing.

Listing 5.4 Using synthesized properties to automatically generate setters and getters

```
#import "CTRentalProperty.h"

@implementation CTRentalProperty

@synthesize rentalPrice, address, propertyType;

- (void)increaseRentalByPercent:(float)percent
  withMaximum:(float)max {
    rentalPrice = rentalPrice * (100 + percent) / 100;
    rentalPrice = fmin(rentalPrice, max);
}

- (void)decreaseRentalByPercent:(float)percent
  withMinimum:(float)min {
    rentalPrice = rentalPrice * (100 - percent) / 100;
    rentalPrice = fmax(rentalPrice, min);
}

@end
```

1 Generate setter and getter methods

The `@synthesize` directive **1** requests that the Objective-C compiler automatically generate the getter and setter methods associated with the specified properties.

Although `@synthesize` is commonly used in tandem with the `@property` directive, its use is entirely optional. The `@synthesize` directive is intelligent and will generate getter or setter methods only if it can't find a suitable method already declared elsewhere in the `@implementation` section. This can be helpful if you'd like the compiler to generate most of your getter and setter methods, but you'd like to override one or two of them in order to perform special behavior, such as logging or input validation. An example of this functionality is demonstrated here:

```
@synthesize rentalPrice;

- (void)setRentalPrice:(float)newRentalPrice {
    NSLog(@"You changed the rental per week to %f", newRentalPrice);
    rentalPrice = newRentalPrice;
}
```

In this case, the `@synthesize` directive generates a getter method for the `rentalPrice` property but uses the explicitly specified `setRentalPrice:` setter.

By default, the `@synthesize` directive assumes that a property named `foo` will store its value in an instance variable also named `foo`. If for some reason this isn't desirable, you can override the name of the instance variable in the `@synthesize` statement. The following declaration, for example, synthesizes a `rentalPrice` property by using an instance variable called `rentalPerWeek`:

```
@synthesize rentalPrice = rentalPerWeek;
```

Use of the `@synthesize` directive results in the compiler generating, or synthesizing, the methods required for a property. Unlike manually written code, the generated code is well tested, thread safe, and efficient while handling the nuances of memory management without a second thought from the developer. Using properties not only reduces the amount of code you need to write but also ensures you avoid common opportunities to introduce bugs. You can have your cake and eat it too!

5.4.3 Dot syntax

With your current knowledge of Objective-C syntax to obtain the current address of a rental property, you must send the object an address message, as demonstrated here:

```
CTRentalProperty *property = ...some rental property...
NSString *address = [property address];
```

Likewise, you could update the address property by sending the `setAddress:` message:

```
CTRentalProperty *property = ...some rental property...
[property setAddress:@"45 Some Lane"];
```

Objective-C, however, provides an alternative syntax for use with properties. This syntax is based on the dot (`.`) operator and may be more comfortable for developers with a C or Java background:

```
CTRentalProperty *property = ...some rental property...
NSString *address = property.address;
NSLog(@"Old address is: %@", address);
```



```
property.address = @"45 Some Lane";
NSLog(@"New address is: %@", property.address);
```

This code sample is identical in behavior to the previous ones that used standard message-send syntax. The dot operator is pure syntactic sugar, and behind the scenes, assigning the value `xyz` to `property.address` is converted into an equivalent call to `[property setAddress:xyz]`. It's important to note, however, that unlike in C# or Java, the dot-style syntax can be used only with getter and setter messages associated with `@property`s; for any other method, the standard Objective-C message syntax is still required.

Danger, Will Robinson, here be sleeping dragons

With great power comes great responsibility and traps for the unwary. When using properties, you should be careful to clearly understand the difference between accessing a property and accessing an associated instance variable directly. Because you use the same identifier for the property and instance variable, the syntax for accessing both can be confusingly similar at times.

As an example, in the implementation of the `CTRentalProperty` class, the following statement updates the value of the `rentalPrice` instance variable,

```
rentalPrice = 175.0f;
```

while the following statement performs the same task by calling the `setRentalPrice:` setter method:

```
self.rentalPrice = 175.0f;
```

Although at first glance the two statements appear similar, the subtle change in source code has a significant impact on observed behavior.

If you directly access an instance variable, you bypass all the memory management, thread safety, and additional logic that may have been implemented in the setter and getter methods. No logic gets executed when the instance variable is read or altered.

As a general rule of thumb, if a property or accessor method exists, it's probably a good idea to prefer these methods over direct access to instance variables. This is especially true if you intend to use features such as Key-Value Observing (KVO). KVO uses dynamic features of Objective-C to effectively "hook" or spy on calls to setter methods to determine changes in an object's state, so direct access of an instance variable may be missed.

One advantage of the dot syntax over calling setter or getter methods directly is that the compiler can signal an error if it detects an assignment to a read-only property. On the other hand, due to the dynamic nature of Objective-C, if you call a nonexistent setter method, the compiler can only generate a warning that the class may not respond to this message. Apart from the warning, the application will successfully compile, and it's only when the application is run that you'll notice that it fails. We discuss the reason why the compiler can't complain about missing setter methods in

chapter 8. For now, let's move on and discuss how you can create new instances of the `CTRentalProperty` class.

5.5 Creating and destroying objects

One of the most common tasks to perform with classes is the creation of new objects. In Objective-C it takes two steps to create an object; in order, you must

- Allocate memory to store the new object.
- Initialize the newly allocated memory to appropriate values.

An object isn't fully functional until both steps are completed.

5.5.1 Creating and initializing objects

For objects that inherit from `NSObject`, memory for new objects is typically allocated by calling the class method `alloc` (short for *allocate*). As an example, you could create a new `CTRentalProperty` object by executing the following line of source code:

```
CTRentalProperty *newRental = [CTRentalProperty alloc];
```

This statement uses the `alloc` method to reserve enough memory to store all the instance variables associated with a `CTRentalProperty` object and assigns it to a variable named `newRental`. You don't need to write your own implementation of the `alloc` method because the default version inherited from `NSObject` is suitable. Most objects, however, do require additional initialization once the memory has been allocated. By convention, this initialization is achieved by calling a method named `init`:

```
CTRentalProperty *newRental = [CTRentalProperty alloc];  
[newRental init];
```

It's even possible to perform both of these steps in a single line by nesting the message sends as follows:

```
CTRentalProperty *newRental = [[CTRentalProperty alloc] init];
```

This works because the `init` method conveniently returns the object it's called upon (`self`) to support such usage.

What do uninitialized instance variables get set to?

You may wonder what value your instance variables will have if your initialization method doesn't explicitly initialize them. Unlike typical C-style memory allocation strategies, the memory returned by `alloc` has each instance variable initialized to the value zero (or its equivalent: `nil`, `NULL`, `NO`, `0.0`, and so on). This means you don't need to redundantly initialize variables to such values.

In fact, you should consider choosing meanings for Boolean variables such that `NO` is the appropriate state of object allocation. For example, `isConfigured` may be a better instance variable name than `needsConfiguration`.

It's important to understand the two-step initialization process used by Objective-C because there are a number of nuisances that can appear subtle but jump up and bite you when you least expect it.

BEWARE OF LURKING GOBLINS

The code snippet that called `alloc` and `init` on separate lines has a silent but potentially fatal flaw. Although it may not be a problem with most classes (`CTRentalProperty` included), it's possible for `init` to return an object different from the one created by `alloc`. If this occurs, the original object becomes invalid, and you can only use the object returned by `init`. In coding terms this means you should always store the return value of `init`; here is a bad example:

```
CTRentalProperty *newRental = [CTRentalProperty alloc];
[newRental init];
```

And here is a good example:

```
CTRentalProperty newRental = [CTRentalProperty alloc];
newRental = [newRental init];
```

Rather than worry about handling such scenarios, it's generally easier to perform both steps at once, as in `[[CTRentalProperty alloc] init]` and not give it a second thought.

You may wonder under what conditions `init` may return a different object from the one you allocated via `alloc`. For most classes this will never occur, but in special circumstances (implementing singletons, caches, or named instances, and so on), a class developer may decide to more tightly control when objects are created, preferring to return an existing object that's equivalent rather than initialize a new one, for example.

FAILURE IS A FACT OF LIFE

It isn't always possible for an initialization method to perform its intended task. For example, an `initWithURL:` method may initialize an object with data fetched from a website. If the URL provided is invalid or the iPhone is in flight mode, `initWithURL:` may not be able to complete its task. In such cases it's common for the `init` method to free the memory associated with the new object and return `nil`, indicating that the requested object couldn't be initialized.

If there's a chance that your initialization method may fail, you may like to explicitly check for this condition, as demonstrated here:

```
CTRentalProperty newRental = [[CTRentalProperty alloc] init];
if (newRental == nil)
    NSLog(@"Failed to create new rental property");
```

Perhaps you've picked up on the fact that initialization methods aren't always called `init` and that they can accept additional parameters. This is quite common, if not the norm, so let's take a look at the why and how behind this.

5.5.2 *init is pretty dumb*

An initialization method that accepts no parameters has limited practical use. You often need to provide additional details to the initialization method in order for it to

correctly configure the object. Typically, a class provides one or more specialized initialization methods. These methods are commonly named using the form `initWithXYZ:`, where `XYZ` is replaced with a description of any additional parameters required to properly initialize the object.

As an example, add the following method declaration to the `@interface` section of the `CTRentalProperty` class:

```
- (id)initWithAddress:(NSString *)newAddress
    rentalPrice:(float)newRentalPrice
    andType:(PropertyType)newPropertyType;
```

This instance method enables you to provide a new address, rental price, and property type for the object about to be initialized. The next step is to provide an implementation for the method in the `@implementation` section of the `CTRentalProperty` class. Do this by adding the following code.

Listing 5.5 Implementing a custom initialization method for `CTRentalProperty`

```
- (id)initWithAddress:(NSString *)newAddress
    rentalPrice:(float)newRentalPrice
    andType:(PropertyType)newPropertyType
{
    if ((self = [super init])) {
        self.address = newAddress;
        self.rentalPrice = newRentalPrice;
        self.propertyType = newPropertyType;
    }
    return self;
}
```

The main part of this method sets the various properties of the object to the new values specified by the parameters. There are, however, a couple of additional features in this listing that deserve extra attention.

The `if` statement performs a number of important steps in a single, condensed line of source code. Working from right to left, it first sends the `init` message to `super`. `super` is a keyword, similar to `self`, that enables you to send messages to the superclass. Before you initialize any of your own instance variables, it's important to provide your superclass a chance to initialize its own state.

The object returned by the superclass's `init` method is then assigned to `self` in case it has substituted your object for another. You then check this value to ensure it isn't `nil`, which would indicate the superclass has determined it can't successfully initialize the object. If you prefer, you could expand these steps into two separate statements, as follows:

```
self = [super init];
if (self != nil) {
    ...
}
```

The behavior of the application would be identical. The first form is a more condensed version of the second. Finally, the method returns the value of `self`, in part to enable the condensed `[[CTRentalProperty alloc] init]`-style creation syntax mentioned previously.

5.5.3 **Combining allocation and initialization**

Because it's common to allocate an object and then want to immediately initialize it, many classes provide convenience methods that combine the two steps into one. These class methods are typically named after the class that contains them. A good example is `NSString`'s `stringWithFormat:` method, which allocates and initializes a new string with the contents generated by the specified format string.

To allow users to easily create a new rental property object, add the following class method to the `CTRentalProperty` class:

```
+ (id)rentalPropertyOfType:(PropertyType)newPropertyType
    rentingFor:(float)newRentalPrice
    atAddress:(NSString *)newAddress;
```

This method draws close parallels to the initialization method (listing 5.5) you just added. The main difference, apart from a slightly different name, is that `rentalPropertyOfType:rentingFor:atAddress:` is a class method, whereas `initWithAddress:andPrice:andType:` is an instance method. Being a class method, `rentalPropertyOfType:rentingFor:atAddress:` allows you to invoke the method without first creating an object. Implement the method as shown in the following listing.

Listing 5.6 Merging allocation and initialization into one easy-to-call method

```
+ (id)rentalPropertyOfType:(PropertyType)newPropertyType
    rentingFor:(float)newRentalPrice
    atAddress:(NSString *)newAddress
{
    id newObject = [[CTRentalProperty alloc]
                    initWithAddress:newAddress
                    rentalPrice:newRentalPrice
                    andType:newPropertyType];
    return [newObject autorelease];
}
```

Notice that the implementation uses the existing `alloc` and `initWithAddress:rentingFor:andType:` methods and returns the newly initialized object to the caller. Another difference is that the object is also sent an `autorelease` message. This additional step deals with a memory-related convention, which we cover in depth in chapter 9.

This just about wraps up our coverage of how to create new classes, but we have one unpleasant task left to cover: how to dispose of, or deallocate, old and unwanted objects. If you don't destroy these objects, they'll eventually consume enough memory that the iPhone will think your application is the next Chernobyl and shut it down to avoid impacting other phone functionality!

5.5.4 Destroying objects

Once you finish using an object, you must put up your hand and tell the runtime that the memory allocated to the object can be reutilized for other purposes. If you forget this step, the memory will forever hold your object even if you never use it again. Unused objects gradually consume the limited memory available to an iPhone until the OS decides you've exhausted your fair share and abruptly shuts down your application.

In Objective-C (at least when targeting the iPhone), it's your responsibility to manage memory usage explicitly. Unlike languages such as C# and Java, Objective-C has no automated detection and reclaiming of unwanted objects, a feature commonly known as *garbage collection*. The following listing demonstrates the typical lifecycle of an Objective-C object from cradle to grave.

Listing 5.7 The typical lifecycle of an Objective-C object

```
CTRentalProperty *newRental = [[CTRentalProperty alloc]
                               initWithAddress:@"13 Adamson Crescent"
                               rentingFor:275.0f
                               andType:TownHouse];
... make use of the object ...
[newRental release];
newRental = nil;
```

The object is allocated by a call to `alloc`. Once allocated and initialized, the object is used until it's no longer necessary. At this stage a `release` message indicates that the object exceeds requirements and its memory can be reused. It can also be handy to blank out (or `nil`) the associated variable to make it easier to determine whether or not a given variable is pointing to a valid object.

Although you should call `release` to indicate your lack of interest in an object, that doesn't immediately guarantee that the object will be destroyed. Other parts of your application may be interested in keeping the same object alive, and it's only when the last reference is released that the object will be freed (more about this in chapter 9).

When `release` finally determines that nobody is interested in keeping the current object alive, it automatically invokes another method, called `dealloc`, to clean up the object. The following listing demonstrates how to implement `CTRentalProperty`'s `dealloc` method.

Listing 5.8 Implementing `dealloc` to clean up the resources acquired by an object

```
- (void)dealloc {
    [address release];
    [super dealloc];
}
```

`dealloc` is an ideal place not only to free any memory allocated by your class but also to tidy up any system resources (such as files, network sockets, or database handles) the class may be holding onto.

One thing to note is that, although declared properties can automatically provide getter and setter implementations, they won't generate code in `dealloc` to free their associated memory. If your class contains properties using `retain` or `copy` semantics, you must manually place code in your class's `dealloc` method to clean up their memory usage. In listing 5.8 this is demonstrated by the `address` property. When destroying a rental property object, you want the copy of the address string you made in the property's setter to be cleaned up.

Finally, most `dealloc` method implementations end with a call to `[super dealloc]`. This gives the superclass a chance to free any resources it has allocated. Notice that the order here is important. Unlike with an `init` implementation, where it's important to give the superclass a chance to initialize before you initialize your own content, with a `dealloc` method, you must tidy up your own resources, and then give the superclass a chance to tidy up.

5.6 Using the class in the Rental Manager application

Now that you've created a class called `CTRentalProperty` and gone through various iterations to improve its source code, let's update the Rental Manager application.

The first step is to open `RootViewController.h` and remove the existing definitions for the `PropertyType` enumeration and `RentalProperty` structure. Replace them with the `CTRentalProperty` class. At the same time, you can add an `NSArray`-based instance variable to store the list of rental properties. With these changes made, you should end up with `RootViewController.h` looking similar to the following listing.

Listing 5.9 `RootViewController.h` storing rental properties in an object-oriented manner

```
@interface RootViewController : UITableViewController {
    NSDictionary *cityMappings;
    NSArray *properties;
}
@end
```

Using an `NSArray` to store the rental properties enables you to eventually add and remove properties while your application runs. This is something you can't easily do with the C-style array used previously.

In comparison, `RootViewController.m` requires a number of larger source code changes, so replace its existing contents with the following listing.

Listing 5.10 `RootViewController.m` updated to use the `CTRentalProperty` class

```
#import "RootViewController.h"
#import "CTRentalProperty.h"
@implementation RootViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString *path = [[NSBundle mainBundle]
                     pathForResource:@"CityMappings"]
    }
```

← 1 **Import the `CTRentalProperty` definition**

```

        ofType:@"plist"];
cityMappings = [[NSDictionary alloc]
    initWithContentsOfFile:path];

properties =
    [[NSArray alloc] initWithObjects:
        [CTRentalProperty
            rentalPropertyOfType:TownHouse
            rentingFor:420.0f
            atAddress:@"13 Waverly Crescent, Sumner"],
        [CTRentalProperty
            rentalPropertyOfType:Unit
            rentingFor:365.0f
            atAddress:@"74 Roberson Lane, Christchurch"],
        [CTRentalProperty
            rentalPropertyOfType:Unit
            rentingFor:275.9f
            atAddress:@"17 Kipling Street, Riccarton"],
        [CTRentalProperty
            rentalPropertyOfType:Mansion
            rentingFor:1500.0f
            atAddress:@"4 Everglade Ridge, Sumner"],
        [CTRentalProperty
            rentalPropertyOfType:Mansion
            rentingFor:2000.0f
            atAddress:@"19 Islington Road, Clifton"],
        nil];
    }

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [properties count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *cellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:cellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:cellIdentifier]
            autorelease];
    }

    CTRentalProperty *property =
        [properties objectAtIndex:indexPath.row];

    int indexOfComma = [property.address
        rangeOfString:@","].location;
    NSString *address = [property.address
        substringToIndex:indexOfComma];
    NSString *city = [property.address
        substringFromIndex:indexOfComma + 2];

```

2 Create new object

3 Update the method


```

cell.textLabel.text = address;
NSString *imageName =
    [cityMappings objectForKey:city];
cell.imageView.image =
    [UIImage imageNamed:imageName];
cell.detailTextLabel.text =
    [NSString
    stringWithFormat:@"Rents for $%0.2f per week",
    property.rentalPrice];

return cell;
}

- (void)dealloc {
    [cityMappings release];
    [properties release];
    [super dealloc];
}

@end

```

3 Update
the
method

4 Release the
memory

The first change found in `RootViewController.m` is the importation of the `CTRentalProperty.h` header file ❶ via an `#import` statement. This statement requests that the Objective-C compiler read the contents of the specified file and interpret it as if its contents were written directly at the point of the `#import` statement. In other words, this line makes the compiler aware of the definition of the `CTRentalProperty` class and allows you to use it. Without this line, the compiler wouldn't know about the existence of the `CTRentalProperty` class—at least, not while compiling `RootViewController.m`.

What about `#include`? What's this nonstandard `#import`?

If you have knowledge of C or C++, you may know that C-based languages traditionally use an `#include` statement to include the contents of another file. Objective-C, by contrast, introduced and recommends the use of the `#import` statement.

The difference in behavior is subtle. If two `#include` statements specify the same file, the compiler will probably complain that it has seen multiple definitions for the classes and type definitions found in the file. This is because the `#include` statement reads the contents of the specified file and passes it to the Objective-C compiler. With multiple `#include` statements for the same file, the compiler sees the source code multiple times.

The `#import` statement, on the other hand, has slightly more smarts to it. If a second `#import` statement is found for the same file, it is, in effect, ignored. This means the contents of the header file are seen only once by the compiler, no matter how many times they're requested to be imported.

`#import` is similar to `#pragma once` and other such solutions that many C and C++ compilers support as extensions to provide similar convenience.

The next major change is found in `viewDidLoad` ❷. When your Rental Manager application first becomes visible, you create a new `NSArray` object and populate it with

a number of rental property objects, which are created using `CTRental`'s `rentalPropertyOfType:rentingFor:atAddress:` class method.

With the array of rental property objects constructed, you're ready to update your implementation of the `tableView:cellForRowAtIndexPath:` method ③. The first change uses the `objectAtIndex:` method available on an `NSArray` to obtain the `CTRentalProperty` object for a particular table view cell and assign it to a variable called `property`. You then access the `CTRentalProperty` object's various properties, such as `address` and `rentalPrice`, to update the display.

The final change ④ makes sure you release the memory storing your array of rental property details when the view gets deallocated as the application shuts down.

This completes the changes required to make the main data storage in the Rental Manager application object-oriented. Although the advantages of your work in this chapter aren't yet obvious, the next few chapters reveal the power of developing the application in an object-oriented manner.

5.7 Summary

Implementing custom classes is a great way to organize your application logic into discrete and easily maintainable units. With a well-structured class hierarchy, it should be possible to make significant changes to the internal implementation of a class without affecting other aspects of your code, as long as you keep the same public interface. As an example, you could separate the storage of a rental property's address into separate street number, street name, suburb, and city instance variables, and then recombine them as part of the setter method for the address property. Users of the `CTRentalProperty` class wouldn't need to be aware of this change because the interface of the class wouldn't have changed.

In the discussions of declared properties, you may have noticed that the same identifier can be used to name multiple elements in a class without causing an error. For example, in an Objective-C class it's possible for

- An instance method to have the same name as a class method
- A method to have the same name as an instance variable

Declared properties are a good example of this. It's common (but not required) for the backing instance variable to have the same name as the property's getter method.

One oddity of Objective-C is the lack of class-level variables. If you require a variable to belong to a class but not to any particular instance, your best solution is to declare a C-style global variable as follows:

```
static NSMutableDictionary *sharedCache;
```

On a similar note, Objective-C lacks visibility modifiers for methods and properties. Unlike instance variables, there are no `@public`, `@protected`, or `@private` attributes for methods. In chapter 6, we cover a partial solution to this situation and explore how to build on the foundations of a single class to create groups of closely related classes.

Extending classes



This chapter covers

- Subclassing
- Extending classes
- Creating custom objects
- Class clusters
- Categories
- Existing class modification

Classes, which consist of instance variables and methods, can be extended by being modified, expanded, or abstracted. Class extension is one of the fundamental ideas of object-oriented programming. The ability to extend a class promotes code reuse and compartmentalization, and it's the sign of a well-thought-out project. We investigate the subtleties of the various approaches to extending classes.

6.1 Subclassing

In chapter 5, you learned how to create your own classes. If you look back, you'll see that the classes you made, such as `CTRentalProperty`, are subclasses of `NSObject`. `NSObject` is the most basic class that Objective-C provides. You'll subclass `NSObject` many times when creating original classes for your application. This isn't to say it's

the only class you can subclass: any class in Objective-C can be extended by subclassing. `NSObject` is explored in more detail later in this chapter.

6.1.1 What is subclassing?

Subclassing is the act of taking a class, with all of its methods and instance variables, and adding to it. A classic example of subclassing logic is used in taxonomy, when classifying species. A human is a mammal; a tiger is a mammal as well. Tigers and humans can both be thought of as subclasses of the mammal class because they share some common features:

- Warm blood
- Vertebrae
- Hair

While tigers and humans don't share a ton of similarities, they do share these features and, as a result, they're both subclasses of the mammal class. This is exactly the idea that's applied when subclassing in an object-oriented programming language. You subclass something when you want to keep everything about it and add to it. Let's go through an example of subclassing in Objective-C.

You'll create a small iPhone application that utilizes some simple subclassing. First, you create a class called `Person` that describes some general attributes of a person: name, age, and gender.

- 1 Open Xcode and create a new View-based iPhone application called `PersonClass`.
- 2 Click `File > New File` and create a new Objective-C Class called `Person` as a subclass of `NSObject`.
- 3 Fill in `Person.h` with the content of the following listing.

Listing 6.1 Creating the `Person` class

```
#import <Foundation/Foundation.h>

typedef enum {
    Male, Female
} Gender;

@interface Person : NSObject {
    NSString *name;
    NSNumber *age;
    Gender gender;
}

- (id)initWithName:(NSString *)_name;
- (id)initWithAge:(NSNumber *)_age;
- (id)initWithGender:(Gender)_gender;
- (id)initWithName:(NSString *)_name
  age:(NSNumber *)_age
  gender:(Gender)_gender;

@end
```

This is a very simple class that has instance variables (ivars) that all people share. Every person has a name, age, and gender. Let's subclass the `Person` class by making `Teacher` and `Student` classes.

Click `File > New File` and create a new Objective-C Class called `Teacher` as a subclass of `Person`, as shown in the following listing.

Listing 6.2 Creating the Teacher class

```
#import <Foundation/Foundation.h>
#import "Person.h"

@interface Teacher: Person {
    NSArray *classes;
    NSNumber *salary;
    NSString *areaOfExpertise;
}

- (id)initWithName:(NSString *)_name
  age:(NSNumber *)_age
  gender:(Gender)_gender
  classes:(NSArray *)_classes
  salary:(NSNumber *)_salary
  areaOfExpertise:(NSString *)_areaOfExpertise;

@end
```

Next, click `File > New File` and create a new Objective-C Class called `Student` as a subclass of `Person`, as shown in the following listing.

Listing 6.3 Creating the Student class

```
#import <Foundation/Foundation.h>
#import "Person.h"

@interface Student: Person {
    NSArray *classes;
    NSNumber *numberOfCredits;
    NSString *major;
}

- (id)initWithName:(NSString *)_name
  age:(NSNumber *)_age
  gender:(Gender)_gender
  classes:(NSArray *)_classes
  numberOfCredits:(NSNumber *)_numberOfCredits
  major:(NSString *)_major;

@end
```

Now you've created your `Teacher` and `Student` classes as subclasses of `Person`. A `Teacher` object now has a name, age, and gender, as defined in `Person`, but it is extended by adding a list of classes, salary, and area of expertise. Similarly, a `Student` has a name, age, and gender as well as a list of classes, a number of credits, and a major. In this example, the *superclass* of both `Teacher` and `Student` is `Person`. The

superclass of a class is the class that a new class is subclassing. Although this is the most straightforward example of subclassing, you can subclass in many different ways, with subtle distinctions between them.

6.2 Adding new instance variables

Now that your class files are created and their header files filled in, you can continue to implement them. You need to start filling in the .m files for all the classes you created. The .h file simply outlines what types of operations and variables a class uses. The .m file is the piece of code that implements the creation of these variables and the execution of the declared methods.

Any instance variable that your subclass has must be initialized. Right now, these variables have types and names, but you must tell the application to allocate some space in memory for them and to connect these variable names to actual memory. This is usually done in two parts. First, the variable needs to have space allocated for it, using the `alloc` method. Second, the variable must be initialized, using the `init` method of your class. Every class should have an `init` method where the instance variables and general setup for the class are performed. Let's set up the `init` methods for your new classes.

Go into the `Person.m` file. You'll see an essentially blank class. You'll make the initialization method for this class so that you can use it in your project. Enter the code from the following listing into `Person.m`.

Listing 6.4 Creating the `init` method for the `Person` class

```
#import "Person.h"

@implementation Person

- (id)init {
    1 A blank initialization
    ← method
    if ((self = [super init])) {
        name = @"Person";
        age = [NSNumber numberWithInt:-1];
        gender = Male;
    }

    return self;
}

- (id)initWithName:(NSString *)_name {
    2 Method takes
    ← name for person
    if ((self = [super init])) {
        name = _name;
        age = [NSNumber numberWithInt:-1];
        gender = Male;
    }

    return self;
}

- (id)initWithAge:(NSNumber *)_age {
    3 Method takes
    ← age for person
    if ((self = [super init])) {
```

```

        name = @"Person";
        age = _age;
        gender = Male;
    }

    return self;
}

- (id)initWithGender:(Gender)_gender {
    if ((self = [super init])) {
        name = @"Person";
        age = [NSNumber numberWithInt:-1];
        gender = _gender;
    }

    return self;
}

- (id)initWithName:(NSString *)_name
    age:(NSNumber *)_age
    gender:(Gender)_gender {
    if ((self = [super init])) {
        name = _name;
        age = _age;
        gender = _gender;
    }

    return self;
}

@end

```

4 Method takes gender for person

5 Method takes name, age, and gender values

With this listing, you create several different `init` methods that you can use when creating the `Person` class. Let's go over them:

- ❶ This is a blank initialization method. Calling it creates a `Person` object that has no name, age, or gender defined. The default values are automatically assigned.
- ❷ This method takes in the name for the person. The name variable is assigned to whatever was passed in, and the remaining variables are set to the default values.
- ❸ This method takes in the age for the person. The age variable is assigned to whatever was passed in, and the remaining variables are set to the default values.
- ❹ This method takes in the gender for the person. The gender variable is assigned to whatever was passed in, and the remaining variables are set to the default values.
- ❺ This method takes in a value for the name, age, and gender of the `Person` object being created. This method defines all the variables of the person class to be nondefault values.

The `init` methods show all the different ways a class can be initialized. If you were making a project that needs to initialize with any of the instance variables that are set, then you would do something like this. Most classes, however, need only one initialization method. In the following listing, you create only one initialization method for the `Teacher` and `Student` classes.

Listing 6.5 Creating the init method for the Teacher class

```

- (id)initWithName:(NSString *)_name
  age:(NSNumber *)_age
  gender:(Gender)_gender
  classes:(NSArray *)_classes
  salary:(NSNumber *)_salary
  areaOfExpertise:(NSString *)_areaOfExpertise {

  if ((self = [super init])) {
    name = _name;
    age = _age;
    gender = _gender;
    classes = _classes;
    salary = _salary;
    areaOfExpertise = _areaOfExpertise;
  }

  return self;
}

```

Now you create a similar method for the Student class. The Student initializer takes the name, age, and gender parameters because it's a subclass of Person as well, but it takes slightly different parameters than the Teacher class. Overall, the method is similar to the Teacher init method, as the following listing shows.

Listing 6.6 Creating the init method for the Student class

```

- (id)initWithName:(NSString *)_name
  age:(NSNumber *)_age
  gender:(Gender)_gender
  classes:(NSArray *)_classes
  numberOfCredits:(NSNumber *)_numberOfCredits
  major:(NSString *)_major {

  if ((self = [super init])) {
    name = _name;
    age = _age;
    gender = _gender;
    classes = _classes;
    numberOfCredits = _numberOfCredits;
    major = _major;
  }

  return self;
}

```

Now that you have all of these methods filled in, you can create Person, Teacher, and Student objects. Before you start using these classes, let's look into expanding the ways you can access all of the attributes of these classes.

6.3 Accessing existing instance variables

You now have these three classes, each with instance variables. You created some initialization methods to fill in the instance variables, but what do you do if you create a

Person object with only a name, and later in your execution you want to fill in the age and gender? Moreover, after you create a Person object, how can you access the name you've set? This issue is handled through getter and setter methods. Getter methods return the value of some instance variable. Setter methods can set the value of an instance variable.

There are many different approaches to making these types of methods for classes, because they're a very common part of an object-oriented class. You'll use a method of manually making the get and set methods for your class.

6.3.1 *Manual getters and setters approach*

The first way to create getters and setters for the instance variables of a class is to manually create the methods. The getter method name retrieves a variable named `instanceVariable` of type `VariableType`, and it should have the following format:

```
- (VariableType)instanceVariable
```

The setter methods should be formatted like this:

```
- setInstanceVariable:(VariableType)_instanceVariable
```

For the Person class, you make getter and setter methods for the name, age, and gender instance variables that the class contains. Add the following code into the Person.h file so that the methods are declared:

```
- (NSString *)name;
- (NSNumber *)age;
- (Gender)gender;

- (void)setName:(NSString *)_name;
- (void)setAge:(NSNumber *)_age;
- (void)setGender:(Gender)_gender;
```

Now you need to fill in the methods in Person.m, as shown in the following listing.

Listing 6.7 Declaring Person's getter and setter methods

```
- (NSString *)name {
    if(name) {
        return name;
    }
    return @"--unknown--";
}

- (NSNumber *)age {
    if(age) {
        return age;
    }
    return [NSNumber numberWithInt:-1];
}

- (Gender)gender {
    if(gender) {
```

```

        return gender;
    }

    return -1;
}

- (void)setName:(NSString *)_name {
    name = _name;
}

- (void)setAge:(NSNumber *)_age {
    age = _age;
}

- (void)setGender:(Gender)_gender {
    gender = _gender;
}

```

Getter and setter methods are pretty self-explanatory. Getter methods return an instance variable with the same name. Setter methods take in an object and set it to an instance variable. The only point of note here is that in the getter methods, make sure to check if the variables exist. If the name getter was called, for instance, and the name was never set, an error would be returned without the check. If the name didn't exist, a nil value would be returned.

This method is more time consuming than the method we examine next, but it also allows a nice cutoff point to perform operations when a class's instance variables are requested. If, for instance, you want to count the number of times a variable is called or set, you can do so with these methods.

This is the classic way that getters and setters used to be created, but most developers now use properties to create them. Properties were introduced by Apple to automatically generate these very common methods. Refer to section 5.4 for more information on this automated approach.

6.4 Overriding methods

Overriding methods is another important aspect of subclassing in Objective-C. Just as a subclass of any class retains the class's instance variables, it also gains access to all the class's *instance methods*. If you call a method to a class and the compiler sees that a method with that name doesn't exist in the class, it goes to the superclass and checks for the method there (see figure 6.1).

Generally, method overriding is the act of replacing a method from the parent class with a method that does something different. This is commonly done when creating class structures, because methods may need to perform slightly differently in a subclass than they do in the class in which they're first declared.

Let's do some method overriding in your new classes. The NSObject class has a method called

```
- (NSString)description;
```

This is a helpful class to override when creating custom classes. It's likely that during development you'll be taking advantage of the NSLog operation to help debug your

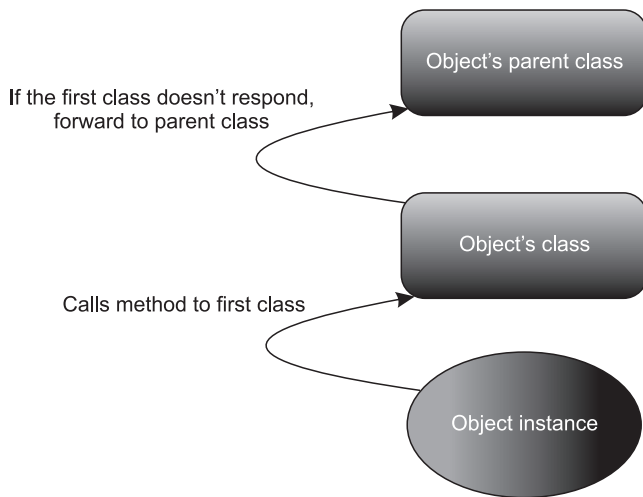


Figure 6.1 The compiler checking the superclass

applications. NSLog supports C-based placeholders such as %@ and %d. When the %@ placeholder is put in an NSString, the placeholder is filled with what is returned by this method. You'll implement this method in your three classes and see it work in an NSLog.

6.4.1 *Overriding the description method*

Overriding a method is done by simply declaring it in the class in which you would like to call it. Start in the Person class, go into the Person.m file, and declare the method shown here:

```

- (NSString *)description {
    if (gender == Male) {
        return [NSString stringWithFormat:
            @"Hi! I am a man, named %@, who is %@ years old",
            name, age];
    } else {
        return [NSString stringWithFormat:
            @"Hi! I am a woman, named %@, who is a %@ years old",
            name, age];
    }
}

```

The description method is declared in NSObject. By placing the same method in your class, you're telling this class that rather than using the NSObject version of the description method, you want this method to be called. Now you override the description methods for both Teacher and Student, taking advantage of Person's description method in the ones you create in Student and Teacher so that you can write as little code as possible:

```

- (NSString *)description {
    if (gender == Male) {
        return [NSString stringWithFormat:
            @"%@. I am a male currently teaching %@ "

```

```

        "for $%@ per year with expertise in %@",
        [super description], classes, salary, areaOfExpertise];
    } else {
        return [NSString stringWithFormat:
            @"%@. I am a female currently teaching %@ "
            "for $%@ per year with expertise in %@",
            [super description], classes, salary, areaOfExpertise];
    }
}

```

Once again, you've overridden the description method so that when you use it in your NSLog method, it will be called rather than the NSObject version of it (see the following listing).

Listing 6.8 Overriding the description method in Student

```

- (NSString *)description {
    if(gender == Male) {
        return [NSString stringWithFormat:
            @"%@. I am a male currently enrolled in %@ "
            "for %@ credits with %@ as my major",
            [super description], classes, numberOfCredits, major];
    } else {
        return [NSString stringWithFormat:
            @"%@. I am a female currently enrolled in %@ "
            "for %@ credits with %@ as my major",
            [super description], classes, numberOfCredits, major];
    }
}

```

In Teacher and Student, you use their superclass's (Person) description method to fill in the first part of the string and then use the individual instance variables of Student and Teacher, respectively. Custom methods you create can be overridden, and so can almost all of the include libraries' methods. Another commonly overridden method is

```
- (void)dealloc
```

This is the method used for memory management when objects need to be released. This method and its implementation are covered in more detail in chapter 9.

All that's left to do with these classes is test them and the methods you've made for them. If you go into your Application Delegate and enter the following code into the applicationDidFinishLaunchingWithOptions: method, you'll be able to see all of your classes and subclasses working together.

Listing 6.9 Test code for your subclasses

```

- (void)applicationDidFinishLaunching:(UIApplication *)application
withOptions:(NSDictionary*)options {
    // Override point for customization after app launch
    [self.window makeKeyAndVisible];
}

```

```

Person *person = [[Person alloc]
                  initWithName:@"Collin"
                  age:[NSNumber numberWithInt:23]
                  gender:Male];

Student *student = [[Student alloc]
                    initWithName:@"Collin"
                    age:[NSNumber numberWithInt:23]
                    gender:Male
                    classes:[NSArray arrayWithObjects:@"English",
                                                         @"Spanish",
                                                         @"Math", nil]
                    numberOfCredits:[NSNumber numberWithInt:12]
                    major:@"CS"];

Teacher *teacher = [[Teacher alloc]
                    initWithName:@"Winslow"
                    age:[NSNumber numberWithInt:30]
                    gender:Male
                    classes:[NSArray arrayWithObjects:@"ARM",
                                                         @"Imerssive Gaming",
                                                         @"Physical Computing", nil]
                    salary:[NSNumber numberWithInt:60000]
                    areaOfExpertise:@"HCI"];

NSLog(@"My person description is:\n%@", person);
NSLog(@"My student description is:\n%@", student);
NSLog(@"My teacher description is:\n%@", teacher);
}

```

Here you create `Person`, `Teacher`, and `Student` objects. At the end, you place a `%@` placeholder in your `NSLog` input. When you pass an object in after the string, the `%@` is replaced with what is returned from each object's `description:` method. With this done, you're going to look at another way to modify classes' functionality behind the scenes through a design method called *clusters*.

6.5 **Class clusters**

Class clusters is the Objective-C answer for the idea of abstraction. Generally, an abstract superclass is declared with methods that are implemented in nonvisible, concrete subclasses. This type of design is called an *abstract factory design* in Apple's documentation. It has the benefit of simplifying the complexity of public classes by revealing only the methods declared in the abstract parent class. The abstract superclass is responsible for providing methods that create instances of the private subclasses.

6.5.1 **Why use class clusters**

Class clusters is really just a fancy way of saying "secret subclassing." They allow developers to create a single doorway into multiple rooms. There are two common motivations for taking this design approach:

- 1 *Performance/Memory*

This type of design allows a developer to make a single class that can represent the creation point for multiple objects that have significantly different memory

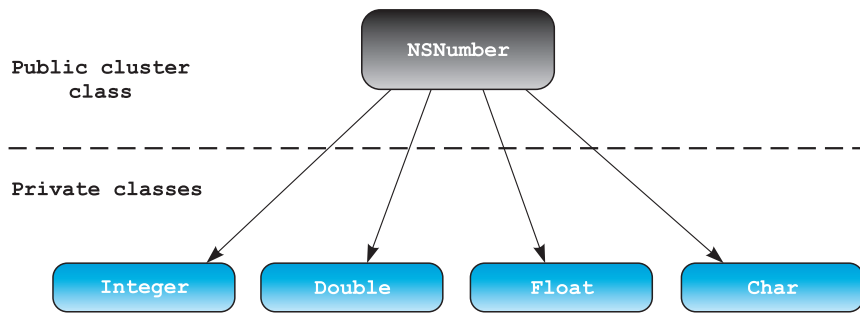


Figure 6.2 Graphical representation of NSNumber

requirements. The classic example of this is the NSNumber class that Apple provides. NSNumber is a cluster representing all sorts of different number types (Integer, Double, Long, Float, Char). All of these types can be converted from one to another, and all share many different methods that act on them. But they all require different amounts of memory. Therefore, Apple made NSNumber the cluster that handles the management of all the hidden subclasses that handle the different types. Figure 6.2 gives a graphical representation of the structure of NSNumber.

2 *Simplicity*

The other motivation for implementing this design is simplicity. As shown previously, creating a class cluster can seriously simplify the interface to a collection of similar classes. All the different objects contained in NSNumber would be very unwieldy to develop without the NSNumber wrapper. Operations such as conversion, or arithmetic operations like division would require developers to be very careful about memory management. NSNumber, however, shifts this responsibility from the developer to the class cluster, greatly simplifying the use of these similar classes.

6.5.2 **Multiple public clusters**

Clusters also give developers the ability to create multiple public class clusters that reference the same private implementation class. NSArray and NSMutableArray act as class cluster interfaces for a hidden implementation class that handles a lot of the class's functionality. The hidden implementation class is also called NSArray but isn't visible to developers. This is a memory- and simplicity-driven design decision. It allows developers of this class to create a single entry point for methods that will act slightly differently depending on the object being worked with. This takes the responsibility to choose the correct method to call from the developer using the class and shifts it to the intelligence embedded in the class cluster design. This means easy development when using Apple's class clusters and lots of flexibility if you want to make your own.

Clusters are a great design scheme for creating some type of custom object or collection of objects. Sometimes, however, you don't want to make something completely

unique, or you don't want to make it a subclass. Sometimes the best thing is to add to an existing class. A common example of this is when an application often needs to perform an operation on a `String`. Twitter, for example, uses a method that adds an `@` symbol before a username string, which is very useful. To address this issue, Objective-C supports a design scheme called *categories*.

6.6 Categories

In the previous section, you learned how to override methods in subclasses using the `description` method from `NSObject`. The categories design scheme is similar to the subclass method of overriding: it allows developers to add methods to a class that may not be there by default. Rather than creating a subclass and adding the method to it, categories allow a developer to add methods to any existing class without subclassing. This is usually done with the common data structure classes in Objective-C (`NSString`, `NSArray`, `NSData`). It's often convenient to add a method or two to these classes that will assist with an application you're making.

6.6.1 Extending classes without subclassing

Reuse is one of the core values of programming. Extending classes is often an effort to minimize the redundancy of code. It compartmentalizes objects and their abilities in a way that allows code reuse instead of copying and pasting code. Sometimes, however, developers want to add to existing classes rather than create their own. Subclassing has an inherent cost because it requires nonstandard class calls throughout an application. If a developer can write a method that attaches to the `NSString` class, for example, it would lend that piece of code to more common reuse.

Categories append content to the interface (header file) and implementation (main file) of existing classes, allowing a developer to create an application with lots of custom functions but few custom classes. Many people who are extending the API provided by Apple use this approach rather than making something completely original.

6.6.2 Using a category

Let's make a simple category class for `NSString`. For this example, assume your application needs to take all the vowels out of strings. You make a category class for `NSString` to accomplish this.

- 1 Create a new View-based project `VowelDestroyer`.
- 2 Click `File > New File` and create a new Objective-C Class called `VowelDestroyer` as a subclass of `NSObject`.
- 3 Insert the following code into `VowelDestroyer.h`:

```
#import <Foundation/NSString.h>
@interface NSString (VowelDestroyer)
- (NSString *)stringByDestroyingVowels;
@end
```

This code tells the compiler that you're adding to the `NSString` an interface called `VowelDestroyer`. This category adds a method called `stringByDestroyingVowels` that takes no parameters and returns an `NSString`.

You implement the method in `VowelDestroyer.m` so that it can be called when this method is called on an `NSString` object. Insert the following code into `VowelDestroyer.m`.

Listing 6.10 `VowelDestroyer.m`

```
#import "VowelDestroyer.h"

@implementation NSString (VowelDestroyer)
- (NSString *)stringByDestroyingVowels {
    NSMutableString *mutableString =
        [NSMutableString stringWithString:self];

    [mutableString replaceOccurrencesOfString:@"a"
        withString:@""
        options:NSCaseInsensitiveSearch
        range:NSMakeRange(0, [mutableString length])];
    [mutableString replaceOccurrencesOfString:@"e"
        withString:@""
        options:NSCaseInsensitiveSearch
        range:NSMakeRange(0, [mutableString length])];
    [mutableString replaceOccurrencesOfString:@"i"
        withString:@""
        options:NSCaseInsensitiveSearch
        range:NSMakeRange(0, [mutableString length])];
    [mutableString replaceOccurrencesOfString:@"o"
        withString:@""
        options:NSCaseInsensitiveSearch
        range:NSMakeRange(0, [mutableString length])];
    [mutableString replaceOccurrencesOfString:@"u"
        withString:@""
        options:NSCaseInsensitiveSearch
        range:NSMakeRange(0, [mutableString length])];
    return [NSString stringWithString:mutableString];
}
@end
```

1 Declare category name

2 Create NSMutableString

In this code, you declare the file to be an implementation of `NSString` again and declare your category name once more ①. Then you implement the method appropriately. In this case you create a new type of string called an `NSMutableString`. If you notice, `NSString`s can't be modified. `NSString` provides methods that modify a string and return a whole new one, but `NSMutableString` can be used to modify an existing string without having the modification methods return a whole new string. If this method were to be used a lot in an application, this implementation would be the best use of memory. After creating the `NSMutableString` ②, you replace each vowel with an empty string using the `replaceOccurrencesOfString:withString:options:range:` method. Using the `NSCaseInsensitive` option with this method matches both

uppercase and lowercase variations of the target string you provide. Once all the vowels have been replaced, you create a new `NSString` object from the mutable string and return it. Remember, you need to import this class into any class that you want to use it, but once you do so, you can call `stringByDestroyingVowels` on any `NSString` or `NSString` subclass.

6.6.3 *Considerations when using categories*

Categories are great for a lot of class extensions, but their limitations should be considered when using them in a project. First of all, a category can add only methods, not instance variables, to a class. If you need to add instance variables to a class, you have to subclass it. With some creativity, a lot of the functionality developers are looking for can be completed in a single method. The other limitation relates to overriding methods. Overriding should be done in subclasses rather than in categories. Otherwise, the compiler may get confused about which method to use, or worse, existing methods using that method and the whole class breaking. This practice should be avoided.

In order to see this category in action you can go to the app delegate of the project you've made to test it out. You'll need to import this class into your `VowelDestroyerAppDelegate.m` using the statement

```
#import "VowelDestroyer.h"
```

With this done, you can make a string in this class and call the `stringByDestroyingVowels` method. To make it easy we put some test code in the `applicationDidFinishLaunchingWithOptions` method. You can see the code below.

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
    withOptions:(NSDictionary*)options {

    // Override point for customization after app launch

    [self.window makeKeyAndVisible];
    NSString *originalString = @"Hello World";
    NSLog(@"Original String: %@", originalString);
    NSString *vowelFreeString = [originalString stringByDestroyingVowels];
    NSLog(@"Vowel Free String: %@", vowelFreeString);

    return YES;
}
```

6.7 *Subclassing in your demo application*

Your Property Rental application is coming along and is already full of code that takes advantage of subclassing. To illustrate the overall structure of your application, see the inheritance chart for your existing code (figure 6.3). The chart shows what classes you've created and from which parent classes they're subclassed. Currently, your application contains three classes and their subclasses.

Keeping the overall structure and inheritance chart of the classes in your project is important when developing. Remember, nearly all classes in Objective-C are eventually

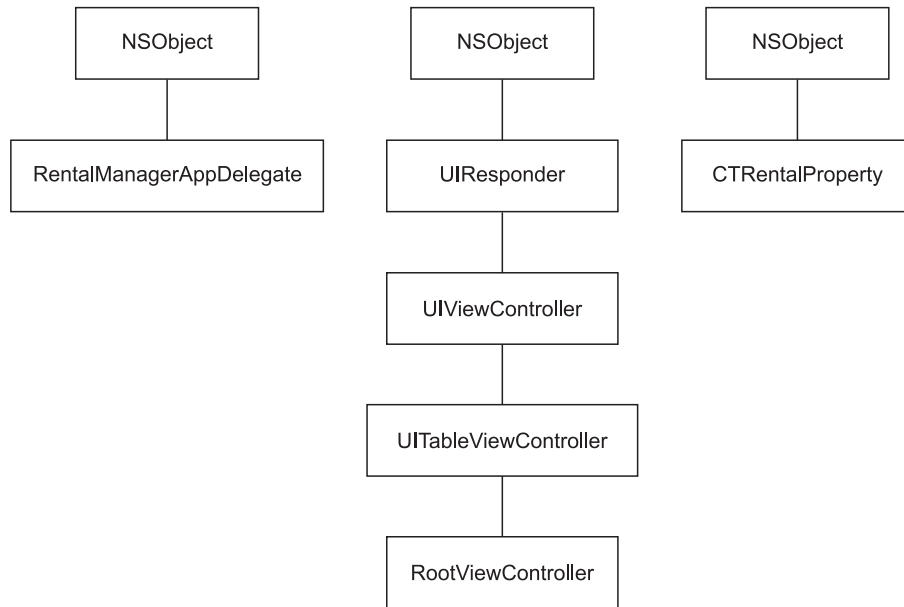


Figure 6.3 The inheritance chart of your project's current classes

going to inherit from `NSObject`; it's important to keep in mind the superclass of the class you're implementing.

6.7.1 Creating and subclassing `CTLease`

The final activity for this chapter is to create a few convenience methods for your rental application. You create a class called `CTLease`, which you subclass to represent a fixed-term lease or a periodic lease. These classes are nice examples of how class extension can be used within the context of a specific project, but will not end up being a part of your overall final design of the Rental Manager. Let's begin by making the `CTLease` class, which will be the parent. In your project, go to `File > New File...` and create a new `NSObject` subclass called `CTLease`. For now, you'll make this class as minimal as possible. In the header, insert the code shown in the following listing.

Listing 6.11 `CTLease.h`

```

#import <Foundation/Foundation.h>

@interface CTLease : NSObject {
}

+ (CTLease *)periodicLease:(float)weeklyPrice;
+ (CTLease *)fixedTermLeaseWithPrice:(float)totalRental
    forWeeks:(int)numberOfWeeks;

@end
  
```

The base class (CTLease) is a subclass of NSObject. It acts as a wrapper for the two hidden classes it represents. The class has two class methods for returning either a periodic lease or a fixed-term lease. The + sign before the declaration of the methods signifies each of these methods as a class method, meaning that these methods are called on CTLease itself rather than on some instance of the CTLease class. Let's look at implementing these methods. Input the code in the following listing to the main file of CTLease.

Listing 6.12 CTLease.m

```
#import "CTLease.h"

#import "CTPeriodicLease.h"
#import "CTFixedLease.h"

@implementation CTLease

+ (CTLease *)periodicLease:(float)weeklyPrice {
    CTLease *lease = [CTPeriodicLease
                     periodicLeaseWithWeeklyPrice:weeklyPrice];
    return [lease autorelease];
}

+ (CTLease *)fixedTermLeaseWithPrice:(float)totalRental
  forWeeks:(int)numberOfWeeks {
    CTLease *lease = [CTFixedLease fixedTermLeaseWithPrice:totalRental
                     forWeeks:numberOfWeeks];
    return [lease autorelease];
}

@end
```

The first thing to note here is that the code won't compile until you create the two accompanying subclasses (CTPeriodicLease and CTFixedTermLease), so when you build, you'll get errors. Don't worry about it right now: you'll remedy this problem shortly. CTLease doesn't even have an initializer in this case because it provides a single interface to create two different types of leases. You take in the appropriate parameters to create each type of lease and then return an autoreleased version of that object. While these methods seem simple, they're important in the overall architecture of your application. Let's look at creating the accompanying CTPeriodicLease and CTFixedTermLease.

6.7.2 *Creating CTPeriodicLease as a subclass of CTLease*

You need to make a concrete subclass for your periodicLease method in CTLease and call it CTPeriodicLease. Once again, you create a new file by going to File > New File.... You create an NSObject subclass, but you'll be changing it once the file is made. Name the file CTPeriodicLease. Once it's created, go into the header and replace it with the code in the following listing.

Listing 6.13 CTPeriodicLease.h

```
#import "CTLease.h"

@interface CTPeriodicLease : CTLease {
    float weeklyRental;
}

@property(n nonatomic) float weeklyRental;

+ (CTLease *)periodicLeaseWithWeeklyPrice:(float)weeklyPrice;

@end
```

Here a CTPeriodicLease is declared as a subclass of CTLease. A CTPeriodicLease contains a float number representing the weeklyrentalCost and provides a class method to create a CTLease object using the weekly price. Now all that's left to do is implement the method in your main file. Use the code shown in the following listing.

Listing 6.14 CTPeriodicLease.m

```
#import "CTPeriodicLease.h"

@implementation CTPeriodicLease

@synthesize weeklyRental;

+ (CTLease *)periodicLeaseWithWeeklyPrice:(float)weeklyPrice {
    CTPeriodicLease *lease = [CTPeriodicLease alloc];
    lease.weeklyRental = weeklyPrice;
    return [lease autorelease];
}

- (NSString *)description {
    return [NSString stringWithFormat:@"%$.2f per week",
        self.weeklyRental];
}

@end
```



Alloc
CTPeriodicLease ①

The creation of an object occurs ① where you allocate a CTPeriodicLease object. You set the lease's weekly rental price and return the autoreleased version of the object. One point of interest here is that this method is designated as returning a CTLease object, yet a CTPeriodicLease object is allocated and returned. The reason this is valid is that CTPeriodicLease is a subclass of CTLease, so CTPeriodicLease is a CTLease but a CTLease isn't necessarily a CTPeriodicLease. Next, the NSObject method description is overridden. You can do this because CTPeriodicLease is also a subclass of NSObject. As you learned earlier, this technique is useful when you want objects to print their attributes out of an NSLog.

6.7.3 Creating CTFixedLease as a subclass of CTLease

You now make another CTLease subclass called CTFixedLease so you can represent a lease as having a different set of terms rather than some periodic set amount. This

type of lease will have a fixed time and a fixed amount, and it won't recur. The header of `CTFixedLease` is similar to that of `CTLease` but in this case has ivars: `totalRental` and `numberOfWeeks`. Insert the code from the following listing into the header file.

Listing 6.15 `CTFixedLease.h`

```
#import "CTLease.h"

@interface CTFixedLease : CTLease {
    float totalRental;
    int numberOfWeeks;
}

@property(nonatomic) float totalRental;
@property(nonatomic) int numberOfWeeks;

+ (CTLease *)fixedTermLeaseWithPrice:(float)totalRental
  forWeeks:(int)numberOfWeeks;

@end
```

The implementation of `CTFixedLease` is similar to that of `CTPeriodicLease`. You set both instance variables for the class and return an autoreleased instance. You also make sure the description method prints out both of your instance messages in the description. Use the code in the following listing for the implementation of `CTFixedLease`'s main file.

Listing 6.16 `CTFixedLease.m`

```
#import "CTFixedLease.h"

@implementation CTFixedLease

@synthesize totalRental, numberOfWeeks;

+ (CTLease *)fixedTermLeaseWithPrice:(float)totalRental
  forWeeks:(int)numberOfWeeks {
    CTFixedLease *lease = [CTFixedLease alloc];
    lease.totalRental = totalRental;
    lease.numberOfWeeks = numberOfWeeks;
    return [lease autorelease];
}

- (NSString *)description {
    return [NSString stringWithFormat:@"$%0.2f for %d weeks",
        self.totalRental, self.numberOfWeeks];
}

@end
```

With this useful collection of classes in place, you can properly model the data for your application as you continue to build it.

6.8 Summary

Object-oriented programming languages have evolved to incorporate all sorts of design practices so that developers are empowered with the tools they need to create large, rich, and efficient applications in as little time as possible. Subclassing is one of the design foundations that have made this achievable. Before diving into coding a project, we recommend that you draw out your class design and look for opportunities to employ subclassing, clusters, and categories. The take-home points for these class extension strategies are as follows:

- If you want to add new instance variables to a class or override a method of a class, use subclassing.
- If you want to hide the implementation of a class, simplify a class's interface, or incorporate more code reuse, create a class cluster.
- If you want to add a method to a class, create a class category.

With a good dose of scrutiny and these design methods in mind, you can create the kind of code that will make your application a pleasure to use. In chapter 7 we look at an important design method applied throughout Objective-C and the Cocoa libraries: protocols. Protocols are sets of methods that a class can expect other classes to implement. This is useful when different classes are passing data and actions among themselves. Protocols are used in many of the APIs provided by Apple and are vital when using `UIKit` elements in your applications.

Now that you've learned all of the ways to create, modify, and group classes based on your design, you need to investigate how you can have these objects interact. By using protocols, the new classes you create can broadcast the kind requirements to objects that it will be interacting with. Apple relies heavily on this design method through all the `UIKit` objects and many Core Foundation objects. Chapter 7 presents an overview of the technology and a look at some of the most popular protocols in the software development kit.

7 *Protocols*

This chapter covers

- Protocol overview
- Defining protocols
- Conforming to a protocol
- Looking at examples of commonly used protocols

Now that you have some experience creating standard and custom objects, you need to look at how these objects interact. If two objects are to work together, you need to make sure they know how to talk to each other. For objects to talk to each other, they need to know what conversation they're having. The conversation between objects is usually represented by the protocol design scheme.

Sometimes when you create a class, you design it to be used as a tool for some other class. If you're going to do this, two things must be in place. First, your object must have a way to send callbacks to whatever class is trying to use it. Second, a class must be able to verify that whatever class is using it has the correct structure for it to communicate whatever it needs. This design pattern is used extensively through the classes provided in the iPhone software development kit (SDK) from Apple. You'll implement some type of protocol methods in every application you make.

Mastering the creation of a protocol for your own class and conforming to protocols for another class are vital to making a sturdy application.

Apple highlights three main motivations for using protocols in your design:

- To declare methods that others are expected to implement
- To declare the interface to an object while concealing its class
- To capture similarities among classes that aren't hierarchically related

The first reason is the most common reason for using protocols. A class will often require some kind of callback method to communicate with the object that's using it. A `Map` object would ask the object using it for the points of interest on the map where it should put pins, for instance. An alert view would want a way to tell the view using it that the user has selected a button from the alert. This idea is used commonly through many of Apple's provided classes. You can tell when a class is using protocols because they'll usually have an instance variable called *delegate* or *data source*. The methods that delegates or data sources are supposed to implement either provide some data that the caller needs or are given the result from the object implementing the protocol. This may seem abstract at the moment, but with more use of the Apple frameworks and the examples in this chapter, the relationship between the protocol definer and implementer will become clearer.

The second and third reasons for creating protocols have to do with design decisions that can come about. Sometimes when creating a reusable piece of code, you want to conceal the actual classes or code implementing some functionality. If so, you can create a protocol to represent the class, which means the user will create a version of the object you created and also complete its protocol methods. This gives a concealed class the ability to call back to a class using it without revealing its "secrets." On those philosophical notes, let's look into some concrete examples of this abstract idea to see how it can assist you in your development and design.

7.1 Defining a protocol

When you want to create a protocol for a project, the first step is to define what the protocol consists of. Protocols, much like instance variables and methods, need to be declared in the header file of a class and defined in the main file. There are two sides to a protocol: the side that defines and calls them and the side that implements them. Protocols are used in many of the classes provided by Apple. When a table view needs to be informed of how many cells are in a given section, it asks its delegate that conforms to the `UITableViewDataSource` protocol. Here we focus on outlining a protocol for a class that another class will conform to.

Let's build a small example using the idea of protocols. You create a class with a protocol, and later in the chapter, you implement that protocol in another class to see how it all fits together. The class you create is a subclass of `UIView`. It'll have a protocol that can inform its delegate when it has finished a certain animation that you'll perform. Let's get started:

- 1 Create a new Window-based iPhone Application project called `myProtocol`.
- 2 Click File > New File....
- 3 Create a `UIView` subclass called `myView`. You define your protocol in the `myView.h` file. Put the code from the following listing there.

Listing 7.1 Filling in `myView.h`, part 1

```
#import <UIKit/UIKit.h>

@protocol animationNotification
- (void)animationHasFinishedWithView:(UIView *)animatedView;
@optional
- (void)animationStartedWithView:(UIView *)animatedView;
@end

@interface myView : UIView {
    id <animationNotification> delegate;
    UIView *boxView;
}
@property (nonatomic, assign) id delegate;
- (void)animate;
@end
```

← 1 **Declare the animationNotification delegate**

This code defines all of the things that make up the `UIView` subclass you'll use along with the `animationNotification` protocol that it defines. Line 1 is particularly important to defining a protocol in this class: it defines a variable of type `id` with name `delegate`. This is the holder for the class that will implement your protocol. It gives you a way to communicate to that class when you need your protocol methods to fire. The class that implements this protocol must set the `delegate` to be itself. We go over code for this when we look into implementing a protocol.

Now that a `delegate` is defined, you can start to define `animationNotification` methods. Once you describe the interface for `myView`, you use the `@` syntax to start the definition of your protocol. Any methods you list *after* this protocol will be required for a class to implement your protocol. You can use the `@optional` marker to list methods that aren't required for a class implementing your protocol. This is all that needs to be done for a protocol to be defined.

7.2 Implementing a protocol

Now that your protocol is declared, it's time to implement it. This takes place in two steps. The first is implementing the functionality of calling the protocol methods from the class declaring the protocol. The second is writing these protocol methods into a class that implements the protocol.

7.2.1 Creating the protocol method callers

Here is where you fill in the code for `myView.m`. This class makes a view with a red square in the middle of it. You add the square to your view and make a method called `animate` that moves the square down by 100 pixels. Let's see what this code looks like in the following listing. The listing shows each method individually so that their purposes can be explained.

Listing 7.2 Filling in `myView.m`, part 2

```
#import "myView.h"

@implementation myView

@synthesize delegate;

- (id)initWithFrame:(CGRect)frame {
    if ((self = [super initWithFrame:frame])) {
        [self setBackgroundColor:[UIColor blackColor]];
        boxView = [[UIView alloc]
                    initWithFrame:CGRectMake(50, 30, 220, 220)];
        [boxView setBackgroundColor:[UIColor redColor]];
        [self addSubview:boxView];
    }
    return self;
}
```

This initialization method sets up the view. You turn the main view's background to black, initialize `boxView`, and set its background to red. Then you add the `boxView` as a subclass of the `myView` class you're working with, and finally you return an instance of the `myView` class. Notice that no work needs to be done in reference to the protocol that `myView` defines. Next we look at the method you call to get the square to move down the screen (see listing 7.3). You accomplish this with the `beginAnimations:context:` method provided by `UIView`. This method is convenient for changing things about `UIKit` elements. Once you set up the animation properties, any attribute you change to `UIKit` elements (position, size, rotation, alpha) will animate from its current state to the altered state.

Listing 7.3 Filling in `myView.m`, part 3

```
- (void)animate {
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:2];
    [UIView setAnimationDelegate:self];
    [UIView setAnimationWillStartSelector:@selector(animationStarted)];
    [UIView setAnimationDidStopSelector:@selector(animationStopped)];
    CGRect newFrame = CGRectMake(boxView.frame.origin.x,
                                boxView.frame.origin.y + 100,
                                boxView.frame.size.width,
                                boxView.frame.size.height);

    [boxView setFrame:newFrame];
    [UIView commitAnimations];
}
```

In this animation, you take the `boxView` and move it down the screen by 100 pixels. Calling the `commitAnimations` method executes the animation. An important configuration step of this animation was setting `animationWillStartSelector` and `animationDidStopSelector`. These tell `UIView` to call certain classes when the animation starts and ends. These method calls are forwarded to the delegate and the protocol methods are called. The only thing left to do is create the `animationStarted` and `animationStopped` methods that alert the delegate (see the following listing).

Listing 7.4 Filling in `myView.m`, part 4

```
- (void)animationStarted {
    if ([delegate
        respondsToSelector:@selector(animationStartedWithView:)])
    {
        [delegate animationStartedWithView:self];
    }
}

- (void)animationStopped {
    if ([delegate
        respondsToSelector:@selector(animationHasFinishedWithView:)])
    {
        [delegate animationHasFinishedWithView:self];
    }
}

- (void)dealloc {
    [boxView release];
    [super dealloc];
}

@end
```

Here is where the protocol methods are called. You tell the delegate to fire off the appropriate method when the animation has started and stopped. You don't want to assume that a delegate is set or that the delegate has implemented the protocol methods. To check this, you use the `respondsToSelector` method. `NSObject` implements this method, so you can call it on nearly any object. It's a great check to include in your code to avoid crashing and generally a good coding practice. If the delegate is assigned and it does implement the protocol methods, you call the method and pass it the view in which the protocols are defined. Now that you've defined your protocol, let's create a class that will conform to it.

7.2.2 *Making a class conform to a protocol*

To make your application delegate conform to the `animationNotification` protocol, you implement both methods from the protocol in your application delegate. First you signal that the application delegate does conform to the `animationNotification` protocol. This is done in the `myProtocolAppDelegate.h` file. The following listing shows the new code for your App Delegate header file.

Listing 7.5 Filling in myProtocolAppDelegate.h

```
#import <UIKit/UIKit.h>
#import "myView.h"

@interface myProtocolAppDelegate : NSObject
    <UIApplicationDelegate, animationNotification> {

    UIWindow *window;
    myView *view;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;

- (void)animate;

@end
```

The class declares that it implements the `animationNotification` protocol. This class also implements the `UIApplicationDelegate` protocol. You can list however many protocols you want your class to conform to between the `<>` with commas separating them. Aside from that, an instance of the `UIView` subclass is declared so that you can tell it to animate when a button is pressed. All that's left to do is fill in the `AppDelegate` with the methods specified by the protocol.

Insert the code from the following listing into your `myProtocolAppDelegate.m` file.

Listing 7.6 Filling in myProtocolAppDelegate.m, part 1

```
#import "myProtocolAppDelegate.h"

@implementation myProtocolAppDelegate

@synthesize window;

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    view = [[myView alloc]
        initWithFrame:[UIScreen mainScreen] bounds]];
    [view setDelegate:self];
    [window addSubview:view];
    UIButton *animateButton = [UIButton
        buttonWithType:UIButtonTypeRoundedRect];
    [animateButton setTitle:@"Animate"
        forState:UIControlStateNormal];
    [animateButton
        addTarget:self
        action:@selector(animate)
        forControlEvents:UIControlEventTouchUpInside];
    [animateButton setFrame:CGRectMake(25, 380, 270, 30)];
    [window addSubview:animateButton];

    [window makeKeyAndVisible];

    return YES;
}
```

← 1 Create myView

← 2 Create the button to call the method

← 3 Make the button animate when tapped

The first thing this code does is fill in the `applicationDidFinishLaunching` method, which is part of the `UIApplicationDelegate` protocol. In here you create a button that calls another method you'll define. You create the `myView` object and set the `myView` delegate to this class ❶. Then you add the view into your window. Next you make a button in code ❷. This is the button you'll tap to call the `animate` method in `myView` ❸. With this done, you must make a method that tells `myView` to animate when the button is pressed. The code is short—check it out; this should be added as a method to `myProtocolAppDelegate.m`:

```
- (void)animate {
    [view animate];
}
```

This code calls the `animate` method in `myView`. All that's left to do is define the protocol methods and implement what you want them to do. For this example, two things happen when the animation starts, as shown in the following listing.

Listing 7.7 Filling in `myProtocolAppDelegate.m`, part 2

```
- (void)animationStartedWithView: (UIView*)animatedView {
    NSLog(@"The animation has started");
    [animatedView setBackgroundColor:[UIColor whiteColor]];
}
                                     Write a log message
                                     ←
                                     Set background to white
                                     ←

- (void)animationHasFinishedWithView: (UIView*)animatedView {
    NSLog(@"The animation has finished");
    [animatedView setBackgroundColor:[UIColor blackColor]];
}
                                     Write a log message
                                     ←
                                     Set background to black
                                     ←

- (void)dealloc {
    [view release];
    [window release];
    [super dealloc];
}
@end
```

Fire up the simulator and see this in action. If your terminal is up, you can see the log messages as well. You've successfully defined and implemented a custom protocol.

7.3 *Important protocols*

Now that you've seen all the guts of making and implementing a protocol, let's look at the most popular protocols used in the development of an iPhone application. Apple relies on the protocol design method for many of its Cocoa classes. In this section we review four sets of protocols that you'll likely use when developing your application. Let's dive in.

7.3.1 `<UITableViewDataSource>`

The `UITableViewDataSource` protocol is a mandatory protocol to have implemented for any table view in your application. A table view is a visual element that's meant to display some collection of information. Unlike any other user interface element in the

iPhone UIKit, UITableView must be provided the data it'll display. Without the data source protocol for a UITableView connected, a table view has no functionality. Let's look at the required methods of this protocol and then look at some of the more popular optional methods.

<UITableViewDataSource> REQUIRED METHOD

This is the first method that needs to be implemented in a class conforming to the UITableViewDataSource:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger) section
```

This code specifies how many cells should be requested in a given section. To understand this method, you must first understand what *sections* are. Table views present sections in two different manners, depending on the style set for the table view.

If the table view has its style set to UITableViewStylePlain, the sections will be separated, by default, by a gray line with some text in it. This is seen in the Contacts portion of the Phone application on the iPhone or when listing artists in the iPod. It's commonly used to separate the first letter of items in a list, but it can also be used to segment data in any way you choose.

If the table view has its style set to UITableViewStyleGrouped, each section will be separated by some space with the top and bottom table view cells having rounded corners. An example of this can be seen in the Settings section of the iPhone. Depending on the look you want in your application, you can select the style.

Many tables in applications have only one section, so this method can usually be implemented with something such as the following:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger) section {
    return 5;
}
```

This code specifies five cells for every section that the table view has. To define how many sections a table view has, you'll need to implement an optional method. If that method isn't defined in the <UITableViewDataSource> protocol, the table view defaults to having only one section:

```
- (UITableViewCell*)tableView:(UITableView*)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

The other required method for a UITableViewDataSource to implement is the above method. It's called as many times as there are cells. Each time it's called, the data source is responsible for returning a UITableViewCell object or subclass. UITableViewCell is a subclass of UIView and is the actual object that the user sees in the table view.

Much like the table view itself, Apple provides a standard UITableViewCell with a few basic configurations:

- UITableViewCellStyleDefault

This is the default style for UITableViewCells. It's a cell view with a black, left-aligned text label. There's also an optional left-aligned image view.

- `UITableViewCellStyleValue1`
This type of cell has a black, left-aligned label and a blue, right-aligned label. This style is used commonly around the iPhone OS, specifically in the Settings application.
- `UITableViewCellStyleValue2`
This type of cell is similar to a `UITableViewCellStyleValue1`: a black, left-aligned label and a blue, right-aligned label. In addition, this view provides a smaller gray, left-aligned text field under the primary text field. These cells are seen in the Phone application.
- `UITableViewCellStyleSubtitle`
This type of cell is similar to a `UITableViewCellStyleValue2`, a black, left-aligned label and a smaller gray, left-aligned text field under the primary text field. The blue, right-aligned label isn't available in this view. These cells are seen in the iPod application.

While many applications use customized subclasses of `UITableViewCell`, the standard cells work well in many cases. The structure for the `tableView:cellForRowAtIndexPath:` is similar for all applications. Apple provides special cell-view retrieval methods to help with memory management when implementing this method. Let's look at the general structure of this method's implementation, shown in the following listing.

Listing 7.8 Returning table view cells for a table view

```

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell;
    NSString *reuseID = @"cellID";

    cell = [tableView dequeueReusableCellWithIdentifier:reuseID];

    if(cell == nil) {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:reuseID] autorelease];

    NSLog(@"Made a new cell view");
    }

    [[cell.textLabel] setText:@"Hello World!"];

    return cell;
}

```

**Initialize
tableView cell** ①
 ←
True if statement ②
 ←
**If true, the cell
is executed** ③

The most important parts of this code happen on three consecutive lines. The first part ① attempts to initialize the cell that's being requested from the `tableView` by using the `dequeueReusableCellWithIdentifier` method. This method looks for a cell that has scrolled off the screen and attempts to reuse it. If it finds a cell that can be recycled, the cell is initialized; otherwise, the `if` statement ② returns `true` and ③ is

executed. This is the manual instantiation of the cell, which is used only if there's no cell to recycle that has recently been scrolled offscreen.

This coding strategy is used in several other places in the iPhone SDK. To show that the view allocation is working, you make a project with a table view and implement `dequeueReusableCellWithIdentifier` just as you see right above. You fill in `tableView:numberOfRowsInSection` with the following:

```
return 100;
```

You can see the resulting application and its terminal output using the following code:

```
Insert UITableViewDataSourceTerminalOutput.png and
    UITableViewDataSourceProtocolApp
```

You have 100 cells that you can scroll through here, with 11 active on the screen at any one time. This means that at the minimum you must create 11 `UITableViewCell`s. When you look at your terminal output, you can see that 12 `UITableViewCell`s were created. This is efficient and ensures that as little memory as possible is used to allow the user to scroll through all 100 cells. This technique is used for memory management with the `<MKMapViewDelegate>` protocol as well. Whether you're using a `UITableViewCell` standard class or a custom subclass, this should be your approach to deliver the cells to the view.

<UITableViewDataSource> OPTIONAL METHODS

The following method is used when a table view is to have more than one section:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
```

When this method isn't implemented, the method defaults to one. One thing to note is that the iPhone OS uses a zero-based count for section and row numbers, so the first element of the first section of a table view will have an index path like the following:

```
NSIndexPath.row = 0
NSIndexPath.section = 0
```

The following code is used to set the title of the header for sections when using default table view headers with the `UITableViewStyleDefault`:

```
- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section
```

Every section begins with a header and can also have a footer at the end. If the following method isn't defined, there'll be only headers, no footers, to separate your sections:

```
- (NSString *)tableView:(UITableView *)tableView
    titleForFooterInSection:(NSInteger)section
```

7.3.2 <UITableViewDelegate>

Unlike the `<UITableViewDataSource>`, the `<UITableViewDelegate>` has no required methods. All the methods in this protocol fall into the following functions:

- Setting information such as cell height and indentation for the entire table view
- Being informed when actions occur to the table view

Let's look at some of the most common functions used in the delegate.

<UITableViewDelegate> SETTER METHODS

The following method is commonly used when doing a modification to the standard table view cells:

```
- (CGFloat)tableView:(UITableView *)tableView
    heightForRowAtIndexPath:(NSIndexPath *)indexPath
```

Sometimes, with a thumbnail image or when you want to use bigger text, you need to make the height of table view cells larger. Many developers waste time trying to supply larger table view cells but get frustrated when the cells don't display properly. You must implement the `heightForRowAtIndexPath:` method if you want cells to be larger than the default 44 pixels. This height can also be modified through Interface Builder in Xcode 4, but a best practice is to modify the height using this delegate method:

```
- (NSInteger)tableView:(UITableView *)tableView
    indentationLevelForRowAtIndexPath:(NSIndexPath *)indexPath
```

This method can modify the entire horizontal alignment of a returned table view cell. Setting the indentation level moves the view content over about 11 pixels per indentation. For example, if you set the indentation level of a cell to three, the content of the cell is moved over approximately 33 pixels. Using the same code from the `UITableViewDataSource` section, implement this method for the table with the code shown in the following listing.

Listing 7.9 Setting an indentation level for a table view

```
- (NSInteger)tableView:(UITableView *)tableView
    indentationLevelForRowAtIndexPath:(NSIndexPath *)indexPath
{
    int val = indexPath.row % 56;
    if(val < 28)
        return indexPath.row;
    else
        return 56 - val;
}
```

This listing shows what incrementing the indentation level for each cell will do. When the indentation level reaches 28, the indentation level is decremented in order to keep the content in the view. You can see an example of what this table view looks like now in figure 7.1.

These methods are required to be implemented together:

```
- (CGFloat)tableView:(UITableView *)tableView
    heightForHeaderInSection:(NSInteger)section
- (UIView *)tableView:(UITableView *)tableView
    viewForHeaderInSection:(NSInteger)section
```

Any table view (grouped or standard) may have a header view. This view can be anything: a single `UIView`, `UIImageView`, or `UILabel`, for instance, or a more complex custom `UIView` subclass. Using the header view delegate method saves a number of lines of code required to customize the headers, and generally is a better practice than attempting to implement a header yourself. If you have a reason to add some type of custom view before your table data is shown, these are the methods you should implement:

```
- (CGFloat)tableView:(UITableView *)tableView
  heightForFooterInSection:
    (NSInteger)section

- (UIView *)tableView:(UITableView
  *)tableView
  viewForFooterInSection:(NSInteger)section
```

This set of methods is identical to the header view methods and likewise must be implemented together. Once again, this is the best practice when you need to add some custom content after your table view section data is presented.

<UITableViewDelegate> ACTION METHODS

Most methods that a class conforming to the `<UITableViewDelegate>` can implement are methods that respond to events that occur in the table view. These methods will inform you of things like cells being added, cells being removed, cells being modified, cells being selected, and a few more types of events that can occur:

```
- (void)tableView:(UITableView *)tableView
  accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
```

Apple provides some standard-style accessory views for `UITableViewCell`s. Accessory views are placed on the right-hand side of a `UITableViewCell`. They're distinct parts of a cell that can be hidden or modified when a table view is being edited. `UITableViewCellAccessoryTypes` come in four default flavors:

- `UITableViewCellAccessoryNone`

This method leaves the accessory view of the `UITableViewCell` blank. This is the default set for `UITableViewCell`s.

- `UITableViewCellAccessoryDisclosureIndicator`

This method leaves the accessory view of the `UITableViewCell` as a bold greater-than sign.

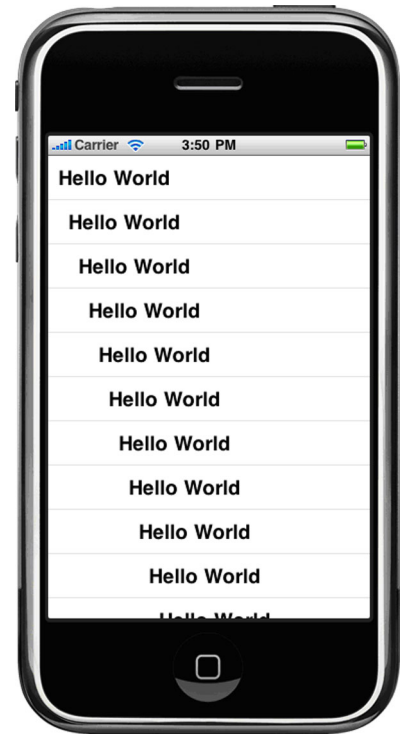


Figure 7.1 Setting an indentation level for a table view

- `UITableViewCellAccessoryDetailDisclosureButton`
This method leaves the accessory view of the `UITableViewCell` as an open right-bracket image in a blue background.
- `UITableViewCellAccessoryCheckmark`
This method leaves the accessory view of the `UITableViewCell` as a checkmark image.

In order for this delegate method to be called, your table view cell must have an accessory for the user to tap. The accessory type of a cell is set during the creation of the cell itself, usually in the `tableView:cellForRowAtIndexPath:`. `UITableViewCell` has two methods for setting an accessory view:

- `setAccessoryType:`
This method takes a `UITableViewCellAccessoryType` parameter. Any other object passed into this method will result in an error.
- `setAccessoryView:`
If you want to have some other type of custom accessory presented in the accessory view, use this method. It takes in a general `UIView` or any subclass of it. This method is useful when you're trying to style an application that doesn't work with the default accessory types provided by Apple. Be careful when providing `UIButton`s here: you can get a single button firing off to two separate methods when it's tapped.

The following methods are similar to the `tableView:accessoryButtonTappedForRowWithIndexPath:` except they're called when any other section of a table view cell is selected. Two points of entry are provided for the selection of a cell, `willSelect` and `didSelect`. This is commonly done throughout other Apple-provided delegates. Multiple methods like these exist to allow developers the ability to react to users at any point during the application flow. `willSelect` is called before the `didSelect`, but a call to `willSelect` is always followed by a call to `didSelect`:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
  willSelectRowAtIndexPath:(NSIndexPath *)indexPath

- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath
```

Just as there's an entry point to react to the user selecting any given cell, the following methods provide points at which to react when a cell is being deselected:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
  willDeselectRowAtIndexPath:(NSIndexPath *)indexPath

- (void)tableView:(UITableView *)tableView
  didDeselectRowAtIndexPath:(NSIndexPath *)indexPath
```

It's common for table views to have alternating background colors, as can be seen in the App Store and iTunes Music Store. Some developers might think the background colors should be set when they create the cell, either in the `UITableViewCell` subclass

or in `tableView:cellForRowAtIndexPath:`, but that isn't the case. When the `UITableView` is delivered a `UITableViewCell`, Apple modifies certain parts of the cell to make it fit nicely into the `UITableView`. When the table view is completely done doing what it needs to do with the cell, it passes the cell through the following method for any final modifications. After this method finishes, the cell is put into the table view and displayed to the user:

```
- (void)tableView:(UITableView *)tableView
    willDisplayCell:(UITableViewCell *)cell
    forRowAtIndexPath:(NSIndexPath *)indexPath
```

7.3.3 <UIActionSheetDelegate>

The `UIActionSheet` is a great interface element provided by Apple. It's used either to inform the user of an alert that takes over an entire application or to provide a prompt that a process is being conducted for the user. Examples of this include choosing a language for an application's content and informing a user that an application is uploading some kind of file. Action sheets have a protocol associated with them to inform the presenting view controller of their activity. Let's look at the methods that compose the `<UIActionSheetDelegate>`.

The following method is called when a button from the `UIActionSheet` is pushed. After this method executes, the `UIActionSheet` is automatically dismissed. Here you respond appropriately to whatever button was selected but don't have to worry about returning anything or dismissing the action sheet. That's all taken care of automatically:

```
- (void)actionSheet:(UIActionSheet *)actionSheet
    clickedButtonAtIndex:(NSInteger)buttonIndex
```

The following methods are available, providing hooks into the application flow. `willPresentActionSheet:` is called just before the action sheet animation begins, and `didPresentActionSheet:` is called as the animation finishes. These different access points allow developers to get the kind of responsiveness that they're shooting for:

```
- (void)willPresentActionSheet:(UIActionSheet *)actionSheet
- (void)didPresentActionSheet:(UIActionSheet *)actionSheet
```

Like the presentation methods, the following methods provide hooks into the action sheet when it's being dismissed. The `actionSheet:willDismissWithButtonIndex:` is called just before the action sheet dismissal animation begins, and `actionSheet:didDismissWithButtonIndex:` is called as the dismissal animation finishes:

```
- (void)actionSheet:(UIActionSheet *)actionSheet
    willDismissWithButtonIndex:(NSInteger)buttonIndex
- (void)actionSheet:(UIActionSheet *)actionSheet
    didDismissWithButtonIndex:(NSInteger)buttonIndex
```

The following is the final method you need to implement when creating an action sheet, and it might not be necessary every time you make one. Action sheets, when they're being created, allow developers to set a "destructive button title." If a developer

sets a title here, this button fits the `actionSheetCancel:` method. It's commonly red (you can see it in the contacts section of the iPhone), and is used to confirm actions such as deleting data or logging off:

```
- (void)actionSheetCancel:(UIActionSheet *)actionSheet
```

7.3.4 *NSXMLParser*

The final protocol we review differs slightly from the protocols we already covered. Apple provides a class to iPhone developers called `NSXMLParser`. Developers use this class when parsing XML. It should be noted that several open source alternatives to `NSXMLParser` are available and used by many developers, but for the sake of consistency, we look at the delegate methods of the standard Cocoa XML Parser.

There is no `<NSXMLParser>` protocol; you'll receive no warning if you don't declare this in the header of the application you're creating. `NSXMLParser` is a fringe design class that follows the principles of protocol design but doesn't explicitly define a protocol. An `NSXMLParser` object, like the other objects, has a parameter called `delegate` that must be defined. Any object defined as the delegate has the option of implementing and collecting 20 different delegate methods. `NSXMLParser` provides delegate methods that handle every point of parsing for both XML- and Document Type Definition (DTD)-based documents.

XML is a type of file that can hold data in a very structured manner. As a quick introduction, XML uses syntax similar to HTML in order to create unique data structures. An example of an XML element to describe a person is shown in the following listing.

Listing 7.10 An author in XML

```
<Author>
  <name>Collin Ruffenach</name>
  <age>23</age>
  <gender>male</gender>
  <Books>
    <Book>
      <title>Objective C for the iPhone</title>
      <year>2010</year>
      <level>intermediate</level>
    </Book>
  </Books>
</Author>
```

XML is a common means of getting data from online sources such as Twitter. XML is also used to facilitate the data required to run your specific iPhone project. iOS development relies heavily on plist files. These files are really just XML and are used by Xcode to get things like your icon name and other application data. Xcode handles the setup of these values.

DTD is a document that describes the structure of the XML that you're working with. The DTD for the XML in listing 7.10 would be the following:

```
<!ELEMENT Author (name, age, gender, books_list(book*))>
<!ELEMENT name (#PCDATA)>
```

```

<!ELEMENT age (#PCDATA)>
<!ELEMENT gender (#PCDATA)>
<!ELEMENT Book (title, year, level)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT level (#PCDATA)>

```

For some applications, examining the structure of the XML they're receiving will change the manner in which the application parses. In this case, you say that the XML contains an element called `Author`. An `Author` is defined by a name, age, and gender, which are simple strings. An author also has a list of `Book` elements. A `Book` is defined by a title, year, and level, which are all simple strings. This analysis ensures that the `NSXMLParser` knows what to do.

Most of the time when you parse XML, you'll be aware of its structure when writing your parser class and won't need to investigate the XML feed's DTD. An example of this would be the Twitter XML feed for a timeline. You'll assume you know the XML structure for your XML and only implement the parsing functions of the `NSXMLParser` delegate to parse the `Author` XML you already looked at.

PARSING AN AUTHOR WITH NSXMLPARSER DELEGATE

The first step when implementing `NSXMLParser` is to create a class that contains the parser object and implement its delegate methods. Let's create a new View-based project called `Parser_Project` and a new `NSObject` subclass called `Parser`. The only instance variables you declare for the `Parser` class are an `NSXMLParser` and an `NSMutableString` to help with some parsing logic you'll implement. Make the `Parser.h` file look like the following:

```

#import <Foundation/Foundation.h>

@interface Parser : NSObject <NSXMLParserDelegate> {
    NSXMLParser *parser;
    NSMutableString *element;
}

@end

```

You now need an XML file to parse. You can take the XML in listing 7.10 and place it in a regular text file. Save the file as `Sample.xml`, and add it to the project. This gives you a local XML file to parse.

Now you need to fill in `Parser.m`, which will contain an `init` method and the implementation of the three most common `NSXMLParser` Delegate methods. Let's start with the `init` method; add the code shown in the following listing into `Parser.m`.

Listing 7.11 Parser.m initializer

```

- (id)init {
    if ((self == [super init])) {
        parser = [[NSXMLParser alloc]
            initWithContentsOfURL:
                [NSURL URLWithString:[NSBundle mainBundle]

```

```

        pathForResource:@"Sample"
        ofType: @"xml"]]];
    [parser setDelegate:self];
    [parser parse];
}
return self;
}

```

Here you initialized your `NSXMLParser` parser using a file URL pointing to the `Sample.xml` file you imported into your project. `NSURL` is a large class with all sorts of initializers. In this case, you're telling the `NSXMLParser` you'll provide a path to a file URL, or a local resource. Then you tell the `NSXMLParser` that the class you're currently in is the delegate of the parser, and finally you tell it you're ready to parse by calling the `parse` method.

Once the `parse` method is called on `NSXMLParser`, the parser begins to call its delegate methods. The parser reads down an XML file much like Latin/English characters are read: left to right, top to bottom. While there are many delegate methods, we focus on three of them:

```

- (void)parser:(NSXMLParser *)parser
  didStartElement:(NSString *)elementName
 namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName
  attributes:(NSDictionary *)attributeDict

```

`parser:didStartElement:namespaceURI:qualifiedName:attributes:` has a lot of parameters passed into it, but it's quite simple for your purposes. This method is called when an element is seen starting. This means any element (between `<>`) that doesn't have a `/`. In this method you first print the element you see starting, and then clear the `NSMutableString` element. You'll see as you implement the next few methods that the `element` variable is used as a string that you add to as delegate methods are called. The `element` variable is meant to hold the value of only one XML element, so when a new element is started, you must be sure to clear the `element` variable. Use the code shown in the following listing for this delegate method.

Listing 7.12 `NSXMLParser` methods

```

- (void)parser:(NSXMLParser *)parser
  didStartElement:(NSString *)elementName
 namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName
  attributes:(NSDictionary *)attributeDict {
    NSLog(@"Started Element %@", elementName);
    element = [NSMutableString string];
}

```

This method is called when an XML element is seen ending, which is indicated by a `/` preceding an element tag name in the xml, such as `</person>`. When this method is called, the `NSMutableString`'s `element` variable is complete. Now print out the value, as seen in the following listing.

Listing 7.13 NSXMLParser methods

```

- (void)parser:(NSXMLParser *)parser
  didEndElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI
  qualifiedName:(NSString *)qName
{
    NSLog(@"Found an element named: %@ with a value of: %@",
          elementName, element);
}

```

This method is called when the parser sees anything between an element's beginning and end. You use this entry point as a way to collect all the characters that are between an XML element beginning and ending: you call the `appendString` method on the `NSMutableString` (listing 7.14). By doing this every time the `parser:foundCharacters:` method is called, by the time the `parser:didEndElement` method is called, the `NSMutableString` will be complete. In this method, you first make sure you've initialized your `NSMutableString` element, and then you append the string you've provided, shown in the next listing.

Listing 7.14 NSXMLParser methods

```

- (void)parser:(NSXMLParser *)parser
  foundCharacters:(NSString *)string
{
    if (element == nil)
        element = [[NSMutableString alloc] init];
    [element appendString:string];
}

```

Now all that's left to do is create an instance of your parser and watch it go. Go to `Parser_ProjectAppDelegate.m` and add the code shown in the following listing into the already existing method.

Listing 7.15 Initializing the Parser

```

- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];

    Parser *parser = [[Parser alloc] init];

    return YES;
}

```

If you run the application and bring up the terminal window (Shift-Command-R), the output shown in the following listing should be generated.

Listing 7.16 Parser output

```

Parser_Project [57815:207] Started Element Author
Parser_Project [57815:207] Started Element name
Parser_Project [57815:207] Found an element named: name with a value of:
    Collin Ruffenach
Parser_Project [57815:207] Started Element age
Parser_Project [57815:207] Found an element named: age with a value of: 23
Parser_Project [57815:207] Started Element gender
Parser_Project [57815:207] Found an element named: gender with a value of:
    male
Parser_Project [57815:207] Started Element Books
Parser_Project [57815:207] Started Element Book
Parser_Project [57815:207] Started Element title
Parser_Project [57815:207] Found an element named: title with a value of:
    Objective C for the iPhone
Parser_Project [57815:207] Started Element year
Parser_Project [57815:207] Found an element named: year with a value of: 2010
Parser_Project [57815:207] Started Element level
Parser_Project [57815:207] Found an element named: level with a value of:
    intermediate
Parser_Project [57815:207] Found an element named: Book with a value of:
    intermediate
Parser_Project [57815:207] Found an element named: Books with a value of:
    intermediate
Parser_Project [57815:207] Found an element named: Author with a value of:
    intermediate

```

You can see that, using the `NSXMLParser` delegate methods, you successfully parsed all of the information in your XML file. From here you could create Objective-C objects to represent the XML and use it throughout your application. XML processing is a vital part of most applications that get their content from some kind of web source: Twitter clients, News clients, or YouTube.

7.4 **Summary**

Protocols are regularly seen when developing using UIKit and Core Foundation for the iPhone. They're one of the foundation design decisions for most of the classes Apple provides. With attentive coding, protocols can make your application efficient and error proof.

We looked at defining your own protocol for a variety of reasons, conforming to protocols, and some specific popular protocols from the iPhone SDK. These protocols are also great to use when creating code you want to reuse. Through a proper understanding and implementation of the protocol design method, you can ensure a well-designed application.

In chapter 8, we look at how you can define your classes to have dynamic typing. Dynamic typing is an important principle to apply when considering memory for your applications. It's used in many protocol implementations, and a good grasp of dynamic typing will further your understanding of protocol design.

Dynamic typing and runtime type information

This chapter covers

- Static versus dynamic typing
- Messaging
- Runtime type information and implementation

Objective-C is a class-based object system in which each object is an instance of a particular class. At runtime, each object is aware of its class by way of a pointer named `isa` that points to a `Class` object (take a look in the debugger window the next time you're stuck at a breakpoint, and you should see `isa` listed in the variable window). The class object describes the data requirements for an instance of the class and its behavior in the form of the instance methods it implements.

Objective-C is also an inherently dynamic environment. As many decisions as possible are delayed until runtime rather than made during compile time. For example:

- Objects are dynamically allocated (via class methods such as `alloc`).
- Objects are dynamically typed. Any object can be referenced by a variable of type `id`. The exact class of the object and therefore the particular methods it provides aren't determined until runtime, but this delay doesn't stop code that sends messages to those objects from being written.

- Messages are dynamically bound to methods. A runtime procedure matches the method selector in a message with a particular method implementation that “belongs to” the receiver object. The mapping can change or be modified dynamically to change the behavior of the application and to introduce aspects such as logging without recompilation.

These features give object-oriented programs a great deal of flexibility and power, but there’s a price to pay. To permit better compile-time type checking and to make code more self-documenting, you can also choose to more statically type your use of objects in Objective-C.

8.1 *Static vs. dynamic typing*

If you declare a variable with a particular object type, the variable can only hold an object of the specified type or one of its subclasses. The Objective-C compiler also issues warning if, from the static type information available to it, it appears that an object assigned to that variable won’t be able to respond to a particular message. As an example, considering the following code snippet:

```
CTRentalProperty *house = ...;
[house setAddress:@"13 Adamson Crescent"];
[house setFlag:YES];
```

Because the variable is statically typed to be of type `CTRentalProperty`, the compiler can verify that it should respond to the `setAddress:` message. On checking the `CTRentalProperty` class, however, the compiler will issue a warning that the class may not respond to the `setFlag:` message, because according to the static type information available to the compiler, the `CTRentalProperty` class doesn’t provide such a message. Static type checks such as the one demonstrated here can be a helpful development aid because they enable errors such as typos or incorrect capitalization to be detected at compile time rather than runtime.

Static typing contrasts with the dynamic typing available when the `id` datatype is utilized, as in the following code sample:

```
id house = ...;
[house setAddress:@"13 Adamson Crescent"];
[house setFlag:YES];
```

Statically typed objects have the same internal data structure as objects declared to be `ids`. The type doesn’t affect the object. It affects only the amount of information given to the compiler about the object and the amount of information available to those reading the source code.

8.1.1 *Making assumptions about the runtime type*

It’s possible to have your cake and eat it too. By this we mean that you can have some of the benefits of the flexibility associated with dynamic typing while having greater compile-time checks approaching the level of those offered by fully static typing.

If you type a variable or return value as an `id` on either side of an assignment operator, no type checking is performed. A statically typed object can be freely assigned to an `id`, or an `id` to a statically typed object. For example, no type checking is performed by the compiler in either of the following two variable assignments:

```
CTRentalProperty *myProperty1 = [someArray objectAtIndex:3];
id myProperty1 = [[CTRentalProperty alloc] initWithRandomData];
```

In the first assignment, `NSArray`'s `objectAtIndex:` message is typed to return an `id` (because an array can hold an object of any type). This means the compiler is happy to allow the value to be assigned to a variable of type `CTRentalProperty`. No error will occur, even if the array is storing an object of an incompatible type (such as `NSNumber` or `NSString`) at array index 3. Those kinds of errors will be detected only when you attempt to access the object later.

For similar reasons, because methods such as `alloc` and `init` return values of type `id`, the compiler doesn't ensure that a compatible object is assigned to a statically typed variable. The following code is error prone and will compile without a warning:

```
CTRentalProperty *house = [NSArray array];
```

Obviously an `NSArray` instance isn't a valid subclass of the `CTRentalProperty` class, and apart from `NSObject`, they share no common inheritance.

A middle ground between the fully dynamic nature of the `id` datatype and a fully static variable declaration is to constrain an `id` variable to any object that implements a specific protocol. This can be done by appending the protocol name to the `id` datatype via a pair of angle brackets, as follows:

```
id<SomeProtocol> house;
```

In this variable declaration, `house` can store an instance of any class, but the class must implement the `SomeProtocol` protocol. If you try to assign an object to a `house` variable that doesn't implement the `SomeProtocol` protocol, the Objective-C compiler will warn during compilation that the object doesn't appear to be a suitable match.

A variable or argument datatype generally can be declared in three different ways:

```
id house;
id<SomeProtocol> house;
CTRentalProperty *house;
```

All three statements provide identical behavior at runtime, but the additional information progressively provided to the compiler enables the compiler to warn the programmer of type incompatibilities.

It's unusual to need to declare an object of type `id`. Generally, you'll know the type of object you're expecting, or at least expect it to provide a certain set of functionality, best described by an `id<Protocol>` style data type.

8.2 *Dynamic binding*

Objective-C, like Smalltalk, can use dynamic typing; an object can be sent a message that isn't specified in its interface. Flexibility is increased because an object can “capture” a message and pass the message along to a different object that can respond to the message appropriately. This behavior is known as *message forwarding* or *delegation*. Alternatively, an error handler can be used in case the message can't be forwarded. If an object doesn't forward a message, respond to it, or handle an error, the message is silently discarded.

Statically typed objects are still dynamically bound

Messages sent to statically typed objects are dynamically bound, just as objects typed `id` are. The exact type of a statically typed receiver is still determined at runtime as part of the messaging process. Here is a display message sent to `thisObject`:

```
Rectangle *thisObject = [[Square alloc] init];
[thisObject display];
```

This code performs the version of the method defined in the `Square` class, not the one in its `Rectangle` superclass.

8.3 *How messaging works*

A lot of introductory texts about Objective-C make a big deal that messages are sent to objects rather than methods being invoked on an object, but what does this really mean, and how does the distinction affect how you go about developing your applications?

If an Objective-C compiler sees a message expression such as the following

```
[myRentalProperty setAddress:@"13 Adamson Crescent"];
```

it doesn't determine which method in the `CTRentalProperty` class implements the `setAddress:` message and hardcode a call to it. Instead, the Objective-C compiler converts the message expression into a call to a messaging function called `objc_msgSend`. This C-style function takes the receiver (object to receive the message) and the name of the message (the method selector) as its first two arguments followed, conceptually at least, by the rest of the arguments specified in the message send, as follows:

```
objc_msgSend(
    myRentalProperty,
    @selector(setAddress:),
    @"13 Adamson Crescent"
);
```

The message send function does everything required to implement dynamic binding. It uses the message name and receiver object to determine the correct method implementation to invoke and then calls it before returning the value as its own return value.

Messages are slower than function calls

Message sending is somewhat slower than calling a function directly. Typically this overhead is insignificant compared to the amount of work performed in the method. The overhead is also reduced by some internal tricks in the implementation of `objc_msgSend`, which manages to reduce the amount of repetitive work performed in scenarios such as a message being sent to all objects in an array.

In the rare case that method sending causes an undue performance hit, it's possible to bypass several aspects of Objective-C's dynamism. Bypassing should be performed, however, only after analysis using tools such as Instruments.

Behind the scenes, an Objective-C method ends up being transformed and compiled as a straight C function, designed to be invoked via `objc_msgSend`. A method like this

```
- (void)setAddress:(NSString *)newAddress { ... }
```

can be imagined to be translated into a C function similar to the following:

```
void CTrentalProperty_setAddress(
    CTrentalProperty *self,
    SEL _cmd,
    NSString *str
)
{
    ...
}
```

The C-style function isn't given a name because it can't be accessed directly, so it isn't made visible to the linker. The C function expects two arguments in addition to those explicitly declared in the Objective-C method declaration. We previously discussed the `self` parameter, which is implicitly sent in a message send to indicate the object that received the message; the lesser-known implicit parameter is named `_cmd`. This parameter holds the selector (and hence name) of the message that was sent in order to invoke this method implementation. Later in this chapter we discuss a couple of techniques whereby a single method implementation can in fact respond to more than one unique method selector.

8.3.1 Methods, selectors, and implementations

In the previous section, a number of new concepts were introduced: method, selector, implementation, message, and message send. What exactly are these terms, and how do they relate to source code?

- *Method*—A piece of code associated with a class and given a particular name, such as `(void)setAddress:(NSString *)newValue { ... }`.
- *Selector*—An efficient way to present the name of a message at runtime. Represented by the `SEL` data type.
- *Implementation*—A pointer to the machine code that implements a method. Represented by the `IMP` data type.

- *Message*—A message selector and set of arguments to be sent to an object, such as send "setAddress:" and "13 Adamson Crescent" to object 0x12345678.
- *Message send*—The process of taking a message, determining the appropriate method implementation to call, and invoking it.

A method selector is a C string that has been registered with the Objective-C runtime. Selectors generated by the compiler are automatically mapped by the runtime when the class is loaded.

For efficiency, the full C string name isn't used for method selectors in compiled code. Instead, the compiler stores each method selector name in a table and then pairs each name with a unique identifier that represents the method at runtime.

A method selector is represented in source code by the SEL data type. In source code you refer to a selector using the @selector(...) directive. Internally the compiler looks up the specified method name and replaces it with the more efficient selector value. As an example, you could determine the selector for a message named setAddress: as follows:

```
SEL selSetAddress = @selector(setAddress:);
```

Although it's more efficient to determine method selectors at compile time with the @selector directive, you may encounter the need to convert a string containing a message name into a selector. For this purpose, there is a function called NSSelectorFromString:

```
SEL selSetAddress = NSSelectorFromString(@"setAddress:");
```

Because the selector is specified as a string, it can be dynamically generated or even specified by the user.

Compiled selectors identify method names, not method implementations. The display method for one class, for example, has the same selector as display methods defined in other classes. This is essential for polymorphism and dynamic binding; it lets you send the same message to receivers belonging to different classes. If there were one selector per method implementation, a message would be no different from a direct function call.

Given a method selector, it's also possible to convert it back into a text string. This is a handy debugging tip to remember. Because every Objective-C method is passed a hidden selector argument named _cmd, it's possible to determine the name of message that invoked the method, as follows:

```
- (void)setAddress:(NSString *)newAddress {
    NSLog(@"We are within the %@ method", NSStringFromSelector(_cmd));
    ...
}
```

The NSStringFromSelector method takes a selector and returns its string representation, resulting in the following log message:

```
We are within the setAddress: method
```

8.3.2 Handling unknown selectors

Because method binding occurs at runtime, it's possible for an object to receive a message to which it doesn't know how to respond. This situation is an error, but before announcing the error, the runtime system gives the receiving object a second chance to handle the message. The process is commonly called *message forwarding* and can be used to easily implement certain design patterns such as the Decorator or Proxy patterns.

When Objective-C can't determine via class metadata which method to invoke for a message selector, it first checks to see if the class wants the entire message sent unchanged to another object. It attempts to invoke the `forwardingTargetForSelector:message`, which could be implemented by a class as follows:

```
- (id) forwardingTargetForSelector: (SEL) sel {
    return anotherObject;
}
```

With this implementation, any unknown message sent to the object is redirected, or forwarded, to the object represented by `anotherObject`. This makes the original object appear from the outside as though it combined the features of both objects.

Obviously the implementation of `forwardingTargetForSelector:` could become more complex. As an example, you could dynamically return different objects on the basis of the particular method selector passed in as an argument.

`forwardingTargetForSelector:` is commonly called the “fast-forwarding path,” because it quickly and easily causes a message to be forwarded to another object without requiring too much code to be developed (or executed during runtime). It does have limitations, however: for instance, it isn't possible to “rewrite” a message sent to redirect a message addressed to one selector to be processed by another; nor is it possible to “drop” a message and ignore it. Fortunately, a “normal forwarding path” goes into action whenever the fast path doesn't provide a suitable implementation for a missing method selector, and this feature provides more flexibility in how a message is handled.

If `forwardingTargetForSelector:` isn't implemented or fails to find a suitable target object (it returns `nil`), the runtime, before announcing an error, sends the object a `forwardInvocation:` message with an `NSInvocation` object as its sole argument.

An `NSInvocation` object encapsulates the message being sent. It contains details about the target, selector, and all arguments specified in a message send and allows full control over the return value.

Because your implementation of `forwardInvocation:` has full access to the `NSInvocation` object, it can inspect the selector and arguments, modify them, and potentially replace them, leading to a wide range of possibilities for altering the behavior.

NSObject's default implementation of `forwardInvocation`

Every object that inherits from `NSObject` has a default implementation of the `forwardInvocation:` message. But `NSObject`'s version of the method invokes `doesNotRecognizeSelector:`, which logs an error message.

As an example, the following implementation of `forwardInvocation:` manages to rewrite the name of a message:

```
- (void)forwardInvocation:(NSInvocation *)anInvocation {
    if (anInvocation.selector == @selector(fancyAddress:)) {
        anInvocation.selector = @selector(address:);
        [anInvocation invoke];
    } else {
        [super forwardInvocation:anInvocation];
    }
}
```

The `if` statement detects if the message being sent to the object uses the `fancyAddress:` selector. If it does, the selector is replaced with `address:` and the message is resent (or invoked):

```
NSString *addr = [myRentalProperty fancyAddress];
```

This message would be equivalent to the following:

```
NSString *addr = [myRentalProperty address];
```

One interesting thing to note when using `NSInvocation` is that you send the object an `invoke` message in order for it to dispatch the message send it encapsulates, but you don't get the message's return value. There's no return statement in the `forwardInvocation:` method implementation, yet it does return a value to the caller. While using `NSInvocation`, you must instead query the `NSInvocation`'s `returnValue` property, something the caller of `forwardInvocation:` does behind the scenes.

A `forwardInvocation:` method can redirect unrecognized messages, delivering them to alternative receivers, or it can translate one message into another or "swallow" some messages so there's no response and no error.

8.3.3 *Sending a message to nil*

How is it possible for a message to be swallowed with no resulting response or error? And what would the return value be in the case of messages that are expected to return a value? For answers, you need to look at another situation, that of sending a message with `nil` as the receiver object.

In C++ or other object-oriented languages, such as Java or C#, attempting to invoke a method on a `NULL` object reference is an error. In Objective-C parlance, it would be like attempting to send a message with a `nil` receiver. Unlike in other languages, though, in Objective-C this is commonly not an error and is completely safe and natural to perform. As an example, a C# developer moving to Objective-C may initially write the following source code:

```
NSString *someString = ...;

int numberOfCharacters = 0;
if (someString != nil)
    numberOfCharacters = [someString length];
```

The `if` statement is designed to avoid the `length` method call when `someString` is `nil`, indicating no string is available:

```
NSString *someString = ...;
int numberOfCharacters = [someString length];
```

If a message is sent to `nil`, or if a target object decides to completely ignore a message, the message in most cases will return the equivalent of zero. This means the code snippet just outlined would set `numberOfCharacters` to zero when you attempted to send the `length` message to the `someString` object.

8.4 Runtime type information

Objective-C provides rich type information at runtime about the objects and classes that live in your application. At its simplest, you can query an object to determine its type by sending it an `isKindOfClass:` message, as shown here:

```
id someObject = ...;
if ([someObject isKindOfClass:[CTRentalProperty class]]) {
    NSLog(@"This object is a kind of CTRentalProperty (or a subclass)");
}
```

The `isKindOfClass:` message returns `true` if the object is an instance of the specified class or is compatible with it (the object is of a subclass).

8.4.1 Determining if a message will respond to a message

If an object receives a message and determines it can't process it, an error typically results. In some scenarios, especially with protocols with `@optional` messages, it can be helpful to determine if a particular message send will successfully execute or result in an error.

Therefore, the `NSObject` class provides a message named `respondsToSelector:` that can be used to determine if an object will respond to (if it'll handle or process) a given selector. You can use the `respondsToSelector:` message as follows:

```
if ([anObject respondsToSelector:@selector(setAddress:)])
    [anObject setAddress:@"13 Adamson Crescent"];
else
    NSLog(@"This object does not respond to the setAddress: selector");
```

The `respondsToSelector:` test is especially important when sending messages to objects that you don't have control over at compile time, particularly if the object is typed as `id`. For example, if you write code that sends a message to an object represented by a variable that others can set, you should make sure the receiver implements a method that can respond to the message.

8.4.2 Sending a message generated at runtime

Sometimes you need to send a message to an object but don't know the name or set of arguments that must be included until runtime. `NSObject` comes to the rescue again by providing messages named `performSelector:`, `performSelector:withObject:`, and

`performSelector:withObject:withObject:..` All of these methods take a selector indicating the message to send and map directly to the messaging function. For example:

```
[house performSelector:@selector(setAddress:)
      withObject:@"13 Adamson Crescent"];
```

This line is equivalent to

```
[house setAddress:@"13 Adamson Crescent"];
```

The power of these methods comes from the fact that you can vary the message being sent at runtime just as easily as it's possible to vary the object that receives the message. A variable can be used in place of the method selector, unlike in a traditional message send, which accepts only a constant in this location. In other words, the following is valid:

```
id obj = get_object_from_somewhere();
SEL msg = get_selector_from_somewhere();
id argument = get_argument_from_somewhere();
[obj performSelector:msg withObject:argument];
```

But this isn't valid:

```
id obj = get_object_from_somewhere();
SEL msg = get_selector_from_somewhere();
id argument = get_argument_from_somewhere();
[obj msg:argument];
```

In this example, the receiver of the message (`obj`) is chosen at runtime, and the message it's sent is also determined at runtime. This leads to a lot of flexibility in the design of an application; with functions such as `NSSelectorFromString`, discussed previously, it's even possible to use selectors that exist only at runtime.

The Target-Action design pattern

The UIKit framework makes good use of the ability to programmatically vary both the receiver and message sent via a message send. `UIView` objects such as `UISlider` and `UIButton` interpret events from hardware such as the touch screen or Bluetooth-connected keyboard and convert these events into application-specific actions.

For instance, when the user taps on a button, a `UIButton` object sends a message instructing the application that it should do something in response to the button press. A `UIButton` object is initialized with details on which message to send and where to send it. This is a common design pattern in UIKit called the Target-Action design pattern.

As an example, you could request that a button send the `buttonPressed:` message to the `myObject` object when the user lifts their finger off the screen with the following initialization:

```
[btn addTarget:myObject
      action:@selector(buttonPressed:)
      forControlEvents:UIControlEventTouchUpInside];
```

The `UIButton` class sends the message using `NSObject`'s `performSelector:withObject:` method, just as we discussed in this section. If Objective-C didn't allow the message name or target to be varied programmatically, `UIButton` objects would all have to send the same message, and the name would be frozen in `UIButton`'s source code.

8.4.3 Adding new methods to a class at runtime

The Objective-C statement `[CTRentalProperty class]` returns a `Class` object that represents the `CTRentalProperty` class. With Objective-C's dynamic features, it's possible to add or remove methods, protocols, and properties from a class by interacting with this object at runtime.

We demonstrate how to dynamically add a new method implementation to an existing class by responding to the class's `resolveInstanceMethod:` message. This is another method invoked on an object when a message sent to it doesn't find a matching method implementation.

At the lowest level of the Objective-C runtime, an Objective-C method is a C function that takes, at a minimum, two additional arguments named `self` and `_cmd` (as discussed previously). Once a suitable function is developed, it can be registered as a method of an Objective-C class using a function called `class_addMethod:`. Add the following code to `CTRentalProperty.m` in the current version of the Rental Manager application:

```
void aSimpleDynamicMethodIMP(id self, SEL _cmd) {
    NSLog(@"You called a method named %@", NSStringFromSelector(_cmd));
}

+ (BOOL)resolveInstanceMethod:(SEL)sel {
    if (sel == @selector(aSimpleDynamicMethod)) {
        NSLog(@"Adding a method named %@ to class %@",
            NSStringFromSelector(sel),
            NSStringFromClass([self class]));
        class_addMethod([self class],
            sel,
            (IMP)aSimpleDynamicMethodIMP,
            "v@:");
        return YES;
    }

    return [super resolveInstanceMethod:sel];
}
```

In the current version of the Rental Manager application, you can then add the following calls to a convenient location, such as the `RootViewController`'s `viewDidLoad` method:

```
id house = [CTRentalProperty rentalPropertyOfType:TownHouse
    rentingFor:420.0f
    atAddress:@"13 Waverly Crescent, Sumner"];
[house aSimpleDynamicMethod];
[house aSimpleDynamicMethod];
[house aSimpleDynamicMethod];
[house aSimpleDynamicMethod];
[house aSimpleDynamicMethod];
```

When this code is executed, you should notice that the debug console lists the following content:

```
Adding a method named aSimpleDynamicMethod to class CTRentalProperty
You called a method named aSimpleDynamicMethod
```

```

You called a method named aSimpleDynamicMethod
You called a method named aSimpleDynamicMethod
You called a method named aSimpleDynamicMethod
You called a method named aSimpleDynamicMethod

```

The five messages that sent the `aSimpleDynamicMethod` message to the `CTRentalProperty` object all correctly invoked the `aSimpleDynamicMethod` function, as evidenced by the results of the `NSLog` call it contains.

An interesting point to notice is the first `NSLog` message, indicating that a method named `aSimpleDynamicMethod` was added to the `CTRentalProperty` class. This method results because the first time the house object receives the `aSimpleDynamicMethod` message, it detects that the object doesn't respond to this selector. Therefore, the Objective-C runtime automatically invokes the object's `resolveInstanceMethod:` method to see if it's possible to correct this problem. In the implementation of `resolveInstanceMethod:`, the Objective-C runtime function `class_addMethod` dynamically adds a method to the class and returns `YES`, indicating that you've resolved the missing method. The four future message sends, which send the `aSimpleDynamicMethod` selector, don't all produce equivalent log messages, because when these copies of the message are received, the Objective-C runtime finds that the object does have a method implementation for the `aSimpleDynamicMethod` selector, and hence it doesn't need to dynamically attempt to resolve the situation.

Forwarding methods and dynamic method resolution are largely orthogonal. A class has the opportunity to dynamically resolve a method before the forwarding mechanism kicks in. If `resolveInstanceMethod:` returns `YES`, message forwarding isn't actioned.

The dangerous practice of method swizzling

Because you can add new methods at runtime, you might be interested to learn that you can also replace an existing method with a new implementation at runtime, even without access to source code for the class in question.

The Objective-C runtime function `method_exchangeImplementations` accepts two methods and swaps them, so that calling one causes the implementation of the other one to be invoked.

This technique is commonly called *method swizzling*, but it's a particularly dangerous practice that usually indicates you're doing something wrong or something not intended by the original creators of the object in question. It's dangerous because it relies on so many undocumented or brittle aspects of a class's implementation.

8.5 Practical uses of runtime type introspection

The iOS development platform has matured with multiple versions of the iPhone operating system and variants for the iPad all exhibiting slightly different capabilities and restrictions.

It's now common to want to develop an application that can use features specific to a newer version of the iOS operating system yet still be able to run the application on a device with an older version of the operating system that doesn't provide the feature. Using a number of techniques discussed in this chapter, this goal can often be achieved.

The first technique to be aware of is the macro `UI_USER_INTERFACE_IDIOM`, which can be used as follows:

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    ... this code will only execute if the device is an iPad ...
}
```

The `UI_USER_INTERFACE_IDIOM` macro can be used to determine if the current device is an iPad or iPhone and to conditionally change the behavior of an application to make sure the look and feel integrates well with the slightly different usage model of both platforms.

In other cases, a class may exist in multiple versions of the iOS software development kit (SDK), yet have new properties or methods added to it. In these scenarios, you can use the `respondsToSelector:` message to determine if the device the application is currently running on provides the specified property or method.

As an example, the version of iOS that introduced multitasking capabilities introduced a property named `isMultitaskingSupported` on the `UIDevice` class to determine if the current device supports multitasking (devices such as the iPhone 3G can't multitask even though they can run the latest version of iOS). If, however, an iPhone owner hasn't updated the device to the newest iOS version, it's possible that attempting to access this property will cause an exception due to the class not responding to the selector. The workaround is as follows:

```
UIDevice *device = [UIDevice currentDevice];
BOOL backgroundSupported = NO;
if ([device respondsToSelector:@selector(isMultitaskingSupported)])
    backgroundSupported = device.multitaskingSupported;
```

In this scenario, the device is assumed not to support background processing, but if the `UIDevice` object indicates it responds to the `isMultitaskingSupported` property, you query it for the final answer.

In cases in which an entire class has been introduced in a new version of iOS, you can check for the availability of the class by using a code sample similar to the following:

```
if ([UIPrintInteractionController class]) {
    ... make use of the UIPrintInteractionController class ...
} else {
    ... do something if the UIPrintInteractionController isn't available ...
}
```

If the `UIPrintInteractionController` (or any other class placed in a similar code snippet) isn't available, the class message responds with `nil`, which evaluates to `false`. Otherwise, it returns a non-`nil` value and proceeds to execute the code in the first block of the `if` statement.

This process works only in applications compiled using version 4.2 of the iOS SDK (or later), because it requires a feature called *weak-linking support*. With code compiled with previous versions of the SDK, an application would fail to load on devices without the `UIPrintInteractionController` class, because part of the application startup process would be verification that all Objective-C classes referenced are available. With the weak-linking support in the iOS SDK4.2, it isn't an error to run an application on a device that's missing a required class, but it does mean the application developer can't rely on the application startup checks verifying everything is available in order for the application to run. Instead, they must implement runtime checks for any feature or class that may not be present across all devices.

The iOS SDK enables a similar feature with C-based APIs. For example, the iOS 3.2 SDK introduced the `CoreText.Framework`. If you want to use its `CTFontCreateWithName()` function in code that must also run on pre-iOS 3.2 devices, you must check for the function's availability before running the application. This can be achieved as follows:

```
if (CTFontCreateWithName != NULL)
    CTFontFontRef font = CTFontCreateWithName(@"Marker Felt", 14.0, NULL);
```

Because of the weak-linking feature of the iOS SDK, if an imported C function isn't available, the function pointer is set to an address of `NULL`. This means you can check whether or not a function exists by checking whether or not the function pointer is `NULL`. Weak linking is unlike a more traditional dynamic-linking scenario in which a missing imported function would result in the application refusing to load at all.

8.6 **Summary**

Objective-C is a mixture of both strongly statically and dynamically typed aspects. Its C-based origins are statically typed and bound, while the Objective-C object-oriented features are, at their heart, very dynamic and perform late, or dynamic, binding of message selectors to method implementations.

The ease of class introspection and modification at runtime enables a number of interesting programming techniques to be implemented with ease. Message forwarding in particular enables the creation of proxy or delay loading objects to be easily developed in a maintainable way that doesn't require modifications to the proxy class each time the proxied class is updated to include a new method or property. In the next chapter, we dive into memory management issues in Objective-C.

Memory management



This chapter covers

- Object lifetime management
- Manual memory management via `retain` and `release` messages
- Autorelease pools and the `autorelease` message
- Object ownership rules
- Memory management rules for custom classes

Many developers new to Objective-C find memory management one of the most difficult concepts to wrap their heads around. This is especially true for those coming from an environment such as Java, .NET, ActiveScript, or Ruby, where memory management is a much more automated, behind-the-scenes process.

In Objective-C all objects are created on the heap (unlike languages such as C++, which also allow objects to be placed on the stack). This means objects continue to consume memory until they're explicitly deallocated. They aren't cleaned up automatically when the code that uses them goes out of scope.

This feature has important connotations for how you program. If your application loses all references (pointers) to an object but doesn't deallocate it first, the

memory it consumes is wasted. This waste is typically called a *memory leak*, and if your application has too many memory leaks, it becomes sluggish or, in worst-case scenarios, crashes due to running out of available memory. On the other hand, if you deallocate an object too early and then reference it again, your application could crash or exhibit incorrect and random behavior.

Memory management isn't exactly tedious manual work, however. The `NSObject` base class provides support for determining how many other objects are interested in using a particular object and automatically deallocating the object if this count ever returns to zero. The difference between Objective-C and languages such as Java is that this support isn't automatic. You must adjust the count yourself as parts of your application use or lose interest in an object.

Quite a few of the problems encountered by novice Objective-C developers can be traced to incorrectly adjusting this count. Let's begin our coverage of Objective-C memory management by looking at exactly what this count represents.

9.1 **Object ownership**

Although most Objective-C memory management tutorials start off discussing the purpose of `NSObject`'s `retain` and `release` messages, it's important to step back and consider the general principles at play. The memory model in Objective-C is based on the abstract concept of object ownership—it just happens that the current implementation achieves these concepts via the use of reference counting.

Automatic garbage collection is available on some Objective-C platforms

Objective-C 2.0 introduced an optional garbage collection feature. You won't see an iPhone book discuss this in depth, though, because it's currently unavailable when targeting the iPhone or iPad platforms. The garbage collection features can be utilized only in applications designed to run on desktop Mac OS X devices, which makes discussion of how it works a rather moot point in an iPhone-focused book.

In Objective-C an owner of an object is simply something (a piece of code or another object) that has explicitly said the equivalent of, "I need this object; please make sure it doesn't get deallocated." This could be the code that created the object, or it could be another section of code that receives the object and requires its services. This concept is illustrated by figure 9.1 in which a rental property object has three "owners": an object representing the property owner, another representing the real estate agent attempting to sell the property, and a general contractor employed to repaint the exterior.

Each object in figure 9.1 has a vested interest in keeping the rental property object from being deallocated. As an owner of an object, each is responsible for informing the object when its services are no longer required. When an object, such as the rental property in figure 9.1, has no active owners (nobody needs it anymore), it can be automatically deallocated by the Objective-C runtime.

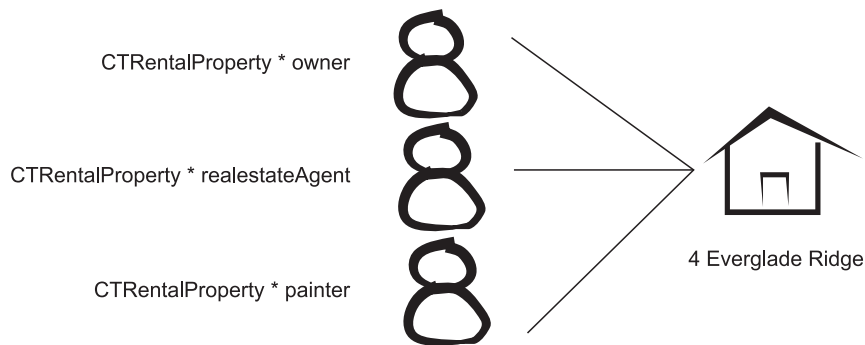


Figure 9.1 An example of an object having more than one owner. Ownership of (and responsibility for) the `CTRentalProperty` object representing 4 Everglade Ridge is shared among three instance variables.

In general, it's important to own an object before interacting with it. But it's possible to use an object without becoming its owner. In such cases it's important to ensure that something else maintains ownership of the object for the entire span of your interaction with it so it doesn't get deallocated before you're done using it.

9.2 Reference counting

Objective-C uses reference counting to implement the concept of object ownership. Every object keeps an internal count of how many times it has been owned (commonly called the object's *retain count*). Each time something puts its hand up and says, "I need this object," the retain count is incremented, and each time something says it no longer needs an object, the count is decremented.

If the counter is decremented back to zero, the object has no current owner and hence can be deallocated by the runtime.

Reference counting isn't another term for garbage collection

Although reference counting ensures objects that are in use aren't deallocated and automatically deallocates objects once they become unwanted, it isn't a form of automatic garbage collection.

In a garbage collection–based environment, a separate thread runs periodically in the background to determine which objects have become inaccessible (have no owners). This thread deallocates any object it comes across that can't be accessed by the application's source code. The programmer doesn't need to do anything manually to ensure this occurs and doesn't need to carefully maintain each object's retain count.

On the other hand, in a reference counting–based environment, the programmer must be careful to manually increment or decrement each object's retain count as code becomes interested or disinterested in the services of an object. Any mistake by the developer in handling the reference count for an object could lead to fatal application errors.

The advantages of a reference counting–based memory management scheme include the following:

- *Minimal overhead*—Relatively little runtime support is required to implement reference counting. This helps increase performance and reduce power consumption, both of which are important on a mobile device.
- *Deterministic behavior*—Because the developers are responsible for allocating and releasing objects, they also have explicit control over exactly when and where these events occur.

The potential disadvantages include the following:

- *Control implies responsibility*—There’s a greater risk of the developer making a mistake that will lead to memory leaks or random application crashes and malfunctions.
- *Additional overhead*—Including the developer in the memory management process means there’s yet another item for developers to consider while developing their applications rather than focusing on the core task of what makes their application unique.

Now that you’re familiar with the concepts involved in reference counting–based memory management, let’s look at how it’s implemented in Objective-C via the `retain`, `release`, and `autorelease` messages and how to avoid memory leaks or the dreaded "Program received signal: "EXC_BAD_ACCESS"." exception.

9.2.1 *Releasing an object*

When an owner of an object is finished using the object, it must tell the object that it wants to give up ownership. This process reduces the internal reference count of the object by one and is typically called *releasing an object* because the operation is performed by sending the object a message called `release`, as demonstrated here:

```
NSMutableString *msg =
    [[NSMutableString alloc] initWithString:@"Hello, world!"];
NSLog(@"The message is: %@", msg);
[msg release];
```

Because this code snippet creates the `NSMutableString` object (via the `alloc` message), it inherently owns the string and must explicitly give up ownership when the object becomes excess to requirements. This is achieved by the `release` message sent to the object on the last line.

In this example, the `release` message also indirectly causes the Objective-C runtime to send the object a `dealloc` message to destroy the object. This occurs because the `release` message is releasing the last (and only) owner of the object. The object’s retain count returns to zero when its last owner is released. If the object has more than one owner, a call to `release` has no visible effect on the application, other than to reduce the retain count, leaving the object with one less owner.

It’s important to note that once ownership of an object is given up, it’s generally not safe to refer to or utilize that object again from the current section of code, even if

the object is also owned, or kept alive, by another object. For example, the following code snippet, which simply rearranges some of the lines from the previous code snippet, has a major flaw:

```
NSMutableString *msg =
    [[NSMutableString alloc] initWithString:@"Hello, world!"];
[msg release];
NSLog(@"The message '%@' has %d characters", msg, [msg length]);
```

Because the third line releases ownership of the NSMutableString object, you can't utilize the object on the last line; the object will have become deallocated. Worse yet, if you execute this code snippet, it may occasionally work! Yet on another run of the application, it may crash or lead to an incorrect message being logged to the debug console. (You'll see in chapter 14 some tools that can help detect such errors.) Such errors typically don't appear as straightforward as demonstrated in the example code snippet and can be rather tricky to find.

One convention to make it easier to detect if a variable is referring to a usable object is to assign the value `nil` to a variable as soon as you release ownership. This is demonstrated next:

```
NSMutableString *msg =
    [[NSMutableString alloc] initWithString:@"Hello, world!"];
[msg release];
msg = nil;
NSLog(@"The message '%@' has %d characters", msg, [msg length]);
```

Assigning `nil` to a variable makes it easier to determine if it's currently pointing to a valid object. As you may remember from previous chapters, `nil` indicates the absence of a value and hence can never be a valid pointer value. For reasons covered in chapter 8, the last line is also perfectly safe to execute even though it attempts to send the `length` message to `nil`. Unlike in other languages, in which attempting to access or refer a variable set to `nil` would crash or generate a `NULL` reference exception, in Objective-C this construct is perfectly valid and has predictable results.

9.2.2 Retaining an object

The opposite of releasing ownership of an object is to claim ownership. In Objective-C this is performed by sending the object a `retain` message. This message essentially says, "I would like to make use of this object; please don't deallocate it until I have completed using it." Internally this causes the object to simply increment its retain count. This means it's important to match calls to `retain` with an equal number of calls to `release`.

Be careful with your bookkeeping

If you don't balance the number of `retain` and `release` messages sent to an object, the object will either be deallocated too early (in the case of too many `release` messages) or not be released at all (in the case of too few `releases` or additional `retains`). With manual reference counting, you as the developer must keep track of ownership requirements of the objects you use and when they need to be retained or released.

As an example, the following listing explicitly sends the `msg` object two additional retain messages, so three release messages must be sent in order to ensure the object eventually returns to a retain count of zero.

Listing 9.1 Correctly matching calls to retain and release

```
NSMutableString *msg =
    [[NSMutableString alloc] initWithString:@"Hello, world!"];
[msg retain];
[msg retain];
NSLog(@"The message '%@' has %d characters", msg, [msg length]);
[msg release];
[msg release];
[msg release];
```

At first glance, the code in listing 9.1 may appear incorrect. The `msg` object is sent three release messages, yet there are only two calls to `retain`. The calls seem unbalanced. But remember from our previous discussions that a call to `alloc` (as shown on the first line) implicitly makes you the owner of the created object. You can consider `alloc` as making an internal call to `retain` on your behalf, and as such, it must be matched with a call to `release`.

9.2.3 Determining the current retain count

For diagnostic purposes, it may be desirable to query an object to determine how many owners it currently has. This can be done by sending it a `retainCount` message, as demonstrated by the following listing. This listing is the same as listing 9.1 except for additional calls to `NSLog`, which log out the current retain count after each statement is executed.

Listing 9.2 Querying the retain count of an object during its lifecycle

```
NSMutableString *msg = [[NSMutableString alloc]
    initWithString: @"Hello, world!"];
NSLog(@"After alloc: retain count is %d", [msg retainCount]);

[msg retain];
NSLog(@"After retain: retain count is %d", [msg retainCount]);

NSLog(@"The message '%@' has %d characters", msg, [msg length]);

[msg release];
NSLog(@"After 1st release: retain count is %d", [msg retainCount]);

[msg release];
// NSLog(@"After 2nd release: retain count is %d", [msg retainCount]);
```

Notice that the log statements report the same values as those manually placed in listing 9.1 and that the last call to `NSLog` is commented out. This is because the last release message caused the object to return to a retain count of zero and hence become deallocated. Once the object is deallocated, it's impossible to communicate with it.

Keep yourself honest

If you regularly find yourself querying objects for their retain count, you should take a second look at what you're doing. Typically there's no reason to be interested in an object's actual retain count, and the result can often be misleading because you may be unaware of other objects that may have also retained the object.

In debugging memory management issues, you should be concerned only with ensuring that your code adheres to the ownership rules and that any calls you make to retain are matched with calls to release.

If you find yourself needing to write code similar to the following to force an object to become deallocated, look for mismatched calls to `retain` and `release` or a misunderstanding of memory management rules rather than brute-forcing a solution:

```
int count = [anObject retainCount];
for (int i = 0; i < count; i++)
    [anObject release];
```

As you experiment with querying retain counts, you may notice that some objects return interesting results. For example, based on your current knowledge, you may expect the following code snippet to print the message "Retain count is 1".

```
NSString *str = @"Hello, world!";
NSLog(@"Retain count is %d", [str retainCount]);
```

If you run this snippet, you'll notice that the retain count is reported as 2147483647. That's a lot of owners interested in keeping this seemingly unimportant string alive! This example highlights a special case. A constant string, such as one introduced via the `@". . ."` construct, is hardcoded into your application's executable. A retain count of 2147483647 (0x7FFFFFFF in hexadecimal) indicates that an object doesn't participate in standard reference counting behavior. If you send such an object a retain or release message and query its `retainCount` before and after, you'll notice it continues to report the same value.

Always consider things could change in the future

Just because an object doesn't do anything obvious when sent a `retain` or `release` doesn't mean that you shouldn't observe standard `retain` and `release` practices when using it.

If you write code that's arbitrarily provided an object, you generally won't know if it's been allocated in such a way that it requires standard memory management or was created in such a manner that improper management of the retain count may be less critical. It's generally easier to treat all objects as if they follow the `retain` and `release` model. It's better safe than sorry—you never know how you might update or alter your application in the future, potentially invalidating assumptions about how a particular object may behave.

Now that you have a solid understanding of the retain and release messages, you can apply that knowledge to virtually any Objective-C or Cocoa Touch memory management technique. These two messages are the fundamental building blocks upon which all other techniques are ultimately placed.

This isn't to say that retain and release are the perfect memory management tools. Both messages have deficiencies or scenarios under which their usage can be painful. As an example, manual management of an object's retain count can get laborious if an object is constantly passed among numerous owners. The next memory management technique we investigate builds on top of the retain and release foundations and is designed to ease such burdens.

9.3 *Autorelease pools*

The retain/release model has difficulties when ownership of an object needs to be transferred or handed off to something else. The code performing the transfer doesn't want to continue retaining the object, but neither do you want the object to be destroyed before the handoff can take place.

Consider the following method, which attempts to create a new NSString object and return it to the caller:

```
- (NSString *)CreateMessageForPerson:(NSString *)name {
    NSString *msg = [[NSString alloc] initWithFormat:@"Hello, %@", name];
    return msg;
}
```

The first line of the CreateMessageForPerson: method allocates a new string object. Since the string is created via the alloc message, the method currently owns the created string. Once the method returns, however, it'll never need to refer to the string again, so before returning, it should release its ownership.

A tempting solution may be to rework the CreateMessageForPerson method as follows:

```
- (NSString *)CreateMessageForPerson:(NSString *)name {
    NSString *msg = [[NSString alloc] initWithFormat:@"Hello, %@", name];
    [msg release];
    return msg;
}
```

This solution, however, has the same bug as previously discussed. The first line creates a new string object with a retain count of 1, and the next decrements the retain count back to zero, causing the string to be deallocated. The pointer to the deallocated object is then returned.

The problem with the retain/release model in this solution is that there's no suitable location to place the call to release.

If the call to release occurs in the CreateMessageForPerson method, you can't pass the string back to the caller. On the other hand, if you don't release the object in the method, you push the responsibility of freeing the string object onto every caller. This not only is error prone but breaks the principle of "match every retain with a release."

Ideally, you'd have a way to signal to Objective-C that you're ready for ownership of an object to be released, but instead of relinquishing ownership immediately, you prefer it to be performed at some point in the future, once the caller has had a chance to claim ownership for itself. The answer to this scenario is called an *autorelease pool*.

To use an autorelease pool, you simply send the object an autorelease message instead of the more immediate release message. This technique is demonstrated by the following version of the `CreateMessageForPerson` method:

```
- (NSString *)CreateMessageForPerson:(NSString *)name
{
    NSString *msg = [[NSString alloc] initWithFormat:@"Hello, %@", name];
    [msg autorelease];
    return msg;
}
```

With the string added to the autorelease pool, you've relinquished ownership of the string, yet it's safe to continue referring to and accessing it until some point in the future when it'll be sent a more traditional release message on your behalf. But what exactly is an autorelease pool, and when will the string be deallocated?

9.3.1 What is an autorelease pool?

An autorelease pool is simply an instance of the `NSAutoreleasePool` class. As your application runs, ownership of objects can be transferred to the most current autorelease pool. When an object is added to an autorelease pool, it takes over ownership of the object and sends the object a release message only when the pool itself is deallocated.

This can be handy when you want to create a bunch of temporary objects. Instead of manually keeping track of each retain and release message you send, you can simply use the objects, safe in the knowledge that they will all eventually be released.

9.3.2 Adding objects to the autorelease pool

Adding an object to the current autorelease pool is easy. You simply send the object an autorelease message instead of a release:

```
[myObject autorelease];
```

This causes the current autorelease pool to retain the object after relinquishing the current owner of its ownership responsibilities.

9.3.3 Creating a new autorelease pool

In general, it's possible to use the services of an autorelease pool without having to explicitly create one. This is because Cocoa Touch creates pools behind the scenes for your use. But it's important to at least know when and how these pools are created so you know when objects will eventually be deallocated and how much "garbage" may accumulate beforehand.

To create a new autorelease pool, you simply create an instance of the `NSAutoreleasePool` class:

```
NSAutoreleasePool *myPool = [[NSAutoreleasePool alloc] init];
```


The `NSAutoreleasePool` class is slightly magic in that creating a new pool also adds the pool to a stack. Any object that receives an autorelease message is automatically added to the pool currently on the top of the stack (the most recently created one).

Extra for experts

The stack of `NSAutoreleasePool` instances is thread specific, so each thread in your application maintains its own stack of autorelease pools.

If you attempt to send an autorelease message while on a thread that doesn't currently have an autorelease pool, you'll get the following output in the Xcode debugger console:

```
*** _NSAutoreleaseNoPool(): Object 0x3807490 of class NSString
    autoreleased with no pool in place - just leaking
```

This output indicates that the autorelease message hasn't managed to find an autorelease pool so has simply cast the object to one side, never to be released until the application exits.

Applications that use UIKit can take advantage of the fact that Cocoa Touch automatically creates and releases a new autorelease pool each time through the application's run loop.

A GUI-based application can be considered to be one big `while(true)` loop. This loop, called a *run loop*, continues to cycle until the application exits. At the top of the loop, Cocoa Touch waits for a new event to occur, such as the device's orientation changing or the user tapping on a control. Cocoa Touch then dispatches this event to your code, most likely a `UIViewController`, to be processed, before returning to the top of the loop and waiting for the next event to occur.

At the beginning of an iteration through the run loop, Cocoa Touch allocates a new `NSAutoreleasePool` instance, and once your view controller has processed the event, the run loop will release the pool before returning to the top of the run loop to await the next event.

You've already created an autorelease pool

In every application you've written so far, you've created an explicit `NSAutoreleasePool`, probably without even knowing it. As an example, open the Rental Manager project and view the file named `main.c`. This should be located in the Other Sources folder and will contain the following code snippet:

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
int retVal = UIApplicationMain(argc, argv, nil, nil);
[pool release];
```

This code creates an autorelease pool that exists for the entire lifetime of your application.

With knowledge of how to create autorelease pools and assign objects to them, we can finally discuss the important concept of how objects in an autorelease pool are eventually deallocated.

9.3.4 Releasing objects in a pool

Once a pool is created, objects that are sent an autorelease message will automatically find it. When the owner of the autorelease pool wants to release ownership of all objects held by the pool, it releases the pool as follows:

```
[myPool release];
```

When the pool is deallocated, it automatically sends a release message to each object assigned to the pool.

Sometimes you want to release all objects in a pool but keep the pool functional for future use. Although you could do the following

```
NSAutoreleasePool *myPool = [[NSAutoreleasePool alloc] init];  
...  
[myPool release];  
myPool = [[NSAutoreleasePool alloc] init];  
...  
[myPool release];
```

the overhead of creating and deallocating each pool could become significant. For this reason, the `NSAutoreleasePool` class also provides a `drain` message, which releases any currently contained objects but keeps the pool ready to accept additional objects. Using `drain`, the previous code snippet can be rewritten as follows:

```
NSAutoreleasePool *myPool = [[NSAutoreleasePool alloc] init];  
...  
[myPool drain];  
...  
[myPool release];
```

Both code snippets are identical in memory management behavior and the associated object lifetimes, but the second avoids the overhead of creating and destroying a second autorelease pool.

9.3.5 Why not use an autorelease pool for everything?

After discovering the concept of autorelease pools, you may wonder why you can't just add every object to an autorelease pool and avoid the need to carefully match retain and release messages. The answer to this is generally one of performance and memory consumption—two issues that are vitally important on a constrained and typically battery-powered mobile device.

When adding objects to an autorelease pool, you're ensured that the object will eventually be released on your behalf, but implicitly you also know this will be at some point in the future. Depending on your application logic, these objects could persist for a comparatively long time after you're done using them—in effect, artificially

Be careful of macro optimizations

When developing your application, be careful not to optimize for performance on a macro level. Although there are performance-based overheads to using an `NSAutoreleasePool`, these overheads could be an insignificant factor in the overall performance of your application.

It's generally better to use performance-monitoring and analysis tools, such as those mentioned in chapter 14, to determine where the bottlenecks occur in your application's performance than to apply general rules of thumb, such as "manual retain/release is more efficient than retain/autorelease."

inflating the memory consumption of your application. This may not be much of a problem on a desktop, but it can be a big problem on a memory-constrained device such as the iPhone. As an extreme case, consider the following code snippet:

```
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 1000; j++) {
        NSString *msg = [[NSString alloc] initWithFormat:
            @"Message number %d is 'Hello, World!'",
            i * 1000 + j];
        [msg autorelease];
        NSLog(@"%@", msg);
    }
}
```

This code creates 100,000 string objects. At the end of each iteration through the inner loop, an additional string is added to the autorelease pool even though that string will never be referred to again by the code snippet. All 100,000 string objects will pile up in the pool, only to be released when the entire code snippet completes and returns to the application's run loop. At least in theory...

If you run this code snippet, you'll probably find that your application crashes well before the 100,000th message is displayed with a message similar to the following:

```
RentalManager (4888) malloc: *** mmap(size=16777216) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
```

This indicates that the Objective-C runtime attempted to allocate some memory for a new object but failed to find enough. This seems extreme considering the code interacts with only a single string at any given point in time.

One solution is to explicitly create your own autorelease pool so you can release the string objects earlier, as demonstrated by the following code snippet:

```
for (int i = 0; i < 100; i++) {
    NSAutoreleasePool *myPool = [[NSAutoreleasePool alloc] init];
    for (int j = 0; j < 1000; j++) {
        NSString *msg = [[NSString alloc] initWithFormat:
            @"Message number %d is 'Hello, World!'",
```

```

        i * 1000 + j];
    [msg autorelease];
    NSLog(@"%@", msg);
}
[myPool release];
}

```

Be careful running these code samples in the iPhone Simulator

When trying out the autorelease-based examples in this section, be aware that the iPhone Simulator won't behave the same as an actual iPhone device (and an iPhone 3GS will behave differently from an original iPhone, iPod Touch, or iPad, and so on).

The iPhone Simulator runs your iPhone applications on your desktop machine using the resources available to it. This means an application running in the simulator typically has access to significantly greater amounts of memory than a real iPhone provides. Likewise, as your desktop machine starts to run out of memory, it'll likely start paging some out to disk, another feature not present on a real iPhone.

You therefore might find that all three examples succeed when run in the iPhone Simulator environment. If so, increase the outer loop counter (*i*) to a much larger number, which will make the application consume more memory. Alternatively, you can run these examples on an actual iPhone device.

In this code snippet, you `alloc` and `release` an autorelease pool each time around the outer loop. This ensures that the pool has, at most, 1000 objects before it's released. Notice that the positioning of the autorelease pool is a tradeoff between efficiency and object lifetimes. The code sample would work equally well if the `NSAutoreleasePool` were placed in the inner loop. But then you would have incurred the expense of allocating and deallocating an `NSAutoreleasePool` 100,000 times, with only one object ever assigned to each pool.

By placing the `NSAutoreleasePool` in the outer loop, you reduce this expense 1000-fold, but you'll consume more memory as the pool grows to 1000 objects.

In some cases, object lifetimes are easily managed without the need for an autorelease pool. For example, the previous two examples could easily be modified to use a release message instead of an autorelease, as demonstrated here:

```

for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 1000; j++) {
        NSString *msg = [[NSString alloc] initWithFormat:
            @"Message number %d is 'Hello, World!'",
            i * 1000 + j];

        NSLog(@"%@", msg);

        [msg release];
    }
}

```

You know the string object isn't required after its contents are logged to the console, so you can explicitly release the object. This means only one `NSString` object exists at a time, drastically reducing the memory usage of the code snippet. Notice also that the release message had to be moved to the end of the inner loop: because the string object is explicitly deallocated, it's no longer safe to refer to it after you've called `release`. This is unlike the situation in which you sent the object an `autorelease` object and could have happily continued to use the object until the `autorelease` pool itself was released. If you didn't move the `release` message send to come after the `NSLog` statement, the log statement would be attempting to log a potentially deallocated object.

9.4 *Memory zones*

As you investigate the Apple Developer documentation or use Xcode's Code Sense feature, you may notice that many objects have an `allocWithZone:` message as well as the more familiar `alloc`. This message and others, such as `copyWithZone:` and `mutableCopyWithZone:`, hint at a concept called *memory zones*, which Objective-C represents via the `NSZone` struct.

By default, when you request an object to be allocated (by sending the class an `alloc` message), Objective-C allocates the required amount of memory from the default memory zone. If your application allocates a large number of objects with varying lifetimes, this can lead to a situation called *memory fragmentation*, as demonstrated by figure 9.2.

Memory fragmentation means that, although there's enough free memory available to the application, it's spread throughout the entire memory space and in potentially inconveniently sized blocks. If your application is allocating a number of objects of similar size or wants to keep them close to each other, the solution may be to place these objects in a separate memory zone.

To create a new memory zone, you use the `NSCreateZone` function, which allocates a specific amount of memory to service future memory allocation requests directed toward it:

```
#define MB (1024 * 1024)
NSZone *myZone = NSCreateZone(2 * MB, 1 * MB, FALSE);
```

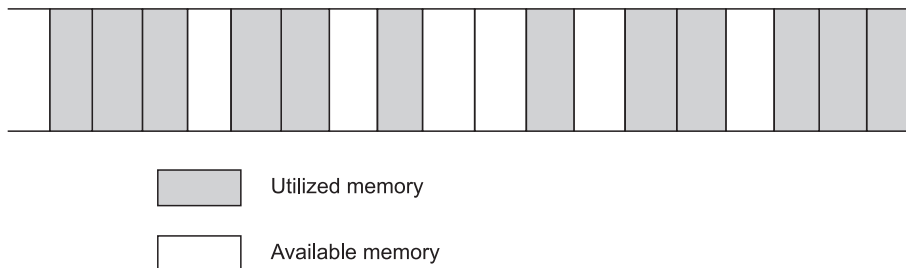


Figure 9.2 A conceptual view of the memory fragmentation problem. Although there are 4 bytes of spare memory available, a request to allocate 4 bytes for a variable of type `int` will fail because those 4 bytes aren't available in a contiguous block.

Memory zones have additional advantages

Although not a consideration for the iPhone and iPad, which don't implement paging to a secondary storage mechanism (such as disk) in their virtual memory implementations, another use of custom memory zones is to ensure the locality of related data.

Custom memory zones are commonly used in audio buffer or streaming-related code, for example, because they help prevent performance impacts caused by swapping buffers in and out of disk-based storage if two commonly used buffers happen to be allocated on different memory pages.

The first argument to `NSCreateZone` specifies how much memory to initially reserve for the pool, and the second argument specifies the granularity of future memory allocations for the pool. In this example, 2 megabytes of memory are preallocated for use by the pool; if all of this memory is eventually consumed, a request to allocate one additional byte will expand the memory zone by a further 1 megabyte. This helps reduce fragmentation and helps ensure the locality of different objects allocated from the same zone.

Once a memory zone is created, you can allocate a new object using memory from the zone by sending an `allocWithZone:` message:

```
NSString *myString = [[NSString allocWithZone:myZone] init];
```

Incidentally, another way to allocate an object from the default zone is as follows. Seeing this example should also help cement the relationship between `init` and `initWithZone:` for you:

```
NSZone *defaultZone = NSDefaultMallocZone();
NSString *myString = [[NSString allocWithZone:defaultZone] init];
```

Sending a class an `alloc` message (or passing `nil` to `allocWithZone:`) causes the object to be allocated from the default memory zone, which can also be obtained via a call to the `NSDefaultMallocZone` function.

Which zone does my object belong to?

To determine which zone an object is allocated to, you can send it a `zone` message and compare the value to a list of known memory zones:

```
NSZone *zoneOfObject = [myObject zone];
if (zoneOfObject == NSDefaultMallocZone()) {
    NSLog(@"The object was allocated from the default zone");
}
```

Objects allocated to a custom memory zone respond to standard memory management methods such as `retain`, `release`, and `autorelease`. But there are a couple of additional functions worth mentioning. When an object is deallocated, the zone

doesn't immediately return that memory to the operating system for reuse. It keeps it in case a future allocation request is made. At any time, a call to `NSRecycleZone` causes such memory to be freed. To free the entire zone, a function called `malloc_destroy_zone` is available. This function deallocates all memory associated with a zone even if that memory is associated with active objects. It's your responsibility to ensure all objects in a zone are properly deallocated before calling `malloc_destroy_zone`.

Using additional memory zones helps prevent memory fragmentation and improves performance. But their use brings an increase in the memory consumption of your application, as memory allocated to a zone can be used to service memory requests only for objects explicitly added to that zone. Therefore, creating additional memory zones should be considered only when performance requirements dictate their need, not as a standard part of your toolbox.

9.5 *Rules for object ownership*

We've discussed two ways to become the owner of an object:

- Create the object via an `alloc` or `allocWithZone:` message
- Send the object a `retain` message

There are other cases to consider, though. For example, a message such as `NSString`'s `stringWithObject` creates a new object but is clearly not named `alloc`, so who owns the resultant object?

In Cocoa Touch all memory management rules ultimately boil down to the following set of principles:

- 1 You own any object you explicitly create.
- 2 You create an object using a message whose name begins with `alloc`, `new`, or contains `copy` (`alloc`, `mutableCopy`, and so on).
- 3 You can share ownership of an existing object by sending it a `retain` message.
- 4 You must relinquish ownership of any object you own by sending it a `release` or `autorelease` message.
- 5 You must not relinquish ownership of an object you don't own.

Based on these more formalized principles, you can determine that the following code snippet observes correct memory management techniques:

```
NSString *stringA = [[NSString alloc] initWithString:@"Hello, world!"];
NSString *stringB = [NSString stringWithString:@"Hello, world!"];
[stringA release];
```

The first string is created by a method named `alloc`, so rules 1 and 2 indicate you own it. Therefore, when you're finished using the object, you must send it a `release` message to satisfy rule 4. The second string is created by a message named `stringWithString:`, so rule 2 indicates you don't own it. Therefore, rule 5 suggests you shouldn't release it.

Taking a look at classes in Cocoa Touch, you'll notice that many follow a pattern whereby an `initWithXYZ` message is matched with a similar class method named `classNameWithXYZ`. For example, `NSString` has

```
- (NSString *)initWithString:(NSString *)str;
+ (NSString *)stringWithString:(NSString *)str;
```

The first line initializes a string you've explicitly allocated with a call to `alloc`. The second performs a similar task, but it doesn't need a string to be allocated beforehand. Internally this method allocates a new string object and initializes it as required before sending it an `autorelease` message and returning it.

9.6 Responding to low-memory warnings

The iPhone operating system implements a cooperative approach when it comes to memory management and provides a warning to each application when the device finds itself in a low-memory situation.

If the amount of free memory drops below a threshold, the operating system attempts to release any excess memory that it holds, and if this doesn't free up enough memory, it also sends currently running applications a warning. If your application receives a low-memory warning, it's your chance to free up as much memory as possible by deallocating unneeded objects or by clearing out cached information that can easily be regenerated when next required.

The UIKit application framework provides access to the low-memory warning via a number of mechanisms:

- Implementing the `applicationDidReceiveMemoryWarning:` message in your application delegate
- Overriding the `didReceiveMemoryWarning` message in your `UIViewController` subclass
- Registering for the `UIApplicationDidReceiveMemoryWarningNotification` notification

We cover each of these scenarios in turn. It's important that your application responds to these warnings. If your application doesn't respond, or you don't manage to free enough memory, the operating system may take even more drastic measures such as "randomly" and abruptly exiting your application in its effort to gain additional free memory.

9.6.1 Implementing the `UIApplicationDelegate` protocol

As discussed in chapter 7, protocols can have optional messages, which you can selectively decide to implement. The `UIApplicationDelegate` protocol (implemented by the `RentalManagerAppDelegate` class in the Rental Manager application) has an optional message called `applicationDidReceiveMemoryWarning:`, which is sent whenever the application detects a low-memory situation. This message can be implemented as follows:

```
- (void)applicationDidReceiveMemoryWarning:(UIApplication *)application
{
    NSLog(@"Hello, we are in a low-memory situation!");
}
```

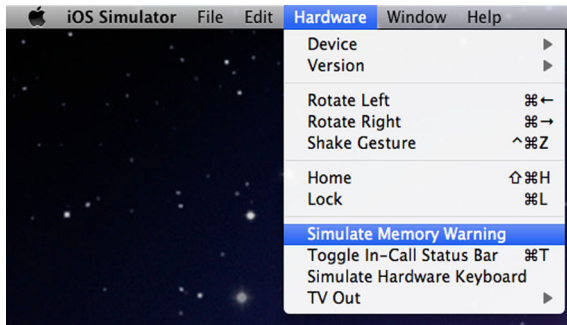



Figure 9.3 The Simulate Memory Warning option found in the Hardware menu of the iPhone Simulator can be used to test how well your application responds to low-memory scenarios.

For reasons outlined previously, the iPhone Simulator isn't a representative example of the kinds of memory constraints your applications will find themselves in while running on a real device. To help developers test their applications in low-memory conditions, the iPhone Simulator provides a way to generate a simulated low-memory situation. This feature can be initiated by selecting Simulate Memory Warning in the Hardware menu (see figure 9.3). The simulator fakes a low-memory situation, no matter how much memory is currently available on your computer.

When you select the Simulate Memory Warning option, you should see that your application delegate is sent an `applicationDidReceiveMemoryWarning:` message, as demonstrated by the following log messages appearing in the Xcode debugger console:

```
Received simulated memory warning.
Hello, we are in a low-memory situation!
```

Notice that the iPhone Simulator also emits a log message stating that the low-memory situation was faked. Obviously, if a real low-memory situation occurs on a device, you wouldn't see this additional log message.

In a typical application it's possible that your `UIApplicationDelegate` implementation may not have easy access to significant resources that it can free on a moment's notice. It's more probable that the individual `UIView`s and `UIViewController`s that make up the visual aspects of your application will have access to the memory resources that can be returned to the operating system. Therefore, the `UIViewController` class provides an alternative mechanism to access low-memory warnings in the form of a method called `didReceiveMemoryWarning` that can be overridden by interested subclasses.

9.6.2 **Overriding `didReceiveMemoryWarning`**

In a `UIViewController` subclass, such as the `RootViewController` class in the Rental Manager application, you can handle the low-memory warning condition by implementing a method named `didReceiveMemoryWarning`:

```
- (void) didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];

    [cityMappings release];
}
```

```
    cityMappings = nil;
}
```

In this implementation, when the operating system requests additional memory to be freed, the `RootViewController` class releases the city mapping `NSDictionary`. This frees memory immediately, but before you next need to look up an image to display for a given city, you'll have to re-create the dictionary and its contents. You'll see how this scenario is handled by the existing source code in a minute, but let's first investigate the implications of a comment inserted by the iPhone project templates.

If you create a new project in Xcode and select the Navigation-based Application template, you may notice that it provides an override for the `didReceiveMemoryWarning` method. This default implementation doesn't perform any additional functionality. It simply forwards the message onto the `UIViewController` superclass, but it does include a comment hinting at an important fact to remember. This comment and a related one in the `viewDidLoad` method are repeated here:

```
didReceiveMemoryWarning
    // Releases the view if it doesn't have a superview.

viewDidLoad
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
```

These comments hint that the UIKit framework will cache views that are currently offscreen, such as when the user temporarily switches to another tab in a tab bar application or drills down to another view in a navigation-based application.

When a view controller is initialized, it loads the associated view from a NIB file (or creates it from scratch via code). Once loaded, the view controller keeps the user interface controls in memory even if the view is temporarily moved offscreen.

Keeping the view in memory is efficient from a performance perspective because it allows the view to be redisplayed without repeating the construction process, but it comes with a potential negative in the form of increased memory consumption. This is especially true if you have a deep hierarchy in a navigation-based UI or if you have many tabs with complex UI requirements that will cause a number of offscreen views to be cached.

The UIKit framework is designed to cope with this scenario. It automatically changes its view-caching behavior whenever your application finds itself in a low-memory situation. When a `UIViewController` receives a `didReceiveMemoryWarning` message, and the view it controls is currently offscreen (it doesn't have a `Superview`, or parent), the `UIViewController` sends the view a release message. In most cases, this is enough to cause the view to immediately become deallocated, and hence this feature reduces the amount of memory consumed by the application without affecting the visual appearance or behavior of what the user can currently see or interact with.

This memory optimization, however, can be undone if your view controller keeps objects in the view alive because they haven't yet been released. This is what is hinted at by the comment in `viewDidLoad`: "Release any retained subviews of the main

view". If you refer back to chapter 1, you'll notice that the `CoinTossViewController` class implemented `viewDidLoad` as follows:

```
- (void)viewDidLoad {
    self.status = nil;
    self.result = nil;
}
```

These statements, which were not discussed at the time, ensure that when the view controller receives a `didReceiveMemoryWarning` message and decides to release the view, the two `UILabel` controls are also released. As a rule of thumb, any subview or control that's retained by an ivar or property in the view controller class should be released whenever the `viewDidLoad` message is received.

Correct memory management requires careful attention to detail

As mentioned in the sidebar "Danger, Will Robinson, here be sleeping dragons" in chapter 5, it's important to note the difference between the following three statements:

```
[status release];          status = nil;          self.status = nil;
```

The first statement sends the object pointed to via the `status` instance variable a `release` message, hopefully freeing its associated resources. The statement doesn't alter the value of the `status` variable, leaving it to point to the now nonexistent object.

The second statement attempts to resolve this by setting the variable to `nil`. Doing this doesn't send the object a `release` message, and hence the `UILabel` control will never be deallocated even though you've lost the ability to refer to it.

The last statement (and its logical equivalent `[self setStatus:nil]`) performs both tasks by invoking the setter method associated with the `status` property. From chapter 5, remember that a property with the `retain` attribute specified automatically sends a `release` message to its old value when the property is being set to a new value, such as the `nil` specified here.

Knowing that a `UIViewController` could decide to deallocate its view and re-create it when next required can have important implications for where you can place initialization logic or store application state. Get it wrong, and the application may appear to behave correctly during development, only to fail the first time your customer sees a low-memory situation develop.

To see this process in action, create a new iPhone application using the Tab Bar Application template and uncomment the source code for the `viewDidLoad` method within the `FirstViewController` class. Then add a breakpoint to the beginning of the `viewDidLoad` and `viewDidUnload` methods.

When you launch the application, you should see that `viewDidLoad` is immediately called to initialize the UI controls loaded from the `FirstView` NIB file.

If you switch to the second tab and then return to the first, you shouldn't notice any additional breakpoints being hit. This is because the UIKit framework keeps the

original view in memory and simply reuses it when the user returns to it (you could handle the `viewDidAppear` message if you needed to perform an action each time the view becomes visible to the user).

By comparison, if you switch to the second tab and then simulate a low-memory situation, the debugger should break in `FirstViewController`'s `viewDidUnload` method, indicating the view controller has decided it can help resolve the low-memory situation by deallocating the view that's currently invisible to the user.

Once the root view has been unloaded, you can continue using the application with no obvious difference in behavior. The next time you return to the first tab, you should notice that your `viewDidLoad` implementation is automatically invoked in order to reinitialize the view that the framework has just reloaded from your NIB file.

Knowing that a view can suddenly and perhaps unexpectedly become deallocated, you should be careful not to store important application state in the view. Apart from breaking the model-view-controller design pattern principles, it won't work reliably. As an example, if you use the enabled state of a `UIButton` or a selected item in a `UITableView` to control the logic or behavior of your application, you may notice your application enters an unexpected state when the view becomes deallocated and then re-created, causing all of the controls to return to the state in which they were declared in the NIB file.

In general, the correct place for view initialization logic is in the `viewDidLoad` method. This logic is invoked whenever the view associated with the view controller is constructed. If you take a look at `RootViewController`'s `viewDidLoad` method implementation within the Rental Manager application, you'll notice this is exactly where the `cityMappings` dictionary is created, ensuring it's also re-created if it's destroyed due to a low-memory situation.

As an extension to this rule of thumb, you may also like to override the matching `viewDidUnload` message as an ideal place to save application state just before the view is removed from memory.

9.6.3 Observing the `UIApplicationDidReceiveMemoryWarningNotification`

Sometimes when writing a support class deep in your application logic, it may be inconvenient for a view controller or application delegate to be responsible for freeing any associated resources in a low-memory situation. For one thing, doing so would break encapsulation, as something external to the class would require knowledge of and access to its internals in order to free them. For such situations, Cocoa Touch provides an alternative solution via a feature called *notifications*.

A notification allows a class to respond to an event, such as a low-memory situation, no matter where the class is positioned in the application; it no longer needs to be a `UIViewController` subclass. In general, an object called the sender will generate a notification and dispatch it to be received by zero or more objects called observers.

The glue that connects notification senders with observers is a class called `NSNotificationCenter`. This class provides a mechanism for senders to broadcast

new notifications and observers to register their interest in receiving particular types of notifications. A notification center is similar in concept to an email distribution list. One or more email addresses (observers) are registered with the mailing list (notification center), and when a new email (notification) is sent by a sender, the mailing list distributes it to all currently registered email addresses. A given email address may not receive all emails sent to the server because it's unlikely the address will be registered with every email list offered by the mail server.

In Objective-C the first step in handling a notification is to define a method you want invoked whenever a suitable notification occurs. For example, you could define the following method in the `CTRentalProperty` class:

```
- (void)HandleLowMemoryWarning:(NSNotification *)notification
{
    NSLog(@"HandleLowMemoryWarning was called");
}
```

Once the method is defined, you must inform the `NSNotificationCenter` that you're interested in observing a particular type of notification. This can be done by sending an `addObserver:selector:name:object:message`, as shown in the following listing.

Listing 9.3 Registering intent to observe memory warning notifications

```
- (id)initWithAddress:(NSString *)newAddress
    rentalPrice:(float)newRentalPrice
    andType:(PropertyType)newPropertyType {
    if ((self = [super init])) {
        self.address = newAddress;
        self.rentalPrice = newRentalPrice;
        self.propertyType = newPropertyType;

        [[NSNotificationCenter defaultCenter]
         addObserver:self
         selector:@selector(HandleLowMemoryWarning)
         name:UIApplicationDidReceiveMemoryWarningNotification
         object:nil];
    }

    return self;
}
```

The observer and selector arguments specify which object should act as the observer and the message that object wishes to be sent when a suitable notification occurs. The last two arguments specify the particular notification you're interested in observing. This example specifies that you want to observe a notification called `UIApplicationDidReceiveMemoryWarningNotification`. Passing `nil` for the object argument means that you're interested in observing this event no matter who the sender is. If you're interested only in notifications generated by a particular object, you can specify that object instead to filter out notifications generated by other objects.

Registering with a notification center indicates your intent to observe a notification event, so it's natural to need to unregister your intent when you no longer want to

addObserver:selector:name:object: provides a lot of flexibility

The `addObserver:selector:name:object:` message provides a lot of flexibility when it comes to deciding how notifications are handled by an observer.

Since it's possible to call `addObserver:selector:name:object:` multiple times for the same observer object, with a different `name` argument each time, it's possible for a single object to process more than one type of notification.

Likewise, via use of the `selector` argument, you can determine if different notification types are handled by separate methods or lumped together and processed by a single method.

If you decide a single method can process multiple types of notifications, the `NSNotification` object passed into the method has a `name` property, which can help distinguish why it is currently invoked.

receive notification events. This can be done by sending a `removeObserver:name:object:`, as demonstrated here:

```
- (void)dealloc {
    [[NSNotificationCenter defaultCenter]
     removeObserver:self
     name:UIApplicationDidReceiveMemoryWarningNotification
     object:nil];

    [address release];
    [super dealloc];
}
```

It's important to unregister your observers before your object is deallocated. Otherwise, when the notification is next generated, the `NSNotificationCenter` will attempt to send a message to the nonexistent object and your application may crash.

9.7 Summary

Responsibility for correct memory management in Objective-C rests significantly with you, the developer. Unlike in garbage-collected environments, which tend to be use-and-forget type environments, the deallocation of objects in Objective-C will occur at the correct point in time only through the proper use of the `retain`, `release`, and `autorelease` messages. Make a mistake, and the object will never be deallocated or, worse yet, it may be deallocated too early and lead to application crashes.

To make things easier, most Objective-C classes strive to stick to a common memory management pattern, which was discussed in section 9.5. By handling memory allocation in a consistent manner, you won't need to constantly refer to documentation in order to determine who's responsible for keeping track of a given object.

An autorelease pool enables some of the benefits of a garbage-collected environment without the implied overhead. An object can be registered to be sent a `release` message at some point in the future. This means you don't use the traditional `retain`

and `release` messages for the object but are still safe in the knowledge that the object will eventually be deallocated.

Since the release of iOS 4.0 and its multitasking capabilities, it's even more important for your application to be a good iPhone citizen. While developing your applications, you should take the extra couple of minutes required to make your application listen to and respond to low-memory notifications. Provided your calls to `retain` and `release` are correctly matched and you can free some resources in response to a low-memory situation, your end user shouldn't see any difference in behavior in your application. Implementing the low-memory notifications may be the difference between a happy customer and an iTunes review comment stating your application randomly crashes.

In chapter 10 you learn all about error and exception handling when developing with Objective-C in your iOS applications.

Part 3

Making maximum use of framework functionality

If every iOS application had to reinvent the wheel, there would be significantly fewer applications in the iTunes App Store, because developers would waste time recreating classes to perform similar functionality. The Cocoa Touch environment provides a number of support libraries and classes that, with a few simple lines of code, enable fairly powerful and complex behavior to be available for application developers to use. To quickly come to market, and focus more time on the aspects of your applications that make it unique, it's important for application developers to know what kind of services the iOS SDK provides, and how to take advantage of them.

This part of the book begins by showing how Objective-C frameworks typically handle error conditions and exceptions. It then takes a deep dive into two semi-related technologies, called *Key Value Coding (KVC)* and *NSPredicate*, that allow flexible sorting and filtering to be introduced within your applications. These two technologies also demonstrate how certain knowledge can be reutilized, as shown in the discussion about using Core Data to work with databases using KVC and NSPredicate.

The second half of this part shows how you can deliver a great user experience within your application while performing many tasks as once, or waiting for external web services to respond. Nothing is worse than an application that

becomes unresponsive for long periods of time, and Grand Central Dispatch (GCD) can help with this. We also introduce some tools that can help with debugging and diagnosing performance- and memory-related issues as your own applications become more complex.

10

Error and exception handling

This chapter covers

- Dealing with NSError
- Creating your own NSError objects
- When and when not to use exceptions

Things go wrong. That's no surprise. And things go wrong in the software you build. Maybe the server that hosts the API you want to talk to isn't reachable, or the XML file you need to parse is corrupt, or the file you want to open doesn't exist. You need to prepare for such cases and handle them gracefully, which means not crashing *and* giving the user an appropriate message.

Cocoa Touch and Objective-C offer two ways of dealing with errors: NSError and exceptions. NSError is the preferred and advisable way to deal with expected errors (like an unreachable host). Exceptions are meant to report and handle programming errors during development and shouldn't be used to handle and report errors to the user at runtime. That makes the purpose of exceptions in Objective-C quite different from other languages, such as Java, where they're used frequently.

In this chapter we look mainly at NSError and at some real-life use cases for exceptions.

10.1 *NSError—handling errors the Cocoa way*

In Cocoa and Cocoa Touch, nearly all methods that could possibly fail for reasons you as a developer can't control take a pointer to an NSError pointer as their last parameter. Why a pointer to a pointer? Because that way the called method can create an instance of NSError and you can access it afterwards through your NSError pointer.

Before we go any further, let's look at an example in the following listing.

Listing 10.1 Trying to load a nonexistent file

```
NSError *error;
NSData *data = nil;

data = [[NSData dataWithContentsOfFile:@"i-dont-exist"
        options:NSDataReadingUncached
        error:&error] retain];

if (data == nil) {
    NSLog(@"An error occurred: %@",
          [error localizedDescription]);
} else {
    NSLog(@"Everything's fine");
}
```

This example is simple: you create an NSError pointer variable called `error`. Then you try to initialize an NSData object with a nonexistent file and pass in the address of your pointer variable (`&error`). Next you check whether or not you got a data object. This step is extremely important: when no error has occurred, the state of the passed-in error pointer is undefined! It's unsafe to do anything with the error pointer unless you're sure that an error did occur. That's why you always need to check the return value of a method before accessing the error object.

When you run this code, you'll notice that the localized description of the error in listing 10.1 wouldn't be very helpful to your users, though:

```
"An error occurred: The operation couldn't be completed. (Cocoa error 260.)"
```

Let's look at NSError in more depth so you know how to get useful information from it to display to your users.

10.1.1 *Getting NSError to talk*

NSError objects have both an error domain and an error code. The domain exists mostly for historical reasons, but it provides useful information because it tells you which subsystem reported the error. The error domain is a string, and the Foundation framework declares four major error domains: `NSMachErrorDomain`, `NSPOSIXErrorDomain`, `NSOSStatusErrorDomain`, and `NSCocoaErrorDomain`. Frameworks like Core Data have their own error domains. They provide namespaces for error codes or group them.

Error codes are numeric and tell you which error occurred. The error codes for the different subsystems are usually defined in header files and can also be found in the online documentation (<http://developer.apple.com/library/ios/navigation/>). The

error codes for Foundation, for example, are defined in `<Foundation/Foundation-Errors.h>`. Because they're numeric, it's convenient to use them in a switch statement to handle each expected error code in the appropriate way. Neither the domain nor the code is useful to users, though. Luckily, `NSError` provides much more information.

The `localizedDescription` method is guaranteed to always return a string that can be displayed to the user. Often it gives a useful explanation of the error. (We look at what to do when the localized description isn't enough in the next section.) `NSError` also has three other methods that might provide useful information to the user: `localizedFailureReason`, `localizedRecoverySuggestion`, and `localizedRecoveryOptions`. These methods aren't guaranteed to return anything, though.

So what can you do when all of these pieces of information aren't enough? You can examine the `userInfo` dictionary!

10.1.2 Examining NSError's userInfo Dictionary

The `userInfo` dictionary of `NSError` is a flexible way to provide a wealth of additional information about an error. You can easily examine its contents by simply writing it to the console with `NSLog`. Let's do that now, and go back to listing 10.1 and change `NSLog(@"An error occurred: %@", [error localizedDescription]);` to `NSLog(@"An error occurred: %@", [error userInfo]);` and run it again. The output will look something like this now:

```
An error occurred: {
    NSFilePath = "i-dont-exist";
    NSUnderlyingError = "Error Domain=NSPOSIXErrorDomain Code=2 \"The operation
        couldn\u2019t be completed. No such file or directory\"";
}
```

Obviously, the information in the `NSUnderlyingError` key would be much more useful to users: "No such file or directory" describes it quite well. The value of `NSUnderlyingError` in the `userInfo` dictionary contains—if it exists—another `NSError` object, the one that actually describes the root of the problem. To display the most detailed error message to your users, you can do something like the code in the following listing.

Listing 10.2 Displaying a more detailed message

```
NSDictionary *userInfo = [error userInfo];
NSString *message;

if (userInfo && [userInfo
    objectForKey:NSUnderlyingErrorKey]) {
    NSError *underlyingError = [userInfo
        objectForKey:NSUnderlyingErrorKey];
    message = [underlyingError localizedDescription];
} else {
    message = [error localizedDescription];
}

UIAlertView *alert = [[UIAlertView alloc]
    initWithTitle:@"Error"
```

```

message:message
delegate:nil
cancelButtonTitle:nil
otherButtonTitles:@"OK", nil];

[alert show];
[alert release];

```

Listing 10.2 first checks whether the error object has a `userInfo` dictionary and whether that dictionary has a value for the `NSUnderlyingErrorKey`. If so, it displays its `localizedDescription`. If not, it displays the main error object's `localizedDescription`.

This technique doesn't produce the best message for users in all cases. During development it's always important to test for different error conditions and look at the contents of the `userInfo` dictionary because different frameworks and different subsystems provide different amounts and kinds of information in the `NSError` objects they create. There's no one-size-fits-all solution to display error messages to your users. In most cases, you know what could go wrong in a certain piece of code. Use that knowledge to create and display the most helpful and user-friendly message possible.

What if you want to create your own `NSError` objects? Let's look at that next.

10.2 *Creating NSError objects*

Adopting Cocoa's error-handling pattern in your own code is simple. You just need to accept a pointer to an `NSError` pointer as an argument to your method, check if one has been provided by the caller, and create an instance of `NSError` with the information describing the error if something went wrong.

Let's look at an example.

10.2.1 *Introducing RentalManagerAPI*

Pretend you have to build a framework that talks to the `my-awesome-rental-manager.com` website. You want to give this framework to third-party developers to use in their iOS projects, and naturally you want to build a beautiful API that handles errors gracefully. One of the framework's methods takes a dictionary of values that represents a classified ad, which is used here for simplicity's sake; in real life you'd use a custom class to represent such an ad. The method publishes the ad to the website after checking that all required values are present. If not, it creates and reports an error that indicates which fields are missing, as shown in figure 10.1.

Since we're concentrating on how to create and handle errors in this chapter, we won't go into every detail of setting up a demo project or designing the user interface for this application in Interface Builder. If you want to use the techniques you've learned in the previous

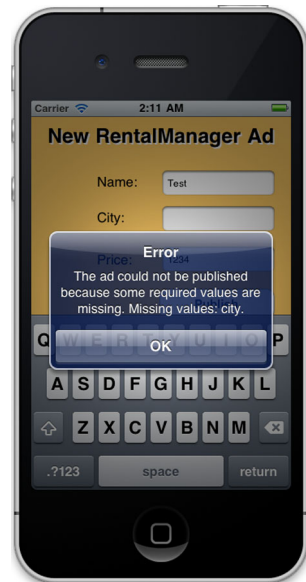


Figure 10.1 The `RentalManagerAPI` application displaying an error

chapters to follow along, by all means do so. Creating a new View-based application in Xcode should get you on the right track. Otherwise you can always examine the RentalManagerAPI project in the downloadable source code for this chapter.

Take a look at the code in the following listing.

Listing 10.3 RentalManagerAPI.h

```
#import <Foundation/Foundation.h>

extern NSString * const RMAMissingValuesKey;
extern NSString * const RMAAccountExpirationDateKey;

extern NSString * const RMAErrorDomain;

enum {
    RMAValidationError = 1,
    RMAAccountExpiredError = 2,
    RMAWrongCredentialsError = 3
};

@interface RentalManagerAPI : NSObject {
}

+ (BOOL)publishAd:(NSDictionary *)anAd
  error:(NSError **)anError;

@end
```

Required value missing
Account expired
Wrong user/password

The next listing contains the rest of the code.

Listing 10.4 RentalManagerAPI.m

```
#import "RentalManagerAPI.h"

NSString * const RMAErrorDomain =
    @"com.my-awesome-rental-manager.API";
NSString * const RMAMissingValuesKey = @"RMAMissingValuesKey";
NSString * const RMAAccountExpirationDateKey =
    @"RMAAccountExpirationDateKey";

@implementation RentalManagerAPI

+ (BOOL)publishAd:(NSDictionary *)anAd error:(NSError **)anError {
    if (anAd == nil) {
        @throw [NSException exceptionWithName:@"RMABadAPICall"
            reason:@"anAd dictionary is nil"
            userInfo:nil];
    }

    NSMutableArray *missingValues = [NSMutableArray array];
    for (NSString *key in [@"name price city"
        componentsSeparatedByString:@" "]) {
        if ([[anAd objectForKey:key] length] == 0) {
            [missingValues addObject:key];
        }
    }

    if ([missingValues count] > 0) {
```

Define custom error domain

If no ad object, throw an exception

Check values missing, if any

```

if (anError != NULL) {
    NSString *description = @"The ad could not be \
        published because some required \
        values are missing.";
    NSString *recoverySuggestion = @"Please provide \
        the missing values and try again.";
    NSArray *keys = [NSArray arrayWithObjects:
        NSLocalizedDescriptionKey,
        NSLocalizedRecoverySuggestionErrorKey,
        RMAMissingValuesKey, nil];
    NSArray *values = [NSArray arrayWithObjects:
        description,
        recoverySuggestion,
        missingValues, nil];
    NSDictionary *userDict = [NSDictionary
        dictionaryWithObjects:values
        forKeys:keys];

    *anError = [[NSError alloc] initWithDomain:
        RMAErrorDomain
        code:RMAValidationError
        userInfo:userDict] autorelease];
}
return NO;
} else {
    return YES;
}
}
@end

```

←
Caller interested in errors?

←
Create an instance of NSError

Let's examine the code. In the header file, some constants are set up: a custom error domain, some special keys for the `userInfo` dictionary, and some error codes. Using constants and enumerators for these things makes your code much more readable and maintainable. In the implementation of the `publishAd:error:` method, you'll notice one interesting thing right away: the use of the `@throw` directive. We briefly look at exceptions in the next section of this chapter, but here you see one common use case for them: catching programming errors. Your method should never be called without an instance of `NSDictionary`, and with this exception you make sure that such an error gets caught during development.

The rest of the method isn't rocket science: it checks if any required values are missing and collects them in an array. If some values are missing, you create a description and some recovery suggestions and put them in a dictionary together with the array of missing values. With that, you create an instance of `NSError` and have the pointer to a pointer (`anError`) point to it by using the asterisk in front of the variable name (`*anError`). That tells the compiler that you want to change the value of the pointer that `anError` is pointing to, not the value of `anError` itself. This is a little confusing, no doubt, but you'll wrap your head around it in no time.

Finally you return `NO` when an error occurred and `YES` otherwise. Now how do you call this method and handle the error it might return?

10.2.2 Handling and displaying RentalManagerAPI errors

Let's look at listing 10.5. What does this code do? First it prepares the ad dictionary from the values of some text fields. Then it declares a pointer of type NSError and calls the `publishAd:error:` method, checking its return value. If it's YES, a success message is displayed. If it's NO (which indicates an error), you check if the error domain is that of the RentalManagerAPI. If so, you check for different error codes and construct an appropriate message. Otherwise, you just use the `localizedDescription` and finally display it in a `UIAlertView`. That's it.

Listing 10.5 Calling the `publishAd:error:` method

```

- (IBAction)publishAd:(id)sender {
    NSArray *keys = [NSArray arrayWithObjects:@"name", @"city",
                                             @"price", nil];
    NSArray *values = [NSArray arrayWithObjects:nameTextField.text,
                                                cityTextField.text,
                                                priceTextField.text, nil];
    NSDictionary *ad = [NSDictionary dictionaryWithObjects:values
                    forKeys:keys];
    NSError *error;
    if ([RentalManagerAPI publishAd:ad error:&error]) {
        UIAlertView *av = [[UIAlertView alloc]
            initWithTitle:@"Success"
            message:@"Ad published successfully."
            delegate:nil
            cancelButtonTitle:@"OK"
            otherButtonTitles:nil];
        [av show];
        [av release];
    } else {
        NSString *message;
        if ([error domain] == RMAErrorDomain) {
            switch ([error code]) {
                case RMAValidationError:
                    message = [NSString
                        stringWithFormat:@"%@\nMissing values: %@.",
                            [error localizedDescription],
                            [[[error userInfo] objectForKey:
                                RMAMissingValuesKey]
                                componentsJoinedByString:@"", "]];
                    break;
                case RMAWrongCredentialsError:
                    break;
                default:
                    message = [error localizedDescription];
                    break;
            }
        } else {
            message = [error localizedDescription];
        }
    }
}

```

Annotations for Listing 10.5:

- Create the "ad"**: Points to the `NSDictionary *ad` creation.
- Create NSError pointer**: Points to the `NSError *error` declaration.
- Call RentalManagerAPI**: Points to the `[RentalManagerAPI publishAd:ad error:&error]` call.
- On success, display message**: Points to the success branch of the `if` statement.
- Custom error domain only**: Points to the `[error domain] == RMAErrorDomain` check.
- Handle errors differently**: Points to the `switch ([error code])` block.
- Messages for validation errors**: Points to the `RMAValidationError` case.
- Other errors, use its message**: Points to the `else` branch of the `if` statement.


```

UIAlertView *av = [[UIAlertView alloc]
    initWithTitle:@"Error"
    message:message
    delegate:nil
    cancelButtonTitle:@"OK"
    otherButtonTitles:nil];
[av show];
[av release];
}

```

← **Display error message**

You should now have a good fundamental understanding of how to deal with errors in Cocoa and how to create your own.

10.3 Exceptions

Exceptions play a different role in Objective-C and Cocoa than they do in languages like Java. In Cocoa, exceptions should be used only for programmer errors, and the program should quit quickly after such an exception is caught. Exceptions should never be used to report or handle expected errors (like an unreachable host or a file that can't be found). Instead, use them to catch programming errors like an argument being `nil` that should never be `nil` or an array having only one item when it's expected to have at least two. You get the idea.

So how do you create and throw exceptions?

10.3.1 Throwing exceptions

Exceptions are instances of `NSException` (or of a subclass thereof). They consist of a name, a reason, and an optional `userInfo` dictionary, much like `NSError`. You can use any string as an exception name or use one of Cocoa's predefined names that can be found at <http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/Exceptions/Concepts/PredefinedExceptions.html>.

Exceptions can be thrown either by using the `@throw` directive or by calling the `raise` method on an instance of `NSException`. The following listing shows an example of creating and throwing an exception with a predefined exception name.

Listing 10.6 Throwing an exception

```

if (aRequiredValue == nil) {
    @throw [NSException
        exceptionWithName:NSInvalidArgumentException
        reason:@"aRequiredValue is nil"
        userInfo:nil];
}

```

Uncaught exceptions cause an application to terminate. During development, that might be just what you want to be alerted that you did something wrong in your program code or logic. How would you go about catching exceptions, though?

10.3.2 Catching exceptions

Code that you expect to potentially throw an exception should be wrapped in a `@try` block, followed by one or more `@catch()` blocks that will catch exceptions that might get thrown or raised by the code in the `@try` block. Optionally, you can add a `@finally` block. The code in a `@finally` block gets run regardless of whether or not an exception occurred. Why can there be multiple `@catch` blocks? Because you can catch different kinds of exceptions, and the Objective-C runtime will pick the best-matching `@catch` block for the exception that's been thrown. Note that each `@try`, `@catch`, and `@finally` block has its own scope, so you can't access an object in the `@finally` block that's been declared in the `@try` block, for example. If you want to accomplish that, you must declare the variable outside of the `@try` block, in its enclosing scope. The following listing shows a simple example.

Listing 10.7 Catching an exception

```
@try {
    NSLog(@"Trying to add nil to an array...");
    NSMutableArray *array = [NSMutableArray array];
    [array addObject:nil];
}
@catch (NSException *e) {
    NSLog(@"Caught exception: %@, Reason: %@", [e name], [e reason]);
}
@finally {
    NSLog(@"Executed whether an exception was thrown or not.");
}
```

Inside the `@try` block, you attempt to insert `nil` into an array. The exception is caught by the `@catch` block, and afterwards the `@finally` block is run. The console output of this code looks like this:

```
Trying to add nil to an array...
Caught exception: NSInvalidArgumentException, Reason: *** -
[NSMutableArray insertObject:atIndex:]: attempt to insert nil object at 0
Executed whether or not an exception was thrown.
```

These exception-handling fundamentals should get you far, especially because exceptions are rarely used in Cocoa development—but they're useful for catching programming errors.

10.4 Summary

Error handling in Cocoa and Cocoa Touch is both simple and flexible. Don't ignore errors in your applications: handle them gracefully and tell your users about errors they need to know about in an understandable way. The concept of exceptions is present in Cocoa as well but plays a minor role compared to other frameworks and languages. Use them as intended: to catch errors during development.

In chapter 11, we dive into an important Cocoa concept that lets you access and manipulate values in powerful ways: Key-Value Coding and `NSPredicate`.

11

Key-Value Coding and NSPredicate

This chapter covers

- Key-Value Coding (KVC)
- Handling `nil` values in KVC
- Filtering a collection with predicates
- Using key paths in predicate expressions

KVC is a feature of the Foundation framework that allows you to access the properties of an object using strings. To obtain the current value of an object's property, you usually send the object an explicit getter message, as demonstrated here:

```
CTRentalProperty *house = ...;  
NSString *addressOfProperty = [house address];
```

Given an instance of the `CTRentalProperty` class developed in chapter 5, this would return the current value of the house's address property. In Objective C 2.0 you could also use the following alternative property accessor syntax:

```
NSString *addressOfProperty = house.address;
```

When using either syntax, it isn't possible at runtime to change which property is accessed, because the property name is explicitly hardcoded into the source code. You can work around this potential problem by using an `if` statement:

```
id value;
if (someConditionIsTrue)
    value = [house address];
else
    value = [house propertyType];
```

One issue with this kind of code structure is that you'd need to be aware of all possible properties at compile time. It's also a potential code maintenance nightmare, because each time a new property is added to the class, you must remember to also add a new clause to the `if` statement. KVC allows you to replace this brittle logic with a single statement similar to the following:

```
NSString *key = @"address";
NSString *value = [house valueForKey:key];
```

The `valueForKey:` message expects a string containing the name of a property and returns its current value. Because the property you want to access is expressed by a string, it's possible to change it at runtime or even generate it dynamically in response to user input and, in the process, modify which property is accessed.

KVC is one of the many technologies that enable more advanced frameworks such as Core Data to successfully interact with your own custom objects that were created separate from the framework. Core Data is covered in detail in chapter 12.

It's common to compare an object against a condition and then perform an action based on this comparison. This is commonly called a *predicate condition*, and in the Foundation framework, it's represented by the `NSPredicate` class. This class internally uses KVC and key paths, and we also cover how to take advantage of `NSPredicate` later in this chapter.

11.1 Making your objects KVC-compliant

KVC is all about conventions. In order for KVC to determine which setter or getter message should be sent to access a given key, you must conform to a specific set of rules. If you attempt to use a non-KVC-compliant object with KVC, you'll get an exception similar to the following at runtime:

```
[<CTRentalProperty 0x1322> valueForKey:] this class is not key value
coding-compliant for the key age
```

Luckily, the conventions are straightforward and easy to satisfy while developing your own custom classes. To support querying a property for its current value via the `valueForKey:` message, an object must provide at least one of the following (in order of preference by the framework):

- A getter method named `getKey` or `key`
- A getter method named `isKey` (typically used for properties of type `BOOL`)
- An instance variable named `_key` or `_isKey`
- An instance variable named `key` or `isKey`

For the matching `setValue:forKey:` message to work correctly, the object must provide one of the following messages:

- A setter method named `setKey:`
- An instance variable named `_key` or `_isKey`
- An instance variable named `key` or `isKey`

By conforming to these conventions, you enable the KVC framework to use Objective-C class metadata to determine how to interact with your objects at runtime. Although the framework supports instance variables beginning with an underscore (`_`) prefix, they should be avoided because the underscore prefix is typically reserved for internal use by Apple and the Objective-C runtime.

Most objects are KVC-compliant by default

It's important to realize that almost all Objective-C objects, including those in Foundation framework, such as `NSString`, are by default suitable for use with KVC techniques. This is because every object ultimately inherits from `NSObject`, and the `@synthesize` compiler feature discussed in chapter 5 produces suitable getter and setter methods automatically by default. In general, you have to go out of your way to produce an object that's incompatible with KVC.

11.1.1 Accessing properties via KVC

Now that you know how to make objects compatible with KVC, let's look at how the KVC mechanism can be used to get and update the current value of properties.

To query an object for the current value of one of its properties, you can use the `valueForKey:` message. A similar `setValue:forKey:` message allows you to specify a new value for a property:

```
NSLog(@"House address is: %@", [house valueForKey:@"address"]);
[house setValue:@"42 Campbell Street" forKey:@"address"];
```

Notice the prototype of the `valueForKey:` method indicates that it accepts an object (`id`). This means you can't pass in values for primitive data types such as `int`, `float`, or `BOOL` directly. You must box these via an instance of the `NSNumber` class. The KVC framework automatically unboxes these values as required, as shown here:

```
NSNumber *price = [house valueForKey:@"rentalPrice"];
NSLog(@"House rental price is $%f ", [price floatValue]);
[house setValue:[NSNumber numberWithFloat:19.25] forKey:@"rentalPrice"];
```

As a convenience measure, KVC also provides the `dictionaryWithValuesForKeys:` and `setValuesForKeysWithDictionary:` methods that enable you to pass in multiple KVC requests at once. In the case of the `dictionaryWithValuesForKeys:` message, you pass in an `NSArray` of property names and get back an `NSDictionary` of property name/value pairs.

11.1.2 Constructing key paths

KVC is much more interesting once you discover that as well as using simple keys that access properties directly on the object being queried, it's possible to use more advanced *key paths* to drill down into child objects. To enable this functionality, you use the `valueForKeyPath:` message instead of `valueForKey:`.

In the key path string, you separate the name of properties with periods, and as the KVC framework parses the string, it steps through your object hierarchy using KVC queries to reach the next level. As an example, the `CTRentalProperty` class has an address property that returns an `NSString`, and an `NSString` object has a `length` property. You can query the character length of a property's address via the following KVC key path query:

```
NSNumber *len = [house valueForKeyPath:@"address.length"];
NSLog(@"The address has %d characters in it", [len intValue]);
```

There is also an equivalent `setValue:forKeyPath:` that can be used to update the current value of a property of a child object, provided it has a suitable setter message.

11.1.3 Returning multiple values

While traversing key paths, you may encounter a property that represents zero or more values. For example, within chapter 6 you created a `Teacher` class which contained an `NSArray` property listing the classes taught by that teacher.

An array property represents a one-to-many relationship. Once an array is found in a key path, the remaining part is evaluated for each element in the array, and the results collated into an array.

As an example, the following query uses `valueForKeyPath:` to determine the classes taught by the list of teachers found within an array called `myTeachers`:

```
NSArray * courses = [myTeachers valueForKeyPath:@"classes"];
NSLog(@"Courses taught by teachers: %@", courses);
```

This query generates output similar to this:

```
Courses taught by teachers: (English, Spanish, Math, ARM, Immersive Gaming,
    Physical Computing)
```

KVC can use collections other than NSArray

Although the examples in this chapter heavily rely on properties that return an `NSArray` collection, the KVC infrastructure can operate with other collection-based data structures, even custom ones you develop yourself.

The key, yet again, is ensuring your class provides support for messages that conform to a specific convention to enable access to the collection. For more information, refer to the section titled "Collection Accessor Patterns for To-Many Properties" in the *Key-Value Coding Programming Guide*.

11.1.4 Aggregating and collating values

While working with one-to-many relationships in key paths, you may sometimes prefer an aggregate result that summarizes the raw data instead of an individual result for each value a key path matches. For example, rather than a list of tenants, you may just need the age of the oldest tenant, the average age of all tenants, or a list of all unique last names.

Key-value paths can perform such calculations by introducing a collection operator into the key path using the @ prefix. In practice, most key paths consist of two parts; see table 11.1.

Table 11.1 Common key path collection operators that aggregate and summarize items in a collection

Operator	Description
@avg	Converts the key path to the right of the operator to a collection of doubles and returns the average value
@count	Returns the number of objects in the key path to the left of the operator
@max	Converts the key path to the right of the operator to a collection of doubles and returns the maximum value
@min	Converts the key path to the right of the operator to a collection of doubles and returns the minimum value
@sum	Converts the key path to the right of the operator to a collection of doubles and returns their sum
@distinctUnionOfObjects	Returns an array containing the distinct objects in the property specified by the key path to the right of the operator

As an example, the following code snippet uses the @count operator to determine the number of rental properties located in the rentalProperties array:

```
NSNumber *count = [rentalProperties valueForKeyPath:@"@count"];
NSLog(@"There are %d rental properties for available", [count intValue]);
```

Internally, KVC queries a key path as normal, but when it detects an aggregate function, it creates a loop that iterates over all matching elements and performs the required calculation. The neat thing is that this process is all hidden from you: you express in a string the result you need. For example, the following two queries return the maximum and average length of all rental property addresses:

```
NSNumber *avg = [rentalProperties valueForKeyPath:@"@avg.address.length"];
NSNumber *max = [rentalProeprties valueForKeyPath:@"@max.address.length"];
```

Another thing you might need is a set of unique values assigned to a given property, such as a list of all unique rental prices. You may be tempted to generate the list via the KVC query:

```
NSArray *rentalPrices = [rentalProperties valueForKeyPath:@"rentalPrice"];
```

This query, however, would result in an array with duplicates if two or more rental properties were rented for the same monetary figure. Using the `@distinctUnionOfObjects` aggregate function resolves this situation:

```
NSArray *rentalPrices =
    [rentalProperties valueForKeyPath:@"@distinctUnionOfObjects.rentalPrice"];
```

The `@distinctUnionOfObjects` operator causes the KVC framework to walk through the array of objects to be returned and filter out duplicates, leaving only unique objects.

11.2 Handling special cases

While using KVC, you may encounter corner cases that can't occur with more traditional property getter and setter access code. Some errors that an Objective-C compiler would normally warn about during compilation will instead be detected at only runtime.

11.2.1 Handling unknown keys

One problem with using strings to specify properties is that there's nothing stopping you from writing a KVC query such as the following

```
id value = [house valueForKeyPath:@"advert"];
```

The `CTRentalProperty` class doesn't contain a property called `advert`, but the Objective-C compiler can't flag this as an error or warning at compile time because all it sees is a valid string constant. The compiler is unaware of how the string will be interpreted or utilized by the KVC framework during runtime.

At runtime, however, attempting to access such as key path will result in the following exception:

```
[<CTRentalProperty 0x1322> valueForKeyPath:] this class is not key value coding-compliant for the key advert]
```

The exception indicates that the KVC infrastructure can't determine how to access a property named `advert` on the current `CTRentalProperty` object (at least according to the rules outlined earlier in this chapter).

Sometimes, however, you may want an object to respond to getting or setting values for key(s) that don't physically exist on the object or at least don't conform to the rules outlined previously. You can make the `CTRentalProperty` object respond to the `advert` key, even though it doesn't contain a property by that name, by overriding the `setValue:forUndefinedKey:` and `valueForUndefinedKey:` messages in the `CTRentalProperty` class, as shown in the following listing.

Listing 11.1 `setValue:forUndefinedKey:` and `valueForUndefinedKey:`

```
- (void)setValue:(id)value forUndefinedKey:(NSString *)key {
    if ([key isEqualToString:@"advert"]) {
        NSArray *bits = [value componentsSeparatedByString:@" "];
        self.rentalPrice = [[bits objectAtIndex:0] floatValue];
        self.address = [bits objectAtIndex:1];
    } else {
```



```

        [super setValue:value forKey:key];
    }
}

- (id)valueForKey:(NSString *)key {
    if ([key isEqualToString:@"advert"]) {
        return [NSString stringWithFormat:@"$%f, %@",
            rentalPrice, address];
    } else {
        [super valueForKey:key];
    }
}
}

```

An important point to consider about custom `setValue:forUndefinedKey:` and `valueForKey:forUndefinedKey:` message implementations is that, internally, they should access setter methods or properties rather than their backing ivars directly. If you access or update an instance variable directly, Key-Value Observing won't always correctly detect changes in a key's value and hence won't notify interested observers. In listing 11.1, therefore, it's important that you refer to `rentalPrice` and `address` as `self.rentalPrice` and `self.address` because, instead of updating the instance variable, this syntax (or the alternative `[self rentalPrice]`, `[self address]`) will invoke the matching setter methods giving KVO a chance to observe the change.

With the additional method implementations in listing 11.1, the `CTRentalProperty` object now appears to have an additional property called `advert` that returns a string in the format "`$price, address`"—for example, "`$175, 46 Coleburn Street`". Following is an example of how this property would be accessed:

```

NSLog(@"Rental price is: %@", [house valueForKey:@"rentalPrice"]);
NSLog(@"Address is: %@", [house valueForKey:@"address"]);

NSString *advert = [house valueForKey:@"advert"];
NSLog(@"Advert for rental property is '%@'", advert);

```

This is not a very realistic implementation of the `setValue:forUndefinedKey:` and `valueForKey:forUndefinedKey:` messages. It would have perhaps been easier, more efficient, and cleaner to add a physical property named `advert` to the `CTRentalProperty` class. A more realistic implementation could use something like an `NSMutableDictionary` to enable a `CTRentalProperty` object to store arbitrary metadata about a rental property.

11.2.2 Handling nil values

A similar problem can occur when a `nil` value is passed to `setValue:forKey:`. As an example, what happens if you execute the following statement?

```
[house setValue:nil forKey:@"rentalPrice"];
```

Because the `rentalPrice` property of the `CTRentalProperty` class is a primitive datatype (float), it can't store the value `nil`. In this case, the default behavior is for the KVC infrastructure to throw an exception, as follows:

```
[<CTRentalProperty 0x123232> setNilValueForKey]: Could not set nil as the
value for the key rentalPrice.
```

In some cases, this is less than desirable. It's possible to override how nil values (typically used to represent the absence of a value) are handled. In the example of `CTRentalProperty`'s `rentalPrice` property, a more sensible approach may be to set the `rentalPrice` value to 0 whenever nil is received. You can do so by overriding another message called `setNilValueForKey:`, as demonstrated here:

```
- (void)setNilValueForKey:(NSString *)key {
    if ([key isEqualToString:@"rentalPrice"]) {
        rentalPrice = 0;
    } else {
        [super setNilValueForKey:key];
    }
}
```

With this implementation of `setNilValueForKey:`, attempting to use KVC to set the `rentalPrice` property to nil instead causes the property to be updated to the value 0.

11.3 Filtering and matching with predicates

Now that you know how to reference and access object properties via string-based key paths, it may not surprise you that a layer of other technologies is built on top of this foundation.

Often, when provided with an object, you'll want to compare it against a certain condition and perform different actions based on the result of the comparison. Similarly, if you have a collection of data, such as an `NSArray` or `NSSet`, you may want to use a specific set of criteria to filter, or select, subsets of the data.

These forms of expressions are commonly called *predicate conditions* and are represented in the Foundation framework by the `NSPredicate` class, which internally uses KVC, and key paths in particular.

11.3.1 Evaluating a predicate

Suppose you want to quickly determine if a `CTRentalProperty` object's rental price is greater than \$500. Mathematically, this can be expressed by the following predicate expression:

```
rentalPrice > 500
```

You can programmatically perform this check with an Objective-C statement such as the following:

```
BOOL conditionMet = (house.rentalPrice > 500);
```

The condition is hardcoded into the source code and can't easily change during runtime or in response to user input. In such a case, the `NSPredicate` class decouples things in a manner similar to what KVC does for getter and setter operations. The example condition could also be evaluated by the following code snippet, which uses the `NSPredicate` class:

```
NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"rentalPrice > 500"];
BOOL conditionMet = [predicate evaluateWithObject:house];
```

Here the predicate expression "rentalPrice > 500" is expressed as a string, which opens up a number of opportunities for it to be generated or changed at runtime. Once an NSPredicate object is created with the required expression, you can utilize its evaluateWithObject: message to compare a specified object against the predicate expression and determine if the object is a match or not.

NSPredicate is used in a number of frameworks as a way to enable clients to pass in application-specific query or filter conditions.

11.3.2 Filtering a collection

Now that you can express a generic predicate condition, you can use it for a number of purposes. A common task is to want to compare a predicate against each object in a collection, such as an NSArray. If you had an array of CTrentalProperty objects called allProperties, for example, you could determine which ones had a rental price greater than \$500 by implementing a code snippet similar to the following:

```
NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"rentalPrice > 500"];

NSArray *housesThatMatched =
    [allProperties filteredArrayUsingPredicate:predicate];

for (CTrentalProperty *house in housesThatMatched) {
    NSLog(@"%@ matches", house.address);
}
```

Once you have an NSPredicate expressing your desired condition, NSArray's filteredArrayUsingPredicate: message can be used to evaluate the predicate against each object in the array and return a new array containing only those objects that match the condition. If instead you have a mutable array, a similar filterUsingPredicate: message allows you to remove any objects that don't match the predicate from an existing array rather than creating a new copy.

11.3.3 Expressing your predicate condition

The simplest form of predicate condition is one that compares a property key path against another value, commonly a constant. An example of this is the rentalPrice > 500 predicate expression demonstrated previously. Other common comparison operators are listed in table 11.2.

Table 11.2 Common comparison operators for use with NSPredicate expressions. Notice that a number of operators have alternative symbols.

Operator	Description	Example
==, =	Equal to	rentalPrice == 350
!=, <>	Not equal to	rentalPrice <> 350
>	Greater than	rentalPrice > 350

Table 11.2 Common comparison operators for use with `NSPredicate` expressions. Notice that a number of operators have alternative symbols. (continued)

Operator	Description	Example
<code>>=, =></code>	Greater than or equal to	<code>rentalPrice >= 350</code>
<code><</code>	Less than	<code>rentalPrice < 350</code>
<code><=, =<</code>	Less than or equal to	<code>rentalPrice <= 350</code>
<code>BETWEEN</code>		<code>rentalPrice BETWEEN{ 400, 1000 }</code>
<code>TRUEPREDICATE</code>	Always evaluates to true	<code>TRUEPREDICATE</code>
<code>FALSEPREDICATE</code>	Always evaluates to false	<code>FALSEPREDICATE</code>

In a predicate expression, numeric constants can be expressed as normal (7, 1.2343, and so on), while string constants are enclosed in single or double quotation marks. Boolean values are expressed as `true` and `false`, but unlike most expression languages, a single constant such as `true` or `false` isn't a valid expression. If you want an `NSPredicate` to always evaluate to `true` or to `false`, you must instead use the keywords `TRUEPREDICATE` or `FALSEPREDICATE` respectively (or use a more complex expression such as `"true = true"`).

It's even possible for predicate expressions to involve calculations. Although somewhat overly complex for the task, another way to express the predicate of rental prices greater than \$500 would be as follows:

```
rentalPrice > 5 * 100
```

Such expressions often don't make a lot of sense, especially when the complete expression is stored as a compile-time string in source code. As you'll discover later, though, they can be helpful when predicates are built at runtime based on user input.

11.3.4 More complex conditions

`NSPredicate` also supports a full range of conditional operators designed for use with properties consisting of string-based content, as shown in table 11.3.

It's also possible to make compound predicates by using the `AND (&&)`, `OR (||)`, and `NOT (!)` operators as well as parentheses. As an example, the following code sample returns a list of rental properties that rent for less than \$500 or are located in Sumner.

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"rentalPrice < 500 || address ENDSWITH 'Sumner'"];

NSArray *housesThatMatched =
    [allProperties filteredArrayUsingPredicate:predicate];

for (CTRentalProperty *house in housesThatMatched) {
    NSLog(@"%@ matches", house.address);
}
```

Table 11.3 String operators available to NSPredicate-based predicate expressions. Regular expressions are a powerful way to validate string values against patterns, such as "valid email address" or "valid phone number".

Operator	Description	Example
BEGINSWITH	Does string begin with the specified prefix?	address BEGINSWITH '17'
ENDSWITH	Does string end with the specified postfix?	address ENDSWITH 'Street'
CONTAINS	Does string contain the specified string somewhere?	address CONTAINS 'egg'
LIKE	Does string match the specified pattern? A ? will match any character, while * matches 0 or more.	address LIKE '?? Kipling*'
MATCHES	Does string match the specified regular expression? In the example to the right, this matches any string that starts with a 4 or a 7.	address MATCHES '[47].*'

Similar to a number of comparison operators that have alternative symbols, the compound predicates all have two valid forms. Parentheses can be utilized to ensure evaluation order is correct in the case of more complex expressions, especially those that include calculations.

Another operator worth mentioning, especially if your object contains enumerated data types such as `CTRentalProperty`'s `propertyType` property, is the `IN` operator. This allows you to simplify the syntax for comparing a value against a number of possible values. For example, if you want to match rental properties that are `TownHouses` or `Units`, you can use the following `NSPredicate` expression:

```
propertyType IN { 0, 1 }
```

The values `0` and `1` match the enumerated types `TownHouse` and `Unit` respectively, meaning that this expression is logically equivalent to the much wordier expression:

```
(propertyType = 0) or (propertyType = 1)
```

11.3.5 Using key paths in predicate expressions

We've demonstrated the use of `NSPredicate` with expressions that compare single property values against hardcoded constants, but it's possible to use virtually any key path in an `NSPredicate` expression. This means you can develop some pretty complex filter conditions.

For example, the following predicate will match all rental properties with addresses consisting of at least 30 characters:

```
address.length > 30
```

It's also possible, using a range of additional operators, to perform set-based conditions whenever an array or similar collection is present in a key path. Table 11.4 lists some examples.

Table 11.4 Set-based operators that can be used in NSPredicate-based expressions when a collection of objects is present in a key path

Operator	Description	Example
ANY	Do one or more objects satisfy the key path expression?	ANY address.length > 50
ALL	Do all objects satisfy the key path expression?	ALL address.length < 20
NONE	Do all objects not match the key path expression?	NONE address.length < 20

11.3.6 Parameterizing and templating predicate expressions

So far, all examples of creating an NSPredicate object have used the `predicateWithFormat:` class message and consisted of a simple string. As the name of the message alludes to, the `predicateWithFormat:` message accepts NSString `stringWithFormat:~style` format arguments as well. For example, the following code snippet produces the predicate expression `rentalPrice > 500 || address ENDSWITH 'Sumner'`:

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"rentalPrice > %f || address ENDSWITH %@", 500.0f, @"Sumner"];
```

Notice that, unlike the original example in which the string `Sumner` had to be escaped with quotation marks, this example uses the `%@` format specifier, and NSPredicate is smart enough to introduce the required quote marks. The following code sample emits the final predicate expression generated by the previous code snippet:

```
NSLog(@"The predicate expression is %@", [predicate predicateFormat]);
```

This feature and others like it can be handy, and they help avoid common problems in generating dynamic expressions, such as improperly escaping strings that contain internal quotation marks.

If you parameterize a predicate expression because you want to dynamically adjust its values, perhaps by hooking up a numeric value to a UISlider control, the NSPredicate class can go one better. Altering the previous code snippet, you can use the following predicate template definition

```
NSPredicate *template = [NSPredicate predicateWithFormat:
    @"rentalPrice > $MINPRICE || address ENDSWITH $SUBURB"];
```

where names beginning with a dollar sign indicate placeholders in the templates where you can substitute different values at runtime. If you attempt to use this template predicate directly, an `NSInvalidArgumentException` is thrown, indicating that you haven't provided values for `MINPRICE` and `SUBURB`. To specify these values, you use the `predicateWithSubstitutionVariables:` message, passing in an `NSDictionary` of variable name/value pairs, as follows:

```

NSDictionary *variables = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:500],
    @"MINPRICE",
    @"Summer",
    @"SUBURB",
    nil];

NSPredicate *predicate = [template
    predicateWithSubstitutionVariables:variables];

```

From this point on, the predicate can be used as demonstrated in the previous sections.

11.4 *Sample application*

To demonstrate in a real-world situation the power of NSPredicate-based filtering and KVC, let's update the Rental Manager application to allow the user to filter the list of available rental properties to show only those properties matching a set of conditions specified by the user.

Replace the contents of RootViewController.h with the following listing.

Listing 11.2 RootViewController.h updated to store a filtered set of rental properties

```

@interface RootViewController : UITableViewController<UIAlertViewDelegate> {
    NSDictionary *cityMappings;
    NSArray *allProperties;
    NSArray *filteredProperties;
}

@end

```

Make sure you also replace the contents of RootViewController.m with the following listing.

Listing 11.3 RootViewController.m is updated

```

#import "RootViewController.h"
#import "CTRentalProperty.h"

@implementation RootViewController

- (void)viewDidLoad {
    [super viewDidLoad];

    NSString *path = [[NSBundle mainBundle]
        pathForResource:@"CityMappings"
        ofType:@"plist"];
    cityMappings = [[NSDictionary alloc] initWithContentsOfFile:path];

    allProperties = [[NSArray alloc] initWithObjects:
        [CTRentalProperty rentalPropertyOfType:TownHouse
            rentingFor:420.0f
            atAddress:@"13 Waverly Crescent, Summer"],
        [CTRentalProperty rentalPropertyOfType:Unit
            rentingFor:365.0f
            atAddress:@"74 Roberson Lane, Christchurch"],
        [CTRentalProperty rentalPropertyOfType:Unit

```

Load city
image names

Create array
for all
properties

```

        rentingFor:275.9f
        atAddress:@"17 Kipling Street, Riccarton"],
[CTRentalProperty rentalPropertyOfType:Mansion
    rentingFor:1500.0f
    atAddress:@"4 Everglade Ridge, Sumner"],
[CTRentalProperty rentalPropertyOfType:Mansion
    rentingFor:2000.0f
    atAddress:@"19 Islington Road, Clifton"],
    nil];
filteredProperties = [[NSMutableArray alloc]
                    initWithArray:allProperties];

self.navigationItem.rightBarButtonItem =
[[UIBarButtonItem alloc]
 initWithTitle:@"Filter"
 style:UIBarButtonItemStylePlain
 target:self
 action:@selector(filterList)];
}

- (void)filterList {
    UIAlertView *alert =
        [[UIAlertView alloc] initWithTitle:@"Filter"
         message:nil delegate:self
         cancelButtonTitle:@"Cancel"
         otherButtonTitles:nil];

    [alert addButtonWithTitle:@"All"];
    [alert addButtonWithTitle:@"Properties on Roads"];
    [alert addButtonWithTitle:@"Less than $300pw"];
    [alert addButtonWithTitle:@"Between $250 and $450pw"];

    [alert show];
    [alert release];
}

- (void)alertView:(UIAlertView *)alertView
  clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex != 0)
    {
        NSPredicate *predicate;

        switch (buttonIndex) {
            case 1:
                predicate = [NSPredicate
                    predicateWithFormat:@"TRUEPREDICATE"];
                break;
            case 2:
                predicate = [NSPredicate
                    predicateWithFormat:@"address CONTAINS 'Road'"];
                break;
            case 3:
                predicate = [NSPredicate
                    predicateWithFormat:@"rentalPrice < 300"];
                break;
            case 4:

```

← Create array for filtered properties

← Add "Filter" button

← Create alert for filtered options

← Check user doesn't click Cancel

← Create filter predicate


```

        predicate = [NSPredicate
            predicateWithFormat:
                @"rentalPrice BETWEEN { 250, 450 }"];
        break;
    }

    [filteredProperties release];
    filteredProperties = [[allProperties
        filteredArrayUsingPredicate:predicate] retain];

    [self.tableView reloadData];
}
}
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return [filteredProperties count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:CellIdentifier] autorelease];
    }

    CTrentalProperty *property =
        [filteredProperties objectAtIndex:indexPath.row];

    int indexOfComma = [property.address rangeOfString:@","].location;
    NSString *address = [property.address
        substringToIndex:indexOfComma];
    NSString *city = [property.address
        substringFromIndex:indexOfComma + 2];

    cell.textLabel.text = address;

    NSString *imageName = [cityMappings objectForKey:city];
    cell.imageView.image = [UIImage imageNamed:imageName];

    cell.detailTextLabel.text =
        [NSString stringWithFormat:@"Rents for $%0.2f per week",
            property.rentalPrice];

    return cell;
}

- (void)dealloc {
    [cityMappings release];
    [allProperties release];
    [filteredProperties release];

    [super dealloc];
}

@end

```

Save new filtered values

Get property from filteredProperties

Get and display correct city image

The core of the logic occurs in the `alertView:clickedButtonAtIndex:` method. This method is invoked when the user taps the right-hand button on the navigation bar and selects a desired filter condition. The method first creates an `NSPredicate` instance detailing which properties should still be displayed in the `UITableView`.

This predicate then filters the array of all rental properties (`allProperties`) using the predicate to come up with a new array (`filteredProperties`) consisting of the subset of rental properties that meet the specified conditions. The `UITableView` is then requested to reload using this second array to display the filtered set of rental properties to the user.

11.5 Summary

KVC- and `NSPredicate`-based logic is a powerful way to query, filter, analyze, and access data from an in-memory object-based data model. And this is just the tip of the iceberg of KVC's true power.

The real power of KVC begins to shine when you realize the level of abstraction it can provide. When we discussed key-path operators such as `@min`, `@max`, and `@distinct-UnionOfObjects`, you may have noticed that you expressed, in a simple string format, what operation was required, then left it up to the framework to determine how to iterate over the data model and any memory management issues or temporary data structures required to produce the results.

In chapter 12 you discover that all the concepts discussed in this chapter are transferrable for use with Core Data-based object models. Although the programming model is equivalent and familiar, behind the scenes, the implementation couldn't be more different. In the case of Core Data, KVC converts key paths and `NSPredicate`-based filter expressions into Structured Query Language (SQL) queries that are executed at the database level.

As a programmer, you for the most part don't care. You worry about the creation of the required filter conditions and leave it up to the framework to determine how best to implement the required logic.

Next up in chapter 12, you'll learn all about Core Data and how it can help you in the creation of robust iOS applications.

12

Reading and writing application data

This chapter covers

- Core Data basics
- Data models
- The `NSFetchedResultsController` class
- Data validation and performance

How useful would the Contacts application be if all your contacts were gone when you quit the application? Or how useful would the Notes application be if all your notes were lost when you turned off your iPhone? Not very useful at all. What is the one thing that makes these and most other applications useful? It's data persistence: the permanent storage of data that "survives" the closing of an application or even the restart or shutdown of the device.

Data persistence is needed in almost every application in one way or another. In a game, you might need to save high scores, user data, or game progress. In a productivity application, you want to save, edit, and delete to-do list items. Or in an RSS reader application, you want to save which feeds the user subscribes to. How do you persist data on the iPhone?

When the iPhone was first introduced, Apple included support for SQLite. SQLite is a lightweight, embeddable database system that can be accessed using the

Structured Query Language (SQL). It was the primary method for persisting (or storing) data on the iPhone. It got the job done, but it required a lot of overhead during development. To use SQLite, you had to do everything “by hand”: You had to create and initialize the database. You had to write SQL code for all CRUD (create, read, update, and delete) operations. You had to manually set up and manage relationships. And as a good object-oriented developer, you had to put the data you got from the database into Objective-C data model objects and take data from those objects again when you wanted to build a database query that updated, created, or deleted data. On top of that, you had to manage primary and foreign keys, and so on and so forth. You get the picture. It was no fun at all to write all that code.

Fortunately, Apple realized that too and brought the Core Data framework to the iPhone with the 3.0 software development kit (SDK). Core Data can be compared to ActiveRecord (if you know Ruby on Rails) or Hibernate (if you’re a Java developer). It handles all the low-level nitty-gritty details of data persistence for you: you don’t have to write SQL code, you don’t have to map between data and data objects, and you don’t have to handle relationships (it won’t fix your marriage, though).

In this chapter, we take a high-level look at all the different moving parts of Core Data and then get our hands dirty by building a simple Core Data application. Please keep in mind that Core Data is an advanced topic, and it would take a whole book to cover everything there is to know about it. But this chapter teaches you everything you need to know to leverage Core Data for all your most common data persistence needs.

12.1 Core Data history

Nearly every application has a data model and model objects (people, invoices, products, departments—you name it). And nearly every application has to save some or all of that data to disk. Functionality that is needed by a substantial number of applications and solutions to problems that developers face repeatedly calls for a framework. That’s why Apple put this data modeling and persistence functionality into a framework and called it Core Data. It was first introduced in 2005 with Mac OS X (10.4) Tiger. Core Data is the *M* in MVC (model-view-controller). It takes care of the complete model layer, including the relationships among the model objects, and makes it extremely easy to save an object graph to and load it from disk.

Before we dive into using Core Data, let’s look into the objects that make Core Data work.

12.1.1 What does Core Data do?

Core Data isn’t just a wrapper for database access. In most cases, however, Core Data uses a SQLite database to store the data on disk. On Mac OS X, developers usually also choose SQLite when working with Core Data, but the option to save the data in XML format is available too, which shows that Core Data isn’t simply a database access library. To explain what’s going on under the hood, we take a brief look at SQLite because it’ll make some of the underpinnings easier to understand.

In this book we've talked extensively about objects as the building blocks of applications. You've seen objects as containing instance variables with types such as `NSString`, `NSNumber`, `NSArray`, and so on. Traditionally developers would create a database table to hold the attributes of these instance variables. A database table can be thought of as a spreadsheet. Each row represents the data needed to make a specific instance of an object, and each column represents one of the instance variables. If you wanted to store four `Person` objects, like the ones you made for chapter 6, the table might look like this.

ID (INT)	Name (String)	Age (INT)	Gender (INT)
1	"Jimmy Miller"	16	0
2	"Brian Shedlock"	22	0
3	"Caitlin Acker"	28	1
4	"Collin Ruffenach"	35	0

Each of the four rows represents a `Person` object, and the four columns represent the instance variables: `ID` as a unique identifier of the object, `Name` as a string, `Age` as an integer, and `Gender` as an integer. You'd have to write translations for the integers in the database access class, with 0 corresponding to `Male` and 1 corresponding to `Female`. Before Core Data, if you wanted to store `Person` objects in a database, you'd create a similar table. Then you'd create a class called something like `DBController`. `DBController` would know the location of the `SQLite` database in the file directories of the application. It would have methods for accessing the database, as shown in the following listing.

Listing 12.1 Example database access methods (`DBController`)

```
- (void)addPersonToDatabase:(Person *)aPerson;
- (Person *)personWithName:(NSString *)aName
- (NSArray *)peopleWithAge:(int)anAge;
- (void)removePeopleWithGender:(Gender)aGender;
- (BOOL)updatePersonWithName:(NSString *)aName toAge:(int)anAge;
```

Any time an action in the application required interaction with the database, these methods had to be used. For a large application with complex objects, `DBController` classes were burdensome. They added a lot of code to a project just to facilitate storage.

Core Data automates the creation of all of these access methods and hides the guts of the database access from the developer. It greatly reduces the amount of code required by a project and improves the access time for most database queries. Unless you're storing a small preference, such as something that can fit easily with a plist, Apple recommends Core Data over a manual `SQLite` implementation.

In Xcode you can create a new project configured to use Core Data as the storage mechanism, which involves importing the `CoreData.framework`. Projects configured to use Core Data also create the Xcode data model that's used to define what your data looks like, also known as its schema.

12.2 Core Data objects

The functionality of Core Data is achieved through four distinct objects that compose what is known as the *Core Data stack*. These are the four major objects that make up Core Data's functionality. We investigate the purpose of each in the next several sections. The singular point of access is through the managed object context, discussed next, so if you're less concerned with the nuts and bolts of the system, you don't need to be too concerned about the persistent store coordinator and the persistent object store.

12.2.1 Managed object context

`NSManagedObjectContext` is the object that you work with to add, access, modify, or delete objects. When you access a single element out of a managed object context, an `NSManagedObject` or a subclass of it is returned. In the `Person` example, a returned `NSManagedObject` could be a `Person` object. You aren't required to create subclasses for every entity, though. You can also work directly with `NSManagedObject` instances, as you'll see in this chapter's demo application.

When you're working with an `NSManagedObject` retrieved from your managed object context, a good way to think of how the logic works is to compare it to a text file: if you open a text file, change a whole bunch of the text, add some, remove some, nothing matters unless and until you save it. So when you retrieve an `NSManagedObject` from your managed object context, the changes you make to it are applied to the database only when you tell the managed object context to save it. We get into the details of these activities later in this chapter, but the important point is to know that by using the managed object context, you can add, retrieve, and save `NSManagedObjects` or your custom subclasses.

12.2.2 Persistent store coordinator

In the stack, you work exclusively with the managed object context, and while a complex application may have several managed object contexts, all Core Data-based projects have only a single store coordinator. The persistent store coordinator sits between a managed object context and a persistent object store. You can think of it as the key holder to the databases. The managed object context must ask the persistent object coordinator if the requested access modification or deletion is allowed. It's important that a storage controller contains lots of checks to validate actions by its user, which is why you use only a single persistent store coordinator. If more than one coordinator exists, conflicting access to the database may occur. The persistent store coordinator manages all of the logic necessary to safely access and modify database tables.

12.2.3 *Managed object model*

The managed object model is an object connected to the persistent store coordinator. The persistent store coordinator uses the managed object model to understand the data it's pulling out of the persistent object store. The model is the actual definition of the data you plan to use with Core Data. It's like a blueprint. Rather than manually making classes for the object you plan to store, you define a managed object model and either work straight with the instances of `NSManagedObject` that Core Data populates for you or let Xcode generate the object classes for you (subclasses of `NSManagedObject`). Xcode provides a special user interface for developers to create and manage their models. Later in the chapter you'll create a model, have it generate Objective-C classes for your objects, and use them through a persistent object store.

12.2.4 *Persistent object store*

At the bottom of the Core Data stack are the persistent object stores. A persistent object store bridges the gap between the objects in a managed object context and a file on disk. There are different types of persistent stores, but on the iOS the `NSSQLiteStoreType` is usually used. A Core Data application can have several persistent object stores, but in iOS applications, you commonly use only one.

12.3 *Core Data resources*

To take advantage of the Core Data technologies, you need to have certain files in your project. These files and their calls rely on the Core Data framework. As discussed in chapters 6 and 7, frameworks are what put the magic in the iPhone SDK. They expose the precreated classes that you take advantage of to make your applications do all the cool things you need them to do. By default, when you create a project, only `UIKit.framework` and `Foundation.framework` are included. There are two ways to configure a project to use Core Data. You can manually add `CoreData.framework` to your project and set everything up by hand, or you can let Xcode do the work for you. Xcode allows you to create projects that utilize Core Data for its storage mechanism. It generates `CoreData.framework` and an Xcode data model, which is the primary file you work with to generate your Core Data models.

12.3.1 *Core Data entities*

Core Data provides an extremely simple interface to use native Objective-C classes as the input and output to a persistent store (mostly a database). In Core Data's terms, a class is an *entity*. Entities are the building blocks of Core Data. They represent Core Data-friendly custom objects—essentially `NSObject`s that Core Data creates, stores, and modifies so that the objects are stored in a database and easy to query. An entity, by default, is a subclass of `NSManagedObject`. `NSManagedObject` is a robust class capable of providing all the functionality most Core Data entity declarations require. Sometimes, however, you need to subclass `NSManagedObject` to add the methods your application specifically requires. Apple promotes the subclassing of this method but is

careful to note that you should not override the core behavior methods. Following are the methods of `NSManagedObject` that should not be overridden:

```
- (id)primitiveValueForKey:(NSString *)key
- (void)setPrimitiveValue:(id)value forKey:(NSString *)key
- (BOOL)isEqual:(id)anObject
- (NSUInteger)hash
- (Class)superclass
- (Class)class
- (id)self
- (NSZone *)zone
- (BOOL)isProxy
- (BOOL)isKindOfClass:(Class)aClass
- (BOOL)isMemberOfClass:(Class)aClass
- (BOOL)conformsToProtocol:(Protocol *)aProtocol
- (BOOL)respondsToSelector:(SEL)aSelector
- (id)retain
- (oneway void)release
- (id)autorelease
- (NSUInteger)retainCount
- (NSManagedObjectContext *)managedObjectContext
- (NSEntityDescription *)entity
- (NSManagedObjectID *)objectID
- (BOOL)isInserted
- (BOOL)isUpdated
- (BOOL)isDeleted
- (BOOL)isFault
```

In most cases, when using Core Data, you won't need to subclass or worry about any of these issues. But if you do, make sure to leave these methods unaltered so that Core Data's functionality isn't impacted.

12.3.2 Core Data attributes

An entity is composed of attributes and relationships. An attribute is a simple type, such as a string or an integer. Table 12.1 lists Core Data's simple attribute types that transform into the listed Objective-C objects.

Table 12.1 Simple and native Core Data object types

Simple type	Objective-C type
Int (16, 32, 64)	NSNumber
Decimal	NSNumber
Float	NSNumber
String	NSString
BOOL	NSNumber
Date	NSDate
Binary	NSData

Many objects can be composed solely of these simple types. Defining them in an entity tells Core Data what the data type of each column for the database table will be. Later you can create usable Objective-C classes out of these entities. Core Data generates the class files that represent these entities and hides all of the nasty work usually required to synthesize a native object in and out of a database table. Besides a type, certain attributes can have other parameters defined. Number values (integers, floats, and so on) can have maximums and minimums defined. Strings can have maximum and minimum lengths specified, along with attached regular expressions. Finally, all attributes, no matter what the type, can be defined as optional or mandatory and be given default values. This is useful for modeling information such as a checkout for a store, because during that process, you don't want to allow the creation of an "order" object without access to all the information you need. There's still a problem, however. Most objects require some combination of simple and complex types. To handle this problem, you can define relationships for Core Data entities.

12.3.3 *Core Data relationships*

Relationships are the connections among entities—a class's complex instance variables. A relationship points to another entity defined in your Xcode data model. When defining a relationship, you must also define the kind of association the object has to the class. It can be a one-to-one or a one-to-many relationship. An element with a one-to-many relationship is represented by a mutable set in the entry. With one-to-many relationships, much like with attribute definitions, you can also assign minimums and maximums for the set composing the relationship. A simple example is a task list that tracks multiple tasks assigned to multiple people. As a companion to your Rental Manager application, to help busy agents manage their schedules and tasks, you'll build an iPhone application that can do the job, including checking off tasks when they're completed. Let's make these objects using Core Data's tools.

12.4 *Building the PocketTasks application*

You start by creating a new Window-based project that uses Core Data for storage (check the Use Core Data for Storage check box when creating the project). Let's call it PocketTasks. Xcode automatically creates a `PocketTasks.xcdatamodel` where you'll define all of your entities. One quick note about the listings in this chapter: for brevity's sake, releasing instance variables in the `dealloc` methods of the different classes you build isn't always explicitly mentioned. You'll find them in the downloadable source code for this chapter, though.

12.4.1 *Examining the Xcode Core Data template*

Before we look at the data model file, let's quickly take a peek at the `PocketTasksAppDelegate.m`. Notice the following three methods:

```
- (NSManagedObjectContext *)managedObjectContext
- (NSManagedObjectContext *)managedObjectContext
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
```

They take care of the Core Data stack. The `managedObjectModel` method creates a new instance of `NSManagedObjectModel` with the data model description file that you'll create in the next step. The `persistentStoreCoordinator` method takes care of accessing the underlying SQLite database file in the application's document directory and creating a new instance of `NSPersistentStoreCoordinator`, which uses it along with the `managedObjectModel`. The `managedObjectContext` method brings it all together by creating a new instance of `NSManagedObjectContext` backed by the `persistentStoreCoordinator`. In most use cases, you won't have to directly deal with the `persistentStoreCoordinator` and the `managedObjectModel` once it's set up here. The `managedObjectContext` is used more frequently, especially to load and save data.

Now that you have an idea about the underpinnings, let's define the data model of the PocketTasks application.

12.4.2 Building the data model

Clicking `PocketTasks.xcdatamodel` brings up Xcode's Data Modeling tool, which you use to define all the entities as well as the attributes and relationships you need.

To create a new entity, click the plus sign labeled `Add Entity` in the lower-left corner of the editor pane of the Xcode window. Call the entity "Person." To add first name and last name attributes to the Person entity, click the + sign in the lower-left corner of the Attributes list and type in the name of the attribute. Call the first attribute "firstName" and, from the Type list, select its datatype to be String. Because all Person objects should have a firstName attribute specified, you should also open the Data Model Inspector (`Cmd-Option-3` or `View > Utilities > Data Model Inspector`) and uncheck the `Optional` check box. Create a second attribute with the same settings and call it "lastName." You've just created your first entity model definition!

But you're not quite done yet: as you've probably guessed, you need a second entity for the tasks. Add another entity, call it "Task," and give it two attributes called "name" of type String and "isDone" of type Boolean. The `isDone` attribute should be optional, the name attribute mandatory (so you have to uncheck the checkmark next to `Optional`).

The Xcode data model editor can also show the data model graphically. In the bottom-right corner of the main editor pane, you should see a toggle button labeled `Editor Style`, which allows toggling between table and graph styles. The graph view is shown in figure 12.1. This graph represents the same data model but displays the data in a manner that may be more familiar and comfortable to database developers.

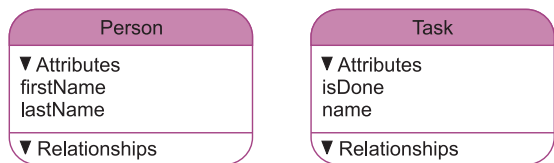


Figure 12.1 Xcode's Data Modeling tool shows a visual representation of the entities in your data model.

12.4.3 Defining the relationships

As the Person and Task models stand right now, they won't be of much use. You want to be able to add multiple tasks to a person. That means you must create a relationship between both of your entities: a task "belongs to" a person, and a person "has many" tasks.

With the Data Modeling tool in the Table Editor style, select the Person entity and click the + button in the Relationships section. Call the relationship "tasks," and select the Task entity as its destination. In the Data Model Inspector pane, next to Plural, check the To-Many Relationship box to tell Core Data that the Person entity can have more than one task. Finally, set the Delete Rule to Cascade. Cascade causes all the tasks that belong to a given Person entity to be deleted when the Person entity is deleted. Take, for example, a person called Peter. Peter has three tasks: go shopping, clean the car, and learn Core Data. When you delete Peter, those three tasks are deleted as well. The other two settings do the following:

- *Nullify*—The child entities (in this case, the tasks) won't be deleted, but they won't "belong" to the deleted Person entity anymore. They'll be orphans.
- *Deny*—Core Data won't allow a parent entity to be deleted as long as it still has children. In this case, Peter can't be deleted if he still has three tasks.
- *No Action*—Leave the destination objects as they are. That means they'll most likely point to a parent entity that no longer exists. In this case, the three tasks would still belong to Peter even though he was deleted—a situation you generally want to avoid.

You've specified that a person has many tasks; now you must specify that a task belongs to a person. Select the Task entity and add a new relationship called "person." Uncheck Optional because you don't want to be able to create unassigned tasks. The destination entity is obviously Person. The Inverse Relationship field lets you select the corresponding relationship in the destination entity (in this case, the tasks relationship of the Person entity). Why is that important? For data integrity, you always want to connect both sides of a relationship: you must always be able to access a person object's tasks, and you must always be able to access a task object's person. That way, you can change a task's assignment from Peter to Mary, and Core Data knows it must update the task's person to Mary and update Peter's and Mary's lists of tasks. With inverse relationships, you can always be sure your data stays consistent and Core Data does The Right Thing. Never forget to set up the inverse relationship.

Leave the Delete Rule at Nullify because you don't want a person to be deleted just because one of their tasks is deleted, but don't want a finished task to stay connected to the person before it's deleted.

Your tasks relationship should look like figure 12.2, and your person relationship should look like figure 12.3.

You're done with the data model. Make sure to save it before moving on to the next step.

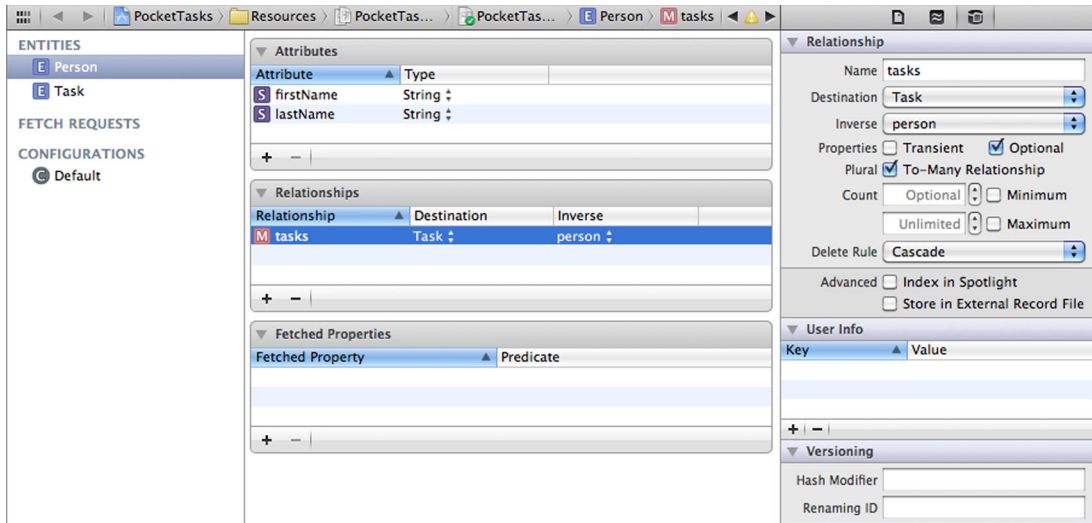


Figure 12.2 The tasks relationship for the Person entity. Relationships can be optional and point to many entities (for example, one person can have many tasks). Notice the inverse relationship that points back to the person.

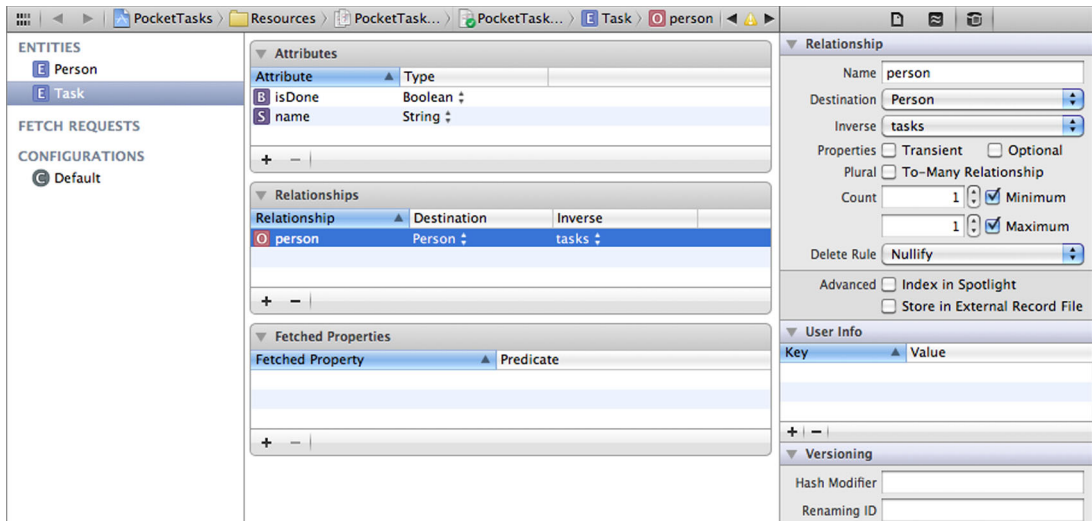


Figure 12.3 The person relationship for the Task entity. It isn't optional (a task has to belong to a person), and it points to only one person, not to many.

12.4.4 Creating Person entities in pure code

Now that the Core Data stack and the data model are set up, let's start using them! Instead of focusing too much on building a pretty UI, you first add some data programmatically and also read data programmatically and output it to the Console. You'll make it pretty later.

Open `PocketTasksAppDelegate.h` and add the two methods from the following listing.

Listing 12.2 Method declarations in PocketTasksAppDelegate.h

```
- (void)createSampleData;
- (void)dumpDataToConsole;
```

Now switch to `PocketTasksAppDelegate.m` and implement the methods from the following listing.

Listing 12.3 Method implementations in PocketTasksAppDelegate.m

```
- (void)createSampleData {
    NSArray *peopleToAdd =
        [NSArray arrayWithObjects:
         [NSArray arrayWithObjects:@"Peter", @"Pan", nil],
         [NSArray arrayWithObjects:@"Bob", @"Dylan", nil],
         [NSArray arrayWithObjects:@"Weird Al", @"Yankovic", nil],
         nil];

    for (NSArray *names in peopleToAdd) {
        NSManagedObject *newPerson =
            [NSEntityDescription
             insertNewObjectForEntityForName:@"Person"
             inManagedObjectContext:[self managedObjectContext]];

        [newPerson setValue:[names objectAtIndex:0] forKey:@"firstName"];
        [newPerson setValue:[names objectAtIndex:1] forKey:@"lastName"];

        NSLog(@"Creating %@ %@", [names objectAtIndex:0],
              [names objectAtIndex:1]);
    }

    NSError *error = nil;
    if (![self managedObjectContext] save:&error) {
        NSLog(@"Error saving the managedObjectContext: %@", error);
    } else {
        NSLog(@"managedObjectContext successfully saved!");
    }
}

- (void)dumpDataToConsole {
    // we'll implement this later
}
```

Once your application is finished launching, you call the `createSampleData` method by making `application:didFinishLaunchingWithOptions:` look like the following listing.

Listing 12.4 Calling the method to create sample data in PocketTasksAppDelegate.m

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    [self createSampleData];
    [self.window makeKeyAndVisible];
    return YES;
}
```

When you build and run this, you'll see only a white screen on your iPhone, but if you watch the Console, you'll see that three new person entities have been created. The

important line here is the call to `NSEntityDescription's insertNewObjectForEntityForName:inManagedObjectContext:` method. It creates a new `Person` managed object and returns it. You can set its values in the next two lines. Notice, however, that this doesn't save anything yet! Only the call to `save:` on `managedObjectContext` saves the data.

For some extra fun, comment the line that sets either the first or the last name and build and run again. Notice in the Console that Core Data refuses to save your data because you defined that both the first and last names are required.

12.4.5 Fetching Person entities in pure code

Naturally, you want access to the data you've saved (otherwise, this whole thing would be pretty pointless, wouldn't it?). Listing 12.5 shows the implementation of the `dumpDataToConsole` method. And it does just that: fetches all `Person` entities ordered by the last name and outputs them to the Console. The ordering is accomplished by an array of `NSSortDescriptors` (just one, in this case). An instance of `NSSortDescriptor` describes how objects should be ordered by setting the name of the property and, optionally, a method or block that should be used to compare objects and by specifying whether the order should be ascending or descending. The method `setSortDescriptors:` of `NSFetchRequest` takes an array of `NSSortDescriptors` because you might want to order by last name and then first name, for example.

Listing 12.5 Dumping all `Person` entities to the Console in `PocketTasksAppDelegate.m`

```
- (void)dumpDataToConsole {
    NSManagedObjectContext *moc = [self managedObjectContext];
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:[NSEntityDescription entityForName:@"Person"
                    inManagedObjectContext:moc]];
    // Tell the request that the people should be sorted by their last name
    [request setSortDescriptors:[NSArray arrayWithObject:
        [NSSortDescriptor sortDescriptorWithKey:@"lastName"
        ascending:YES]]];
    NSError *error = nil;
    NSArray *people = [moc executeFetchRequest:request error:&error];
    [request release];

    if (error) {
        NSLog(@"Error fetching the person entities: %@", error);
    } else {
        for (NSManagedObject *person in people) {
            NSLog(@"Found: %@ %@", [person valueForKey:@"firstName"],
                [person valueForKey:@"lastName"]);
        }
    }
}
```

Finally, you just need to change the `application:didFinishLaunchingWithOptions:` method to call the `dumpDataToConsole` method instead of the `createSampleData` method, as in the following listing.

Listing 12.6 Changes to `application:didFinishLaunchingWithOptions:`

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    [self dumpDataToConsole];
    [self.window makeKeyAndVisible];

    return YES;
}
```

Build this and run it. You should see the following output in the Console:

```
Found: Bob Dylan
Found: Peter Pan
Found: Weird Al Yankovic
```

You created an instance of `NSFetchRequest` and set it up to fetch the `Person` entities and told it sort the array of results by last name. If you're familiar with database programming, you can imagine `NSFetchRequest` to be your `SQL SELECT` statement: you can tell it what kind of data you want, how it should be sorted, and possibly also how it should be filtered (“only people whose last name is ‘Smith’”).

You can see that both creating and fetching data with Core Data isn't too hard. Now that you're sure your data model works and the Core Data stack is set up correctly, you can move on to fleshing out a usable application.

12.4.6 Adding a master `TableView`

Being able to create only the same three people over and over is unlikely to meet users' needs. Let's fix that and build a UI that allows users to add, edit, and delete people. We won't go over every line of code needed to build this UI; instead we focus on the parts that are relevant to Core Data. You can find the complete source code online.

In Xcode select `File > New > New File....` In the sheet that appears select `iOS > Cocoa Touch > UIViewController subclass` and click `Next`. Then select `UITableViewController` as the class to subclass and uncheck `With XIB for User Interface`. Click `Next` and call it “`PeopleViewController`.” Open `PeopleViewController.h` and add one instance variable and a custom `init` method, as shown in the following listing.

Listing 12.7 The `PeopleViewController.h` file

```
#import <UIKit/UIKit.h>

@interface PeopleViewController : UITableViewController
    <NSFetchedResultsControllerDelegate> {
    NSFetchedResultsController *resultsController;
    NSManagedObjectContext *managedObjectContext;
}

- (id)initWithManagedObjectContext:(NSManagedObjectContext *)moc;

@end
```

The `NSFetchedResultsController` class makes it easy and memory efficient to fetch data with Core Data and display it in a table view. In most of your Core Data iOS applications, you'll want to use an `NSFetchedResultsController`, especially when you're dealing with large amounts of data.

Now switch to `PeopleViewController.m` and implement the methods shown in the following listing.

Listing 12.8 Methods to add to `PeopleViewController.m`

```
- (id)initWithManagedObjectContext:(NSManagedObjectContext *)moc {
    if ((self = [super initWithStyle:UITableViewStylePlain])) {
        managedObjectContext = [moc retain];

        NSFetchedRequest *request = [[NSFetchedRequest alloc] init];
        [request setEntity:[NSEntityDescription entityForName:@"Person"
            inManagedObjectContext:moc]];
        [request setSortDescriptors:[NSArray arrayWithObject:
            [NSSortDescriptor sortDescriptorWithKey:@"lastName"
            ascending:YES]]];

        resultsController = [[NSFetchedResultsController alloc]
            initWithFetchRequest:request
            managedObjectContext:moc
            sectionNameKeyPath:nil
            cacheName:nil];
        resultsController.delegate = self;

        [request release];

        NSError *error = nil;
        if (![resultsController performFetch:&error]) {
            NSLog(@"Error while performing fetch: %@", error);
        }

        return self;
    }
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [[resultsController sections] count];
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [[[resultsController sections]
        objectAtIndex:section] numberOfObjects];
}

- (void)configureCell:(UITableViewCell *)cell
    atIndexPath:(NSIndexPath *)indexPath {
    NSManagedObjectContext *person =
        [resultsController objectAtIndex:indexPath];
    cell.textLabel.text = [NSString stringWithFormat:@"%@ %@",
        [person valueForKey:@"firstName"],
        [person valueForKey:@"lastName"]];
}
```

Create NSFetchedRequest by lastName

Create NSFetchedResultsController

Performing the fetch request

Configure the table cell


```

        cell.detailTextLabel.text = [NSString stringWithFormat:@"%i tasks",
                                     [[person valueForKey:@"tasks"] count]];
    }

- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"PersonCell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                 initWithStyle:UITableViewCellStyleSubtitle
                 reuseIdentifier:CellIdentifier] autorelease];
        cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
    }

    [self configureCell:cell atIndexPath:indexPath]; ← Call configureCell

    return cell;
}

```

In the `init` method you set up the same `NSFetchRequest` you use in the `dumpData-ToConsole` method. Then you set up the `NSFetchedResultsController` with that `NSFetchRequest` and your model object context (don't worry about the `cacheName` for now). Finally you perform the fetch request.

You can guess what the `numberOfSectionsInTableView:` and the `tableView:numberOfRowsInSection` methods do. But it's good to note how easy `NSFetchedResultsController` makes it to implement these methods. In this example you have only one section, but if the data were sectioned and you provided an appropriate `sectionNameKeyPath` while setting up `NSFetchedResultsController`, you'd get all the section handling for free.

Finally, the `tableView:cellForRowAtIndexPath:` method sets up each cell that should be displayed: it calls `configureCell:atIndexPath:` to set the title to the full name of the person and the subtitle to the number of tasks that person has. Notice the use of the `tasks` relationship you created while setting up the data model. Why configure the cell in a separate method? So you can reuse the code when you need to update a cell and thus avoid duplicating code.

Make sure you also add `[managedObjectContext release];` to `PeopleViewController`'s `dealloc` method so you don't create any memory leaks.

All that's left to do before you try out `NSFetchedResultsController` is to once again change the `application:didFinishLaunchingWithOptions:` method in `PocketTasksAppDelegate.m`, as shown in the following listing.

Listing 12.9 Putting `PeopleViewController` to use in `PocketTasksAppDelegate.m`

```

- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    PeopleViewController *peopleVC =
        [[PeopleViewController alloc]

```

```

initWithManagedObjectContext:[self managedObjectContext]];
UINavigationController * navCtrl =
    [[UINavigationController alloc]
     initWithRootViewController:peopleVC];
[peopleVC release];
peopleVC = nil;

[self.window addSubview:[navCtrl view]];
[self.window makeKeyAndVisible];

return YES;
}

```

You also need to import `PeopleViewController.h`. But you're a pro now, so we won't tell you how to do that.

Next, build and run. If everything's set up correctly, you should see the result in figure 12.4.

12.4.7 Adding and deleting people

Displaying the sample people is quite nice, but adding and deleting them would be the cherry on top. Let's add that cherry.

Using listings 12.10 and 12.11, you first add a `PersonDetailViewController` by selecting `File > New > New File...` and creating a new `UIViewController` subclass with an XIB for the user interface. You add two `UITextField` IBOutlets, an ivar of the type `NSManagedObjectContext` for the new person and one for the managed object context, and an initializer that takes an `NSManagedObjectContext`. Next, you add the two text fields in Interface Builder and connect them to the outlets. Here's what the `PersonDetailViewController.h` file should look like.

Listing 12.10 `PersonDetailViewController.h`

```

#import <UIKit/UIKit.h>

@interface PersonDetailViewController : UIViewController {
    IBOutlet UITextField *firstNameTextField;
    IBOutlet UITextField *lastNameTextField;

    NSManagedObjectContext *managedObjectContext;
    NSObject *person;
}

- (id)initWithManagedObjectContext:(NSManagedObjectContext *)moc;

@end

```

The interesting part of the `PersonDetailViewController` is the `saveAndDismiss` method. It creates a new person entity and dismisses the modal view controller if the save was successful.

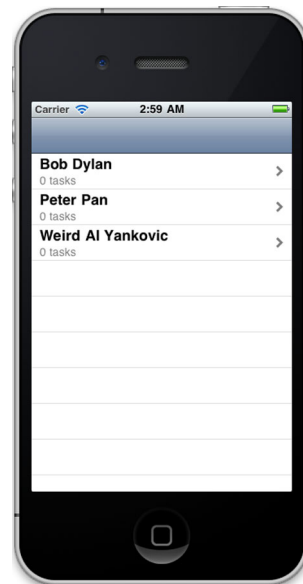


Figure 12.4 People in the PocketTasks application are loaded using Core Data and displayed in a tableview.

Listing 12.11 `init` and `saveAndDismiss` methods in `PersonDetailViewController.m`

```

- (id)initWithManagedObjectContext:(NSManagedObjectContext *)moc {
    if ((self =
        [super initWithNibName:@"PersonDetailViewController"
         bundle:nil])) {
        managedObjectContext = [moc retain];
    }
    return self;
}

- (void)saveAndDismiss {
    if (!person) {
        person = [NSEntityDescription
            insertNewObjectForEntityForName:@"Person"
            inManagedObjectContext:managedObjectContext];
    }

    if ([[firstNameTextField text] length] < 1 ||
        [[lastNameTextField text] length] < 1) {
        UIAlertView *alert =
            [[UIAlertView alloc] initWithTitle:@"Error"
             message:@"First and last name can't be empty."
             delegate:nil
             cancelButtonTitle:nil
             otherButtonTitles:@"OK", nil];
        [alert show];
        [alert release];
    } else {
        [person setValue:[firstNameTextField text] forKey:@"firstName"];
        [person setValue:[lastNameTextField text] forKey:@"lastName"];

        NSError *error = nil;
        if (![managedObjectContext save:&error]) {
            NSLog(@"Unresolved error %, %@", error, [error userInfo]);
        } else {
            [self dismissModalViewControllerAnimated:YES];
        }
    }
}

```

To use your new detail view controller, import it to `PeopleViewController.m` and add a slew of methods shown in the following listing.

Listing 12.12 Using `PersonDetailViewController` in `PeopleViewController.m`

```

- (void)viewDidLoad {
    [super viewDidLoad];
    self.navigationItem.leftBarButtonItem = self.editButtonItem;
    UIBarButtonItem *addButton = ← Add an Add button to right navigation bar
    [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
        target:self

```

**Add UINavigationController's
editButtonItem**

←

```

        action:@selector(addPerson)];
self.navigationItem.rightBarButtonItem = addButton;
[addButton release];
}

- (void)addPerson {
    PersonDetailViewController *detailController =
        [[PersonDetailViewController alloc]
         initWithManagedObjectContext:managedObjectContext];

    UINavigationController *controller =
        [[UINavigationController alloc]
         initWithRootViewController:detailController];

    detailController.navigationItem.rightBarButtonItem =
        [[[UIBarButtonItem alloc]
         initWithBarButtonSystemItem:UIBarButtonSystemItemSave
         target:detailController
         action:@selector(saveAndDismiss)]
         autorelease];

    [self presentViewController:controller animated:YES];

    [controller release];
    [detailController release];
}

- (void)tableView:(UITableView *)tableView
  commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
  forRowAtIndexPath:(NSIndexPath *)indexPath {

    if (editingStyle == UITableViewCellEditingStyleDelete) {
        [managedObjectContext deleteObject:[resultsController
                                           objectAtIndex:indexPath];

        NSError *error = nil;
        if (![managedObjectContext save:&error]) {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }
}

- (void)controllerWillChangeContent:
  (NSFetchedResultsController *)controller {
    [self.tableView beginUpdates];
}

- (void)controller:(NSFetchedResultsController *)controller
  didChangeObject:(id)anObject
  atIndexPath:(NSIndexPath *)indexPath
  forChangeType:(NSFetchedResultsControllerChangeType)type
  newIndexPath:(NSIndexPath *)newIndexPath {

    UITableView *tableView = self.tableView;

    switch(type) {
        case NSFetchedResultsControllerChangeInsert:
            [self.tableView

```

Create a PersonDetailViewController

Add a Save button

Present it modally

Delete the Core Data object

Set table view for changes

Content changes from results controller

```

        insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
        withRowAnimation:UITableViewRowAnimationFade];
        break;

    case NSFetchedResultsControllerDelete:
        [self.tableView
         deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationFade];
        break;

    case NSFetchedResultsControllerUpdate:
        [self.tableView
         reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationFade];
        break;

    case NSFetchedResultsControllerMove:
        [self.tableView
         deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationFade];
        [self.tableView
         insertRowsAtIndexPaths:[NSArray arrayWithObject:newIndexPath]
         withRowAnimation:UITableViewRowAnimationFade];
        break;
    }
}

- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller {
    [self.tableView endUpdates];    ← All changes complete, so animate them
}

```

What does all this code do? First, in the `viewDidLoad` method, you set up Edit and Add buttons. Notice that you get `editButtonItem` for free from `UIViewController`. In the `addPerson` method, you set up and present the view controller to add a new person. Next, in the `tableView:commitEditingStyle:forRowAtIndexPath:` method, you take care of deleting entries from the managed object context and save it afterwards. Finally, that really long method at the end, `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:`, gets called by the `NSFetchedResultsController` every time the result set changes in any way. Now when you add or delete a person from the managed object context, the fetched results controller calls this method, and it updates the table view because you set the instance of `PeopleViewController` as the delegate of the `NSFetchedResultsController` in listing 12.8.

Build and run your application: you should be able to add and delete people now.

12.4.8 *Managing tasks*

Okay, you can list, add, and delete people in your application. The only thing missing is the ability to add tasks to a person. To do that, you create another `UITableViewController` subclass and call it `TasksViewController`. It'll look basically the same as the `PeopleViewController` with only a few differences (you can copy and paste the code from `PeopleViewController` as a starting point): it'll have a person instance

variable of the type `NSManagedObject` and an `initWithPerson:` method that looks like the following listing.

Listing 12.13 Initializing `TasksViewController` with a person (`TasksViewController.m`)

```
- (id)initWithPerson:(NSManagedObject *)aPerson {
    if ((self = [super initWithStyle:UITableViewStylePlain])) {
        NSManagedObjectContext *moc = [aPerson managedObjectContext];
        person = [aPerson retain];

        NSFetchedResultsController *request = [[NSFetchRequest alloc] init];
        [request setEntity:[NSEntityDescription entityForName:@"Task"
            inManagedObjectContext:moc]];
        [request setSortDescriptors:
            [NSArray arrayWithObject:
                [NSSortDescriptor sortDescriptorWithKey:@"name"
                    ascending:YES]]];

        [request setPredicate:
            [NSPredicate predicateWithFormat:@"person == %@", person]];

        resultsController = [[NSFetchedResultsController alloc]
            initWithFetchRequest:request
            managedObjectContext:moc
            sectionNameKeyPath:nil
            cacheName:nil];

        resultsController.delegate = self;

        [request release];

        NSError *error = nil;
        if (![resultsController performFetch:&error]) {
            NSLog(@"Error while performing fetch: %@", error);
        }
    }

    return self;
}
```

Most of this method should look familiar, but you'll notice one difference: the use of a predicate. What does the predicate do? It's a filter statement, much like the `WHERE` clause in a SQL query. In this case you want to fetch only tasks that belong to a given person, so you create a predicate that uses the person relationship, defined in the `Task` entity, as a filter criteria to return only those tasks whose person relationship matches the specified person.

The following listing shows the rest of the interesting methods.

Listing 12.14 Necessary methods in `TasksViewController.m`

```
- (void)viewDidLoad {
    [super viewDidLoad];

    UIBarButtonItem *addButton = ◀ Add an Add button to
    [[UIBarButtonItem alloc]          right navigation bar
}
```

```

        initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
        target:self
        action:@selector(addTask)];
self.navigationItem.rightBarButtonItem = addButton;
[addButton release];
}

- (void)configureCell:(UITableViewCell *)cell
  atIndexPath:(NSIndexPath *)indexPath {
    NSManagedObject *task =
        [resultsController objectAtIndex:indexPath];
    cell.textLabel.text = [task valueForKey:@"name"];

    if ([[task valueForKey:@"isDone"] boolValue]) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    } else {
        cell.accessoryType = UITableViewCellAccessoryNone;
    }
}

- (void)addTask {
    NSManagedObject *task =
        [NSEntityDescription
         insertNewObjectForEntityForName:@"Task"
         inManagedObjectContext:[person managedObjectContext]];

    [task setValue:[NSString stringWithFormat:@"Task %i",
        [person valueForKey:@"tasks"] count] + 1]
        forKey:@"name"];
    [task setValue:[NSNumber numberWithInt:NO] forKey:@"isDone"];
    [task setValue:person forKey:@"person"];
    NSError *error = nil;
    if (![person managedObjectContext] save:&error) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    } else {
    }
}

- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
    NSManagedObject *task =
        [resultsController objectAtIndex:indexPath];
    if (![task valueForKey:@"isDone"] boolValue) {
        [task setValue:[NSNumber numberWithInt:YES] forKey:@"isDone"];

        NSError *error = nil;
        if (![task managedObjectContext] save:&error) {
            NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
            abort();
        }
    }
    [self.tableView
     reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
     withRowAnimation:UITableViewRowAnimationFade];
}
}

```

Create a new Task

Associate task with person

Save it

Display a checkmark for complete tasks

Update the task in the row

In `viewDidLoad` you add a button to add new tasks. The `configureCell:atIndexPath:` method configures a table cell to display the name of the task and to show a checkmark if the task is done. The `addTask` method adds a new task and sets its name to Task 1, Task 2, and so on. Feel free to add a user interface that lets the user enter a real name for a task. Notice that the person relationship of the task in this method is set to make the new task belong to the selected person. Also notice the use of `[NSNumber numberWithInt:NO]`. Why can't you just say `[task setValue:NO forKey:@"isDone"]`? Because you always have to provide an object for the value, never a primitive value. That's why you have to box an integer, double, float, or Boolean value in an instance of `NSNumber`.

Finally, the `tableView:didSelectRowAtIndexPath:` method marks a task as done by setting the value, saving the context, and calling `configureCell:atIndexPath:` to display the checkmark.

To make your new `TasksViewController` appear when you tap on a person, you have to go back to `PeopleViewController.m` once more. Using the code in listing 12.15, import `TasksViewController.h` and implement the `tableView:didSelectRowAtIndexPath:` method to create an instance of `TasksViewController` and push it onto the navigation controller stack.

Listing 12.15 `tableView:didSelectRowAtIndexPath:` in `PeopleViewController.m`

```
- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    TasksViewController *tasksVC =
        [[TasksViewController alloc]
         initWithPerson:[resultsController objectAtIndex:indexPath:indexPath]];
    [self.navigationController pushViewController:tasksVC animated:YES];
    [tasksVC release];
}
```

Now build and run the application. You should be able to add tasks and set them to done by tapping on them (see figure 12.5).

12.4.9 Using model objects

So far you've been using `NSManagedObject` to access your people and tasks. While this works just fine, it would be much nicer to be able to say `[person firstName]` (or `person.firstName`, the dot notation for properties) instead of `[person valueForKey:@"firstName"]`. You also need the full name of a person in several places in your application. Rather than putting the first and last names together every time you need the full name, you can more efficiently just say `[person fullName]`. Core Data provides an easy way to do so using model object classes, which are custom subclasses of `NSManagedObject`. You can add your own methods to model object classes and have nice accessor methods for all the properties and relationships.

To convert `PocketTasks` to use model object classes, you first let Xcode generate the appropriate classes for you. Select `File > New > New File...`, and then select

NSObject subclass from the iOS > Core Data section. Click Next and select the PocketTasks data model. Click Next again and put a checkmark next to Person and Task (see figure 12.6).

Finally, click Finish.

Build and run the application. Your project should still work just as before.

When you examine the source files for the Person and the Task classes, you'll find accessors for your properties and relationships. Let's add one method to the Person class. Open Person.h and add this line:

```
@property (readonly) NSString *fullName;
```

Next, open Person.m and add the implementation:

```
- (NSString *)fullName {
    return [NSString stringWithFormat:@"% %@ %@",
        self.firstName, self.lastName];
}
```

Now you have the code to put together the full name in a single place: in the model. You can see why it almost always makes sense to work with these NSObject subclasses in your Core Data project: it's easier to access the properties and relationships, it gives you the opportunity to add custom logic to your models, and it makes the code easier to read and maintain.

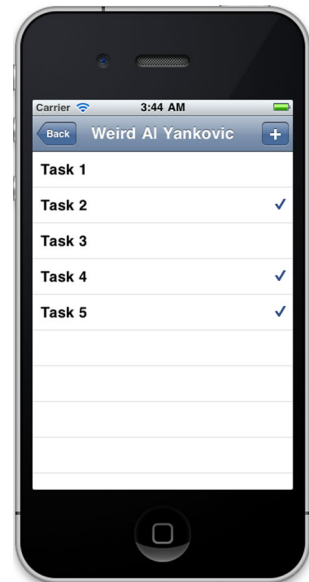


Figure 12.5 Tasks can be added and marked as done by tapping on them. Completed tasks have a checkmark on the right.

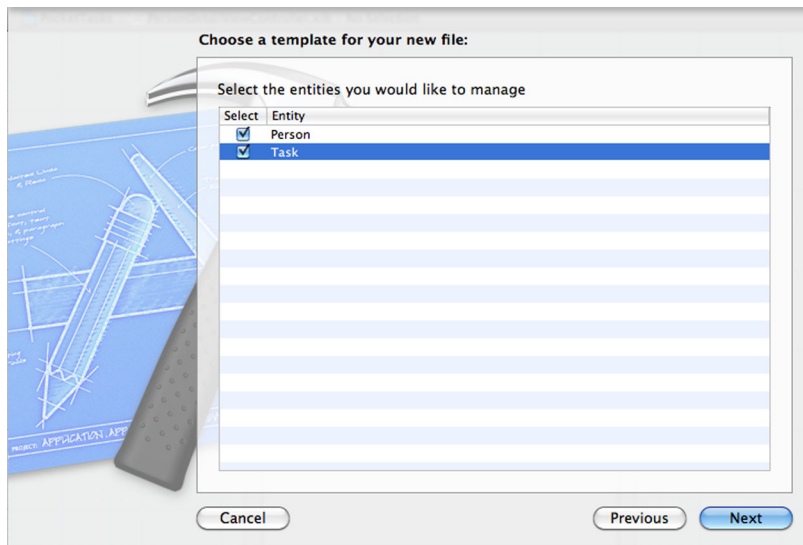


Figure 12.6 Xcode can automatically generate Objective-C classes for the entities in a data model.

How do you use your new model classes now? We look at the `addTask` method in `TasksViewController.m` together, and you'll be able to convert the rest of the application yourself. If you get stuck, check out the code for the completed project online.

First, make sure you import `Person.h` and `Task.h` and that you change the person instance variable type from `NSManagedObject` to `Person`. The new `addTask` method will look like the following listing.

Listing 12.16 Modifying `addTask` in `TasksViewController.m` to use `Person` class

```
- (void)addTask {
    Task *task = [NSEntityDescription
                 insertNewObjectForEntityForName:@"Task"
                 inManagedObjectContext:[person managedObjectContext]];
    task.name =
        [NSString stringWithFormat:@"Task %i", [person.tasks count] + 1];
    task.isDone = [NSNumber numberWithInt:NO];

    [person addTasksObject:task];

    NSError *error = nil;
    if (![person managedObjectContext] save:&error) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    } else {
    }
}
}
```

That looks much nicer, doesn't it? And notice that the `addTasksObject:` method of the `Person` class is used to add new tasks to the current person. That makes your code a lot more readable, and it's much clearer to see what's going on. It won't be a problem for you to change the rest of the code to use the new model classes.

12.5 Beyond the basics

You've built your first Core Data application. You know how to create a data model and how to fetch, save, and even filter and sort data. That's really all you need to know to use Core Data in your own applications. As mentioned earlier, however, Core Data is a complex framework, and there's still more to learn. It's beyond the scope of this chapter to cover Core Data in depth, but in the next three sections, we touch on a few important topics and provide further resources to point you in the right direction.

12.5.1 Changing the data model

Change is the one thing that's constant in every software project. The day will likely come when you'll have to add a property to an entity, add a new entity, or add a new relationship to your data model. When you do, you'll have to take some extra steps to ensure that your user's data will still work with the new version of your application. You won't change your data model—instead, you'll add a new one but keep the old version so it can read data that was saved with that version of your data model.

Data migration can be a complex task, but when the change to the data model is small (for example, adding a new property or making a non-optional property

optional), Core Data does most of the work for you. Such migrations are called *light-weight migrations*. Let's see what this looks like in practice: you'll add a new property—age—to the Person entity. To do that, you must first create a new version of your data model. Select the PocketTasks.xcodatamodel, then select Editor > Add Model Version from the menu. A new file called PocketTasks 2.xcdatamodel will appear in the PocketTasks.xcdatamodel directory. This new version should be used by your application from now on. For that to take effect, select PocketTasks.xcdatamodel and choose “PocketTasks 2” from the “Current” select box in the “Versioned Data Model” section of the first tab of the Utilities pane. The file will get a little green checkmark. Now add the new age property to the Person entity in the new version you just created. Make it optional, and set the type to Integer16. Save it and build and run. The application will crash and report errors in the Console: the saved data is incompatible with the new data model! You have to tell Core Data to automatically migrate it for you. Open PocketTasksAppDelegate.m and find the persistentStoreCoordinator method. Change it to look like the following listing.

Listing 12.17 Automatically migrate a data model in PocketTasksAppDelegate.m

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    if (__persistentStoreCoordinator != nil) {
        return __persistentStoreCoordinator;
    }

    NSURL *storeURL =
        [[self applicationDocumentsDirectory]
         URLByAppendingPathComponent:@"PocketTasks.sqlite"];

    NSDictionary *options =
        [NSDictionary dictionaryWithObjectsAndKeys:
         [NSNumber numberWithInt:YES],
         NSMigratePersistentStoresAutomaticallyOption,
         [NSNumber numberWithInt:YES],
         NSInferMappingModelAutomaticallyOption, nil];

    NSError *error = nil;
    __persistentStoreCoordinator =
        [[NSPersistentStoreCoordinator alloc]
         initWithManagedObjectModel:[self managedObjectModel]];

    if (![__persistentStoreCoordinator
         addPersistentStoreWithType:NSSQLiteStoreType
         configuration:nil
         URL:storeURL
         options:options
         error:&error])
    {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    return __persistentStoreCoordinator;
}
```

All you do is pass in two options that tell Core Data to try to automatically determine what has changed between the data models and to try to automatically migrate the data.

Build and run it again—everything should work.

If you can, you should try to avoid substantial changes to your data model, especially after you've shipped your application and people are already using it. Put some time and thought into your data model before you start coding.

If you need to perform a more complex migration, check out Apple's Core Data Model Versioning and Data Migration Programming Guide (<http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/CoreDataVersioning/Introduction/Introduction.html>).

12.5.2 Performance

Using `NSFetchedResultsController` is already one of the best things you can do regarding performance of your iOS Core Data application. There are a few more things to keep in mind, though. If you know that you'll always need access to a certain relationship of your entity, you should tell Core Data to fetch those child entities with the initial fetch request. Otherwise, Core Data would have to execute another fetch request for every entity. With 100 people, that would make 101 fetch requests (the initial one to fetch the people and one for each person to get the tasks). Using `NSFetchResult`'s `setRelationshipKeyPathsForPrefetching:` method avoids this overhead. In your example, to improve the performance when you fetch people, it would look like this:

```
[request setRelationshipKeyPathsForPrefetching: [NSArray  
    arrayWithObject:@"tasks"]];
```

This code loads the tasks right away along with the people.

Another performance concern is binary data: unless the data is very small (smaller than 100 kb), you shouldn't store it directly in the entity it belongs to. If the data is bigger than 100 kb, consider storing it in a separate entity and setting up a relationship between the data entity and the entity it belongs to (maybe a profile picture that belongs to a person). Anything bigger than 1 MB should not be stored in a Core Data entity but on disk as a file. Only the path to the file should be stored using Core Data.

More information about performance can be found in Apple's Core Data Programming Guide: <http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CoreData/Articles/cdPerformance.html>.

12.5.3 Error handling and validation

You undoubtedly noticed that every time you save the managed object context, you pass in a pointer to an `NSError` object. So far, you haven't really handled those errors. In a real application users have a right to be informed about anything that goes wrong with their data. So how do you handle errors in Core Data?

If you encounter an error from which you can't recover (can't write data to disk or something catastrophic like that), you should inform the user about it with an

UIAlertView and then use the `abort()` function to exit the application. Apple discourages the use of the `abort()` function and asks developers to display a message that tells the user to quit the application by pressing the Home button. While this works too, the `abort()` function has one advantage: it generates a crash log. The log can be extremely useful for tracking down the problem. It's up to you to decide how you'll handle fatal errors, but `abort()` isn't as bad as it's made out to be if you use it responsibly.

Validation errors must also be dealt with: a required value is missing, a string is too short, or a number is too large. Such errors can be recovered from, and you should tell the user about them and the user a chance to fix them.

Listing 12.18 shows one way to handle and display validation errors (for example, in `PersonDetailViewController`). Please note that you should adapt this method to your application and your user's needs. As it stands, the messages aren't quite user friendly but are definitely better than just saying "Saving failed."

Listing 12.18 One way to handle and display validation errors

```
- (void)displayValidationError:(NSError *)anError {
    if (anError &&
        [[anError domain] isEqualToString:@"NSCocoaErrorDomain"]) {
        NSArray *errors = nil;

        if ([anError code] == NSValidationMultipleErrorsError) {
            errors = [[anError userInfo] objectForKey:NSDetailedErrorsKey];
        } else {
            errors = [NSArray arrayWithObject:anError];
        }

        if (errors && [errors count] > 0) {
            NSString *messages = @"Reason(s):\n";
            for (NSError *error in errors) {
                NSString *entityName =
                    [[[error userInfo]
                     objectForKey:@"NSValidationErrorObject"]
                     entity]
                     name];
                NSString *attributeName =
                    [[error userInfo] objectForKey:@"NSValidationErrorKey"];
                NSString *msg;

                switch ([error code]) {
                    case NSManagedObjectValidationError:
                        msg = @"Generic validation error.";
                        break;
                    case NSValidationMissingMandatoryPropertyError:
                        msg = [NSString stringWithFormat:
                            @"The attribute '%@' mustn't be empty.",
                            attributeName];
                        break;
                    case NSValidationRelationshipLacksMinimumCountError:
                        msg = [NSString stringWithFormat:
                            @"The relationship '%@' doesn't have enough entries.",
```

Make sure it's a built-in error

Extract multiple errors; wrap single errors

Iterate over the errors

Get entity name

Get attribute name

Build appropriate error messages

```
        attributeName];
    break;
case NSValidationRelationshipExceedsMaximumCountError:
    msg = [NSString stringWithFormat:
        @"The relationship '%@" has too many entries.",
        attributeName];
    break;
case NSValidationRelationshipDeniedDeleteError:
    msg = [NSString stringWithFormat:
        @"To delete, the relationship '%@" must be empty.",
        attributeName];
    break;
case NSValidationNumberTooLargeError:
    msg = [NSString stringWithFormat:
        @"The number of the attribute '%@" is too large.",
        attributeName];
    break;
case NSValidationNumberTooSmallError:
    msg = [NSString stringWithFormat:
        @"The number of the attribute '%@" is too small.",
        attributeName];
    break;
case NSValidationDateTooLateError:
    msg = [NSString stringWithFormat:
        @"The date of the attribute '%@" is too late.",
        attributeName];
    break;
case NSValidationDateTooSoonError:
    msg = [NSString stringWithFormat:
        @"The date of the attribute '%@" is too soon.",
        attributeName];
    break;
case NSValidationInvalidDateError:
    msg = [NSString stringWithFormat:
        @"The date of the attribute '%@" is invalid.",
        attributeName];
    break;
case NSValidationStringTooLongError:
    msg = [NSString stringWithFormat:
        @"The text of the attribute '%@" is too long.",
        attributeName];
    break;
case NSValidationStringTooShortError:
    msg = [NSString stringWithFormat:
        @"The text of the attribute '%@" is too short.",
        attributeName];
    break;
case NSValidationStringPatternMatchingError:
    msg = [NSString stringWithFormat:
        @"The text of the attribute '%@" "
        "doesn't match the required pattern.",
        attributeName];
    break;
default:
    msg = [NSString stringWithFormat:
```


13

Blocks and Grand Central Dispatch

This chapter covers

- Understanding block syntax
- Handling memory management
- Using blocks in system frameworks
- Learning Grand Central Dispatch
- Performing work asynchronously

Have you ever been to New York City's Grand Central Terminal? It's a huge place with trains arriving and departing constantly. Imagine having to manage such an operation by hand—Which train should come in on which track? Which one should change tracks at what time? Which engine should be attached to which train?—and so on and so forth. You'd probably quit your job after 30 minutes and go back to programming!

Being a programmer, however, you might find yourself in a similar situation when you have to write a multithreaded application with different parts of your application being executed at the same time: you must make sure you have enough resources to run another thread; you must make sure that multiple threads can't manipulate the same data at the same time; you must somehow handle the case of one thread being dependent on other threads, and so on. It's a headache. And it's error prone, just like trying to run Grand Central Terminal by yourself.

Thankfully, Apple introduced a technology called Grand Central Dispatch (GCD) in Mac OS X 10.6 and brought it over to iOS devices in iOS 4. GCD dramatically simplifies multithreaded programming because it takes care of all the complicated heavy lifting. Along with GCD, Apple added a new language feature to C: blocks. Blocks are just that: blocks of code, or anonymous functions, if you will. GCD and blocks make for a very powerful duo. You can now write little pieces of code and hand them over to GCD to be executed in a parallel thread without any of the pain multithreaded programming normally causes. Parallel execution has never been easier, and with GCD there's no longer an excuse not to do it.

Before we dig any deeper into GCD, let's examine blocks: how to create them, how to run them, and what to watch out for.

13.1 *The syntax of blocks*

The syntax of blocks might scare you at first—it scared us! But we'll hold your hand as we decrypt and demystify it together, so don't fret.

Let's first look at a simple block example in the following listing.

Listing 13.1 Simple block example

```
int (^myMultiplier)(int, int) = ^int (int a, int b){
    return a * b;
};

int result = myMultiplier(7, 8);
```

We told you it would look a little scary. This listing creates a new block that takes two integers as arguments, returns an integer, and stores it in a variable called `myMultiplier`. Then the block gets executed, and the result of 56 gets stored in the integer variable called `result`.

The listing has two parts: the variable declaration and the block literal (the part after the equals sign). Let's look at both in depth (see figure 13.1).

You can think of a block variable declaration as a C function declaration because the syntax is almost identical. The difference is that the block name is enclosed in parentheses and has to be introduced by a caret (^). The following listing might clarify it even more.

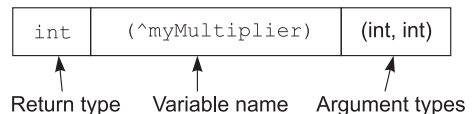


Figure 13.1 The syntax of a block variable declaration

Listing 13.2 Declaring a block variable and assigning to it

```
int (^myMultiplier) (int, int);

myMultiplier = ^int (int a, int b){
    return a * b;
};

myMultiplier(4, 2);
```

What if you want to declare multiple block variables of the same type (meaning taking the same arguments and having the same return type)? The following listing illustrates the wrong way and the right way to do it.

Listing 13.3 Using typedef to declare a block type for reuse

```
int (^myTimes2Multiplier) (int);
int (^myTimes5Multiplier) (int);
int (^myTimes10Multiplier) (int);

myTimes2Multiplier = ^(int a) { return a * 2; };
myTimes5Multiplier = ^(int a) { return a * 5; };
myTimes10Multiplier = ^(int a) { return a * 10; };

typedef int (^MultiplierBlock) (int);

MultiplierBlock myX2Multi = ^(int a) { return a * 2; };
MultiplierBlock myX5Multi = ^(int a) { return a * 5; };
MultiplierBlock myX10Multi = ^(int a) { return a * 10; };
```

The wrong way is to declare each block variable the long way. Although the code will compile and work fine, it isn't good style to do it this way, plus it might get difficult to maintain this kind of code later. The right way to use typedef is to define your own block type. It works exactly the same way as declaring a block variable, but this time the name in the parentheses is treated as a type name, not as a variable name. That way, you can declare as many blocks of the type `MultiplierBlock` as you like.

With the block variable declaration out of the way, let's look at the more interesting part, the part you'll use much more often: block literals. In the previous code examples, you saw a lot of block literals, but let's take a closer look at their syntax now (see figure 13.2).

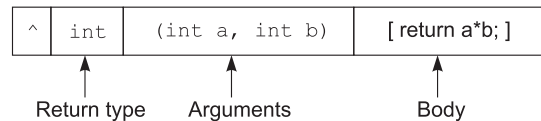


Figure 13.2 The syntax of a block literal

A block literal always starts with the caret (^), followed by the return type, the arguments, and finally the body—the actual code—in curly braces. Both the return type and the arguments are optional. If the return type is void, you can omit it altogether, and the same applies to arguments. The return type is also optional if it can be inferred automatically by the compiler through the `return` statement in the block's body. The next listing shows some valid long and short ways to write block literals.

Listing 13.4 Some long and short ways to write block literals

```
^void (void) { NSLog(@"Hello Block!"); };
^{ NSLog(@"Hello Block!"); };

^int { return 2001; };
^{ return 2001; };

^int (int a, int b) { return a + b; };
^(int a, int b) { return a + b; };
```

You see that it's convenient to omit unnecessary void arguments or return types in block literals. Note that if you try to run the code in listing 13.4, it won't do anything because you never actually execute these blocks. It's like declaring a list of functions that are never called. Blocks are executed the same way functions are: by adding a pair of opening and a closing parentheses to them. Normally, you'd assign a block literal to a variable and then execute the block using the variable (as you saw in listings 13.1 and 13.2). You can also execute block literals that don't take any arguments directly. The following listing shows both approaches.

Listing 13.5 Executing blocks

```
int (^myMultiplier)(int, int) = ^int (int a, int b){
    return a * b;
};

myMultiplier(7, 8);

^{ NSLog(@"Hello Block!"); }();
```

So far, blocks don't seem that different from functions—but they're very different in one special way.

13.1.1 *Blocks are closures*

Blocks have access to the variables that are available or defined in their lexical scope. What does that mean? Let's look at an example in the following listing.

Listing 13.6 Capturing variables

```
void (^myBlock) (void);

int year = 2525;

myBlock = ^{ NSLog(@"In the year %i", year); };
myBlock();
```

You can access the local variable `year` from inside the block. Technically, though, you're not accessing the variable. Instead, the variable is copied or frozen into the block by the time you create it. The following listing illustrates frozen variables.

Listing 13.7 Captured variables are frozen

```
void (^myBlock) (void);

int year = 2525;

myBlock = ^{ NSLog(@"In the year %i", year); };
myBlock();

year = 1984;

myBlock();
```

Changing the value of a variable after it's captured by a block doesn't affect the frozen copy in the block. But what if you want it to do that? Or what if you want to also

change the value from inside the block? Then you need to use the `__block` storage type, which practically marks the variables inside a block as mutable. The effect is that changes made to the variable outside the block are picked up inside the block, and vice versa (see the following listing).

Listing 13.8 The `__block` storage type

```
void (^myBlock) (void);

__block int year = 2525;
__block int runs = 0;

myBlock = ^{
    NSLog(@"In the year %i", year);
    runs++;
};

myBlock();

year = 1984;

myBlock();

NSLog(@"%i runs.", runs);
```

The interesting thing is that the local variables that are captured by a block live on even when the function or method they're defined in ends. That's a very powerful feature that we explore in more depth later in this chapter. For now, let's at least look at an example in the following listing.

Listing 13.9 Captured variables survive the end of a function

```
typedef void (^MyTestBlock) (void);

MyTestBlock createBlock() {
    int year = 2525;

    MyTestBlock myBlock = ^{
        NSLog(@"In the year %i", year);
    };

    return Block_copy(myBlock);
}

void runTheBlock() {
    MyTestBlock block = createBlock();

    block();

    Block_release(block);
}
```

Listing 13.9 clearly shows that the local variable `year` gets captured in the block and is still available in the block after the `createBlock` function returns. Also notice the use of `Block_copy` and `Block_release`, which brings us to our next topic.

13.1.2 *Blocks and memory management*

Blocks start their life on the stack, just like any other local variable in a function or method. If you want to use your block after the destruction of the scope in which it was declared (for example, after a function returns as in listing 13.9), you must use `Block_copy` to copy it to the heap. To avoid memory leaks, you must always use `Block_release` to release any block that you've copied with `Block_copy` when you don't need it anymore.

In Objective-C, blocks are also always Objective-C objects, so you can send them the familiar `copy` and `release` messages too.

What about objects that are captured by a block? In Objective-C all objects referenced inside a block are automatically sent a `retain` message. When the block gets released, all those objects are sent a `release` message. The only exceptions are objects with the `__block` storage type: they aren't automatically retained and released. When objects that are instance variables are referenced inside a block, the owning object instead of the instance object is sent a `retain` message. The following listing should make this clearer.

Listing 13.10 Automatic retain and release

```
typedef void (^SimpleBlock)(void);
@interface MyBlockTest : NSObject
{
    NSMutableArray *things;
}
- (void)runMemoryTest;
- (SimpleBlock)makeBlock;
@end
@implementation MyBlockTest
- (id)init {
    if ((self = [super init])) {
        things = [[NSMutableArray alloc] init];
        NSLog(@"1) retain count: %i", [self retainCount]);
    }
    return self;
}
- (SimpleBlock)makeBlock {
    __block MyBlockTest *mySelf = self;
    SimpleBlock block = ^{
        [things addObject:@"Mr. Horse"];
        NSLog(@"2) retain count: %i", [mySelf retainCount]);
    };
    return Block_copy(block);
}
- (void)dealloc {
    [things release];
}
```

← **SimpleBlock type definition**

← **ivar used inside of a block**

← **Pointer to MyTestBlock**

← **"things" ivar of current MyTestBlock**

← **Print retainCount with non-auto-retained reference**

```

    [super dealloc];
}
- (void)runMemoryTest {
    SimpleBlock block = [self makeBlock];
    block();
    Block_release(block);
    NSLog(@"3) retain count: %i", [self retainCount]);
}
@end

```

When you create an instance of the class `MyBlockTest` and run its `runMemoryTest` method, you'll see this result in the console:

- 1 retain count: 1
- 2 retain count: 2
- 3 retain count: 1

Let's examine what's happening here. You have an instance variable called `things`. In the `makeBlock` method, you create a reference to the current instance of `MyBlockTest` with the `__block` storage type. Why? You don't want the block to retain `self` because you use `self` in the `NSLog` statement inside the block, you want the block to retain `self` because you use one of its instance variables, `things`. Next, you reference `things` in the block and print the current `retainCount` of the object that owns `things` to the console. Finally, you return a copy of the block.

In the `runMemoryTest` method, you call the `makeBlock` method, run the returned block, and release it. Finally you print the `retainCount` again. This example demonstrates that your instance of `MyBlockTest` has been automatically retained because you used one of its instance variables—`things`—in a block that you copied to the heap. To make this even clearer, comment out the first line of the block (`[things addObject...]`) and run it again. You'll see that the retain count of the `MyBlockTest` instance is always 1. Because you're no longer referencing any of its instance variables inside the block, the instance of `MyBlockTest` is no longer automatically retained.

A final caveat you about working with blocks: A block literal (`^{...}`) is the memory address of a stack-local data structure representing the block. That means that the scope of that data structure is its enclosing statement (for example, a `for` loop or the body of an `if` statement). Why is that important to know? Because you'll enter a world of pain if you write code similar to the following listing.

Listing 13.11 Be careful about block literal scopes

```

void (^myBlock) (void);
if (true) {
    myBlock = ^{
        NSLog(@"I will die right away.");
    };
} else {

```

```

    myBlock = Block_copy(^{
        NSLog(@"I will live on.");
    });
}
myBlock();

```

Remember that block literals are valid only in the scope in which they are defined. If you want to use them outside of that scope, you must copy them to the heap with `Block_copy`. If you don't, your program will crash.

Now that you have a fundamental understanding of blocks, let's look at some common places where you'll encounter them in Cocoa Touch.

13.1.3 Block-based APIs in Apple's iOS frameworks

A great and growing number of Apple's framework classes take blocks as parameters, often greatly simplifying and reducing the amount of code you have to write. Typically, blocks are used as completion, error, and notification handlers, for sorting and enumeration, and for view animations and transitions.

In this section we look at a few simple but practical examples so you can get familiar with how blocks are used in the system frameworks.

The following listing shows how you can easily invoke a block for each line in a string.

Listing 13.12 Enumerating every line in a string with a block

```

NSString *string = @"Soylent\nGreen\nis\npeople";

[string enumerateLinesUsingBlock:
    ^(NSString *line, BOOL *stop) {
        NSLog(@"Line: %@", line);
    }];

```

You can surely guess what the output will look like. Pretty nifty, right?

Listing 13.13 demonstrates the use of two block-based APIs. The first one, `objectsPassingTest:`, invokes a given block for each object in a set. The block then returns either YES or NO for each object, and the method returns a new set consisting of all the objects that passed the test. The second one, `enumerateObjectsUsingBlock:`, invokes the given block once for every object in the set. It's basically equivalent to a for loop.

Listing 13.13 Filtering and enumeration using blocks

```

NSSet *set = [NSSet setWithObjects:@"a", @"b", @"cat",
    @"c", @"mouse", @"ox", @"d", nil];

NSSet *longStrings =
    [set objectsPassingTest:
        ^BOOL (id obj, BOOL *stop) {
            return [obj length] > 1;
        }];

```

```
[longStrings enumerateObjectsUsingBlock:
    ^(id obj, BOOL *stop) {
        NSLog(@"string: %@", obj);
    }];
```

Running the code in listing 13.13 will write the words `cat`, `mouse`, and `ox` to the console because they passed the test of being longer than one character.

The example in the following listing shows how to use a block as a notification handler.

Listing 13.14 Using a block as a notification handler

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[[[UIDevice currentDevice]
 beginGeneratingDeviceOrientationNotifications];

[nc addObserverForName:
 UIDeviceOrientationDidChangeNotification
 object:nil
 queue:[NSOperationQueue mainQueue]
 usingBlock:^(NSNotification *notif){
     UIDeviceOrientation orientation;
     orientation = [[UIDevice currentDevice] orientation];
     if (UIDeviceOrientationIsPortrait(orientation)) {
         NSLog(@"portrait");
     } else {
         NSLog(@"landscape");
     }
 }];
```

Listing 13.14 first gets a reference to the default notification center and then tells the current device instance to start sending notifications when the device's orientation has changed. Finally, a block is added to the notification center as an observer for the orientation change notification. Don't worry about the use of `NSOperationQueue` here—we cover that later. The great thing about using blocks as handlers for various events is that you have the code that handles the event right there. You don't have to search for some target method somewhere else in your code. Your code is much more readable, less cluttered, more concise, and easier to understand.

It's clear how versatile blocks are and that they will undoubtedly make your work easier. Next we talk about a very special area in which blocks really shine: asynchronous and parallel execution.

13.2 Performing work asynchronously

Performing work asynchronously means doing multiple things at the same time—or at least making it seem as if you do.

To illustrate, think of a big supermarket with 20 employees who can work the register but only a single register is open. What a disaster! The employees would fight over the register, and the customers would get furious because they have to wait in long lines. It's much better when the store has 20 cash registers open: 20 customers can be

helped at the same time, everything moves a lot quicker, customers don't get mad, and everyone has something to do. In your iOS app, you don't want to have only a single cash register open—your customers would get mad too. Certain tasks, such as downloading data from the internet, can take a long time. Performing such tasks synchronously—one after another—would block your application and render it unresponsive to the user until the tasks are done. You don't want that. Instead, you want your application to stay responsive at all times and to perform tasks—especially long-running ones—asynchronously, notifying your user or updating the UI once the task has finished. GCD in connection with blocks makes this extremely easy to do.

13.2.1 Meet GCD

GCD manages a pool of threads and safely runs blocks on those threads depending on how many system resources are available. You don't have to worry about any of the thread management: GCD does all the work for you.

To explain how GCD works, let's go back to our analogy of New York City's Grand Central Terminal. Everyone knows that trains are made up of cars and that they run on tracks. A train station often has multiple tracks so that multiple trains can enter and leave the station at the same time. In GCD, blocks are the cars that make up a train. And dispatch queues are the tracks that these trains run on. GCD comes with four prebuilt queues: three global queues with low, default, and high priority and one main queue that corresponds with your application's main thread. GCD also allows you to build your own dispatch queues and run blocks on them. Both the main dispatch queue and the dispatch queues that you create on your own are serial queues. The blocks you put on a serial queue are executed one after the other, first in, first out (FIFO). Blocks that you put on the same serial queue are guaranteed to never run concurrently, but blocks on different serial queues do run concurrently, just like multiple trains on different tracks can run parallel to each other.

The only exceptions are the three concurrent global queues: they do run blocks concurrently. They start them in the order they were added to the queue, but they don't wait for one block to finish before they start running the next one.

All this might still seem a bit confusing and theoretical, so let's look at some code.

13.2.2 GCD fundamentals

The following listing shows how to run an anonymous block on the default priority global queue.

Listing 13.15 Running a block on a global queue

```
dispatch_async(  
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),  
    ^{  
        NSLog(@"Hello GCD!");  
    });
```

The function `dispatch_async` takes two arguments: the queue the block should be run on and the block itself. `dispatch_get_global_queue` does just that: it returns one of the three global queues (low, default, or high priority). The second parameter should always be 0 because it's reserved for future use. The `dispatch_async` function returns right away and dispatches the block to GCD to be run on the given queue. Pretty easy, right?

You can also create your own serial dispatch queues, as shown in the following listing.

Listing 13.16 Creating your own serial dispatch queue

```
dispatch_queue_t queue;
queue = dispatch_queue_create("com.springenwerk.Test", NULL);

dispatch_async(queue, ^{
    NSLog(@"Hello from my own queue!");
});

dispatch_release(queue);
```

The function `dispatch_queue_create` takes a name and a second parameter, which should always be `NULL` because it's reserved for future use. To avoid naming conflicts, you should use your reverse domain name to name the serial dispatch queues you create. You then pass them to `dispatch_async` just like any other queue. Because dispatch queues are reference-counted objects, you have to release them to avoid memory leaks.

That's all the basics you need to get up and running with GCD. That wasn't too bad, was it?

It might still seem like dry theory to you, though, so let's build a small application that uses GCD and shows how all this works in real life.

Real estate agents like to show their clients pictures of beautiful homes in the area they're interested in. That's exactly what you'll build: an application called `RealEstateViewer` that lets you search for images of real estate in any location of your choice.

13.2.3 Building `RealEstateViewer`

Create a new Window-based application in Xcode and call it `RealEstateViewer`. Add a new `UITableViewController` subclass to your project (Cocoa Touch Class > `UIViewController` subclass with the `UITableViewController` subclass selected). Call it `ImageTableViewController`. Next, include it in your application delegate and set the window's view to the `ImageTableViewController`'s view, as shown in listings 13.17 and 13.18 (see figure 13.3).

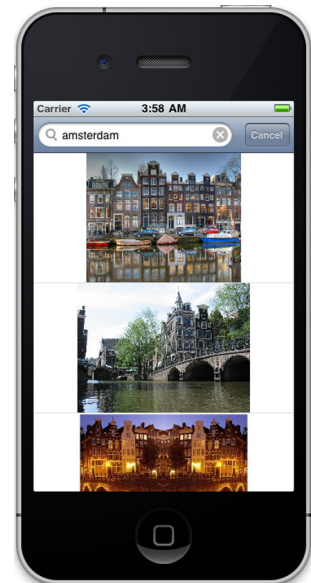


Figure 13.3 The finished `RealEstateViewer` application

Listing 13.17 `RealEstateViewerAppDelegate.h`

```
#import <UIKit/UIKit.h>
#import "ImageTableViewController.h"

@interface RealEstateViewerAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    ImageTableViewController *imageTableViewController;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;

@end
```

Be sure to also implement the following listing.

Listing 13.18 `RealEstateViewerAppDelegate.m`

```
#import "RealEstateViewerAppDelegate.h"

@implementation RealEstateViewerAppDelegate
@synthesize window;

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    imageTableViewController = [[ImageTableViewController alloc] init];

    [window addSubview:[imageTableViewController view]];
    [window makeKeyAndVisible];

    return YES;
}

- (void)dealloc {
    [imageTableViewController release];
    [window release];
    [super dealloc];
}

@end
```

That's all the boilerplate code you have to write. The rest of the action takes place inside the `ImageTableViewController`. Because you'll be talking to Google's Image Search API, which returns data in the JSON format, you must add Stig Brautaset's JSON framework to your application. That sounds more complicated than it is: just download it from <http://stig.github.com/json-framework/> and copy all the files from the `Classes` folder to your application's `Classes` folder (or you can take the files from the source code for this chapter).

Now you'll add a search bar to the top of the table, a few delegate methods, an ivar to hold your search results, and a little bit of code to perform the image search. The next two listings show you what your `ImageTableViewController` should look like at this point.

Listing 13.19 ImageTableViewController.h

```
#import <UIKit/UIKit.h>

@interface ImageTableViewController : UITableViewController
    <UISearchBarDelegate> {

    NSArray *results;
}

@property (nonatomic, retain) NSArray *results;

@end
```

Listing 13.20 ImageTableViewController.m

```
#import "ImageTableViewController.h"
#import "JSON.h"

@implementation ImageTableViewController
@synthesize results;

#pragma mark -
#pragma mark Initialization

- (id)initWithStyle:(UITableViewStyle)style {
    if ((self = [super initWithStyle:style])) {
        results = [NSArray array];

        UISearchBar *searchBar =
            [[UISearchBar alloc]
             initWithFrame:CGRectMake(0, 0,
                                     self.tableView.frame.size.width, 0)];
        searchBar.delegate = self;
        searchBar.showsCancelButton = YES;
        [searchBar sizeToFit];

        self.tableView.tableHeaderView = searchBar; ← Create UISearchBar;
        [searchBar release];                                     set as headerView

        self.tableView.rowHeight = 160;
    }
    return self;
}

#pragma mark -
#pragma mark UISearchBarDelegate methods

- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar {
    NSLog(@"Searching for: %@", searchBar.text);
    NSString *api = @"http://ajax.googleapis.com/ajax/"
        "services/search/images?v=1.0&rsz=large&q=";
    NSString *urlString =
        [NSString
         stringWithFormat:@"%real%20estate%20%@",
          api,
          [searchBar.text
           stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding]];
    NSURL *url = [NSURL URLWithString:urlString]; ← Create a URL
        with a search
        string
```

```

[NSThread sleepForTimeInterval:1.5];
NSData *data = [NSData dataWithContentsOfURL:url];
NSString *res = [[NSString alloc] initWithData:data
                encoding:NSUTF8StringEncoding];

self.results = [[[res JSONValue] objectForKey:@"responseData"]
                objectForKey:@"results"];

[res release];
[searchBar resignFirstResponder];
[self.tableView reloadData];
}

- (void)searchBarCancelButtonClicked:(UISearchBar *)searchBar {
    [searchBar resignFirstResponder];
}

#pragma mark -
#pragma mark Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [results count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier]
                autorelease];
    } else {
        for (UIView *view in cell.contentView.subviews) {
            [view removeFromSuperview];
        }
    }

    UIImage *image =
        [[results objectAtIndex:indexPath.row] objectForKey:@"image"];

    if (!image) {
        image = [UIImage imageWithData:
                [NSData dataWithContentsOfURL:
                 [NSURL URLWithString:
                  [[results objectAtIndex:indexPath.row]
                   objectForKey:@"unescapedUrl"]]]];
    }
}

```

Sleep for 1.5 seconds; fake slow network

Parse JSON data; assign to results ivar

Use blocking method to load network results

Attempt to get cached image for requested row

Load image with a blocking method

```

        [[results objectAtIndex:indexPath.row]
         setValue:image forKey:@"image"];
    }

    UIImageView *imageView =
        [[[UIImageView alloc] initWithImage:image] autorelease];

    imageView.contentMode = UIViewContentModeScaleAspectFit;
    imageView.autoresizingMask =
        UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight;
    imageView.frame = cell.contentView.frame;

    [cell.contentView addSubview:imageView];

    return cell;
}

#pragma mark -
#pragma mark Memory management

- (void)dealloc {
    [results release];
    [super dealloc];
}

@end

```

← Cache the image

Now build and run your application. You should have a fully functional real estate image searcher. But you'll notice immediately how terrible the user experience is: when you put in a search term, the whole application freezes for a few seconds, and when you scroll down, it freezes a couple of more times. Completely unacceptable! What's happening here? This code is doing a horribly wrong thing: executing long-running blocking tasks—getting the search results and downloading the images—on the main thread. The main thread must always be free to handle UI updates and incoming events. That's why you should never do anything “expensive” on the main thread. How can you use blocks and GCD to fix these problems? You can put both the code that queries the image search API and the code that downloads an image into blocks and then hand those blocks to GCD. GCD will execute them in a parallel thread and thus not block the main thread.

13.2.4 Making the image search asynchronous

The following listing shows what the GCD-based asynchronous image search looks like.

Listing 13.21 Asynchronous image search with GCD

```

- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar {
    NSLog(@"Searching for: %@", searchBar.text);
    NSString *api = @"http://ajax.googleapis.com/ajax/"
        "services/search/images?v=1.0&rsz=large&q=";
    NSString *urlString = [NSString
        stringWithFormat:@"%0real%20estate%20%@",
        api,
        [searchBar.text
        stringByAddingPercentEscapesUsingEncoding:

```

```

       :NSUTF8StringEncoding]];
NSURL *url = [NSURL URLWithString:urlString];

// get the global default priority queue
dispatch_queue_t defQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

void (^imageAPIBlock)(void);

imageAPIBlock = ^{
    [NSThread sleepForTimeInterval:1.5];

    NSData *data = [NSData dataWithContentsOfURL:url];

    NSString *res = [[NSString alloc]
        initWithData:data
        encoding:NSUTF8StringEncoding];

    NSArray *newResults =
        [[res valueForKey:@"responseData"]
        valueForKey:@"results"];

    [res release];

    dispatch_async(dispatch_get_main_queue(), ^{
        self.results = newResults;
        [self.tableView reloadData];
    });
};

dispatch_async(defQueue, imageAPIBlock);

[searchBar resignFirstResponder];
}

```

What does this code do? First it references one of the three global concurrent queues: the default priority queue. Then it declares a block that performs the network communication with the image search API and takes care of the JSON parsing. When it's done, it calls `dispatch_async` again (that's right, you can call `dispatch_async` from inside a block). The target is the main queue, which corresponds to the application's main thread (the one that takes care of the UI and events). You pass in an anonymous block that sets the new results and tells the tableview to reload. Why don't you do this right in the `imageAPIBlock`? For two reasons: First, UI elements should be updated only from the main thread. Second, an ugly race condition is prevented: imagine starting two searches in very quick succession. Because the three global queues execute blocks concurrently, it could happen that two blocks try to update the results array at the same time, which would most likely make your application crash. Because the main queue always waits for one block to be done before it executes the next, you can always be sure that only one block tries to update the results array at any given time.

When you run your application now, notice that the search works much more smoothly. But it still freezes for a short time and multiple times when you scroll. You still have to get the loading of the images off of the main thread. Let's do that next.

13.2.5 Making the image loading asynchronous

With listing 13.22, you change your `tableView:cellForRowAtIndexPath:` method quite a bit. You check whether you already have the requested image. If so, you just set up a `UIImageView` with it and return the cell. If not, you load the image in a block, and the block calls back to the main thread by dispatching another block to the main queue, which caches the image and tells the `tableView` to reload the cell for the affected row. That causes the `tableView:cellForRowAtIndexPath:` method to be called again, but this time it finds a cached image and is happy.

Listing 13.22 Asynchronous image loading with GCD

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    } else {
        for (UIView *view in cell.contentView.subviews){
            [view removeFromSuperview];
        }
    }

    UIImage *image =
        [[results objectAtIndex:indexPath.row] objectForKey:@"image"];

    if (!image) {
        void (^imageLoadingBlock)(void);

        UIActivityIndicatorView *spinner =
            [[UIActivityIndicatorView alloc]
             initWithActivityIndicatorStyle:
             UIActivityIndicatorViewStyleGray];

        spinner.autoresizingMask =
            UIViewAutoresizingFlexibleLeftMargin |
            UIViewAutoresizingFlexibleRightMargin |
            UIViewAutoresizingFlexibleTopMargin |
            UIViewAutoresizingFlexibleBottomMargin;

        spinner.contentMode = UIViewContentModeCenter;
        spinner.center = cell.contentView.center;
        [spinner startAnimating];

        [cell.contentView addSubview:spinner];
        [spinner release];

        imageLoadingBlock = ^{
            image = [UIImage imageWithData:
                [NSData dataWithContentsOfURL:

```

**Attempt to fetch cached image;
mark variable editable**

**Create variable
to hold a block**

**Create a spinner to add
to the table view cell**

**Create block to
load the image**


```

[NSURL URLWithString:
 [[results objectAtIndex:indexPath.row]
  objectForKey:@"unescapedUrl"]]];

[image retain];

dispatch_async(dispatch_get_main_queue(), ^{
  [[results objectAtIndex:indexPath.row]
   setValue:image
   forKey:@"image"];
  [image release];
  [spinner stopAnimating];

  // reload the affected row
  [self.tableView
   reloadRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
   withRowAnimation:NO];
});

dispatch_async(
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
  imageLoadingBlock);
} else {
  UIImageView *imageView =
    [[[UIImageView alloc] initWithImage:image] autorelease];

  imageView.contentMode = UIViewContentModeScaleAspectFit;
  imageView.autoresizingMask =
    UIViewAutoresizingFlexibleWidth |
    UIViewAutoresizingFlexibleHeight;
  imageView.frame = cell.contentView.frame;

  [cell.contentView addSubview:imageView];
}

return cell;
}

```

Dispatch anonymous block to main queue

Cache image

Reload affected row

Asynchronously dispatch image loading block

Use and display cached image

This code first checks whether a cell is available for reuse. If so, it removes all of the cell's subviews (the image view and the spinner). Then the `__block` storage type is used for the image variable because you want to be able to set its value from inside the block in case you don't have a cached version of it yet. Inside the block, you need to retain the image because `__block` keeps it from being retained automatically. Finally you release the image again in the block that's run on the main queue because you first added it to the matching results dictionary, which retains the image for you.

When you run your application now, everything should work smoothly and never freeze once.

13.3 Summary

We covered a lot of ground in this chapter. We looked at blocks, a powerful and versatile new addition to the C language, and we got our feet wet with GCD, an easy way to add concurrency to your application and keep it responsive at all times. There's a lot

more to learn about GCD, but that would go beyond the scope of this chapter. We covered the most important use case: performing work in the background and calling back to the main thread to update the UI. To dig deeper into GCD, you should look at Apple's Concurrent Programming Guide at <http://developer.apple.com/library/ios>.

Chapter 14, our final chapter, covers advanced debugging techniques.

14

Debugging techniques

This chapter covers

- Creating a buggy application
- Using NSLog
- Controlling leaks with Instruments
- Detecting zombies

There's nothing worse than being close to releasing a new application and in your final once-over finding a particularly nasty application crash that seems to have no obvious cause. Even during day-to-day development of your application, it's unlikely that you'll write perfect code the first time around, so understanding how to debug your application in the Xcode environment is an important skill.

The Xcode environment integrates a large number of debugging and code analysis tools, but as usual, their true power is only unleashed if you know which tool to use and how and when to use it. A common complaint for new Objective-C developers who are used to managed environments such as C# or Java is that memory management and correct handling of object retain counts (retain, release, autorelease, and dealloc message handling) is hard; so Xcode provides extensive support for detecting and diagnosing memory-related errors.

14.1 Building an application, complete with bugs

Let's create an application and purposely add some errors so that you can discover, observe, and resolve them. The application isn't overly complex, but it helps demonstrate the debugging tools at your disposal. In Xcode create a new Navigation-based application named `DebugSample`, and then replace its `RootViewController` `tableView: numberOfRowsInSectionInSection:` and `tableView:cellForRowAtIndexPath:` methods with the implementation in the following listing.

Listing 14.1 A sample `tableView` implementation with intentional bug

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSectionInSection:(NSInteger)section {
    return 450;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        NSLog(@"We are creating a brand new UITableViewCell...");
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }

    NSLog(@"Configuring cell %d", indexPath.row);
    cell.textLabel.text =
        [NSString stringWithFormat:@"Item %d", indexPath.row] retain];

    return cell;
}
```

When you build and run this application, you should see a `UITableView` consisting of 450 items. You should also notice that, as you flick through the table, it emits a log message indicating which cell is currently being configured, as shown in figure 14.1.

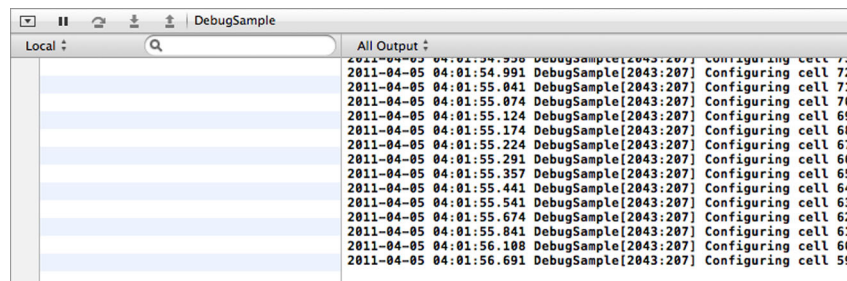


Figure 14.1 Console output from the `DebugSample` application: a new log message is emitted for each cell that's configured for the `UITableView`.

14.2 Understanding NSLog

In this book you've utilized the `NSLog` function extensively to emit diagnostic messages to the Xcode debugger console window. What you may not realize is that `NSLog` continues to log messages even when your application is used outside of the debugger. It even logs messages once a user purchases your application from the iTunes App Store and runs it on their own device. Where do these log messages end up, and how do you retrieve them?

To answer these questions, deploy the `DebugSample` application onto your real iPhone or iPad device, and run the application a couple of times (see appendix A for details if you haven't done this before). When you reconnect the device to your computer, you can bring up the Xcode Organizer window (Shift-Command-2), which should look similar to figure 14.2.

This window can be used for a wide range of tasks, such as uninstalling applications, maintaining device provisioning profiles, and capturing application crash reports or screenshots. In this case, you want to be in the Console section, as shown in figure 14.3.



Figure 14.2 The Xcode Organizer window showing a list of connected iOS devices. Using the various tabs, you can control different aspects of your connected device, such as installing and uninstalling applications, maintaining provisioning profiles, capturing screenshots, and reviewing application crashes and log files.

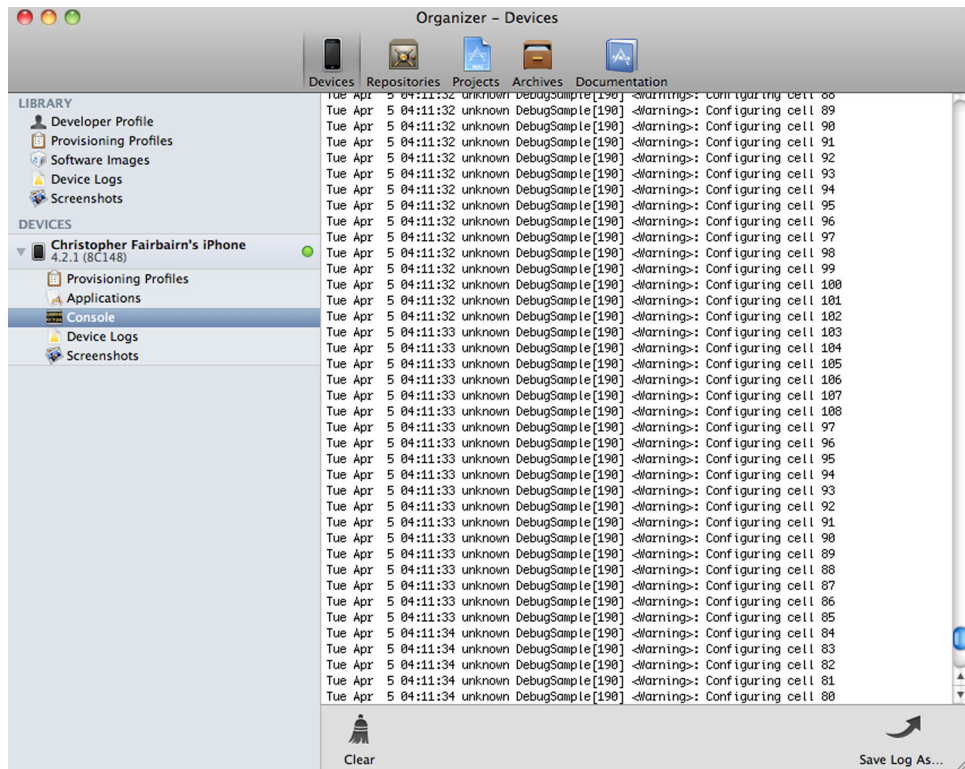


Figure 14.3 The Devices console section of the Organizer window shows log messages not only from your application's NSLog calls but also from other system applications and iOS platform services.

Scrolling through the console, you should be able to find the log entries produced by DebugSample's calls to NSLog while the device was disconnected from the computer. While connected, if you launch the application on your device, you should even notice that the console window updates in real time.

Although it can be handy during development to log a large amount of content via NSLog, it's often less than desirable in builds which customers will obtain. Instead of manually commenting out unnecessary calls to NSLog, you can use some C preprocessor magic to automatically ensure calls to NSLog appear only in debug builds. Open the DebugSample_Prefix.pch file and insert the contents shown in the following listing.

Listing 14.2 Helpful macros that improve logging flexibility during development

```
#ifdef DEBUG
#   define LogDebug(...) NSLog(__VA_ARGS__)
#else
#   define LogDebug(...) do {} while (0)
#endif

#define LogAlways(...) NSLog(__VA_ARGS__)
```

Listing 14.2 defines two C preprocessor macros called `LogAlways` and `LogDebug`. `LogAlways` is an alias for `NSLog`. The definition of `LogDebug` alters depending on whether or not the state of `DEBUG` is defined. If `DEBUG` is defined, `LogDebug` is defined to be a call to `NSLog`, but if `DEBUG` isn't defined, the following `do while` statement is substituted for calls to `LogDebug`:

```
do { } while (0)
```

You may think this a strange substitution to make and that you could replace calls to `LogDebug` with an empty statement such as follows:

```
#define LogDebug(...)
```

The problem with this implementation is that it could introduce subtle and hard-to-detect changes into your application's behavior. Imagine the following source code is using the `LogDebug` macro:

```
LogAlways("Process started");
if (a != b)
    LogDebug(@"A and B don't match!");
LogAlways("Process completed");
```

If, during release builds, calls to `LogDebug` were replaced with an empty statement, the Objective-C compiler would, in effect, see the following source code, which is completely different from the behavior you originally intended:

```
LogAlways("Process started");
if (a != b)
    LogAlways("Process completed!");
```

By substituting calls to `LogDebug` with `do { } while (0)`, you can safely assume that end users won't see the debug messages emitted by `NSLog` and at the same time ensure the substitution won't cause compilation errors or unexpected changes in behavior. During debug builds, any use of the `LogDebug` macro is automatically converted to a standard call to `NSLog`, so builds you create in Xcode for development or diagnostic purposes retain the full set of log messages.

In the source code for `tableView:cellForRowAtIndexPath:` replace one of the calls to `NSLog` with `LogAlways`, and replace the other with a call to `LogDebug`. Then rerun your application and notice that, as you scroll through the table view, the Xcode console window is logging content identical to what it previously logged. This is because, by default, iOS project templates in Xcode configure their projects to define the `DEBUG` preprocessor symbol. But if you stop the application, switch to Xcode to make a release version of your application (Product > Edit Scheme... > Info, then change the Build Configuration option), and rerun the application, you should notice that only the calls to `LogAlways` make it to the console window. Calls to `LogDebug` are stripped out by the C preprocessor before the Objective-C compiler has a chance to compile the source code.

As you start to introduce macros such as `LogDebug` and `LogAlways` into an existing code base, it can become laborious to search and replace all instances of `NSLog` or

Additional schemes can be helpful

Although the Xcode project templates produce a project with one scheme by default, you can add other schemes to your project.

You could, for instance, create different schemes for “light” and “full” versions of your application. Using different C preprocessor defines, you could configure your source code with code similar to the following

```
#ifdef FULL_VERSION
...
#endif
```

to alter the behavior of the two application variants without needing to maintain two completely separate Xcode projects.

other functions that need to be updated. Luckily, Xcode provides support for performing such tasks in a more efficient manner. From a terminal window, you can navigate to your project’s folder and use the following command:

```
tops replace "NSLog" with "LogAlways" *.m
```

This command replaces all calls to the NSLog function with calls to LogAlways. It performs this task for all *.m source code files found in the current directory. Tops is smart enough not to alter other references to NSLog, such as those in comments; it has built-in knowledge of Objective-C syntax and performs more than a simple search and replace. For safety purposes, if you want to double check what a call to tops will perform, add the -dont argument to the command line. This argument causes tops to perform as normal, except it won’t modify a single file. Instead, it sends to the console a list of changes it would have performed had the -dont argument not been present.

It’s also possible to perform this task directly from Xcode’s text editor: move the mouse over a call to NSLog, right-click, and select Refactor. In the dialog box that appears, type in LogAlways and click the preview button. You’ll get a list of files Xcode will change, and selecting each one will graphically highlight the changes that will occur once you click the Apply button.

That sums up improved the NSLog situation—but there’s another problem with the sample application. If you run the application, it appears to be working correctly, but with time (and especially on a real device), it will eventually crash. It has a memory leak, but how do you find it?

14.3 Bringing memory leaks under control with Instruments

In Xcode select Product > Profile. In addition to the iOS Simulator, you’ll see an application called Instruments. From Instruments’ initial screen, select the Leaks trace template and click Profile. A screen similar to figure 14.4 will be displayed.

While Instruments is monitoring the execution of your application, select the Allocations instrument and scroll around the UITableView. Notice that the line labeled All Allocations (representing all memory allocated in the application) keeps indicating that

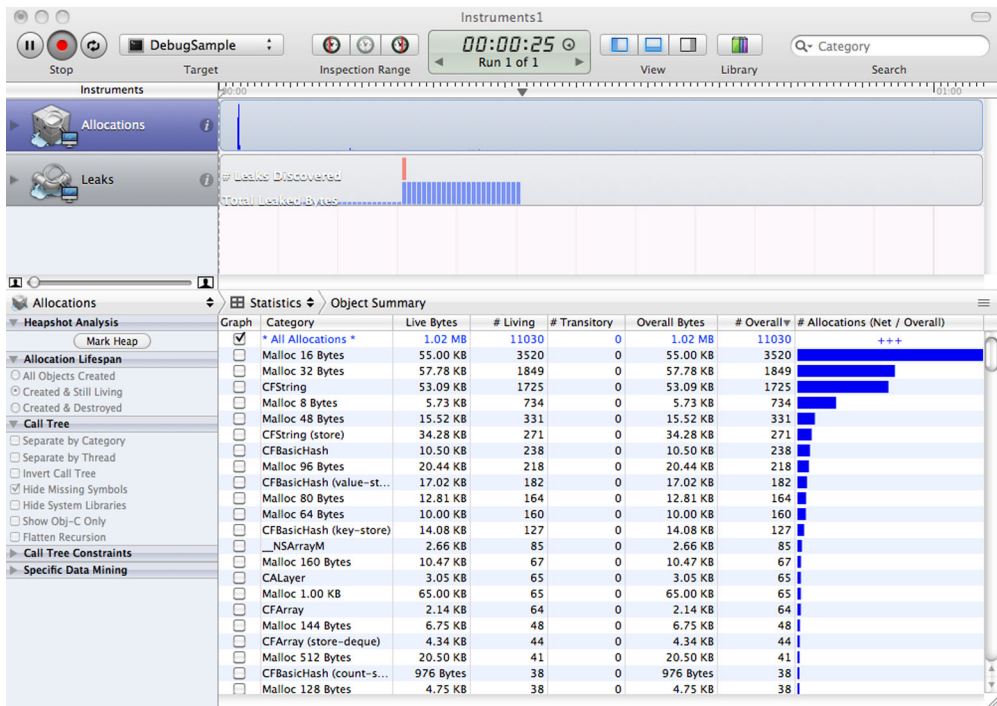


Figure 14.4 Instruments is a flexible application that can attach one or more diagnostics tools, called instruments, to your application. As your application runs, the instruments monitor its performance and behavior and can highlight potential problems or opportunities to improve your application.

memory is being consumed. You can confirm this by looking at the column labeled # Living, which indicates the total number of objects currently alive in the application. As you continue to scroll around the UITableView, this number steadily increases at a rate that seems proportional to your scrolling. Notice also that the graph labeled Leaks indicates that leaks have been detected and a steadily increasing blue line indicates how much memory is being leaked.

Stop the application by clicking the Stop button in the Instruments toolbar, and you'll get a list of all the objects in memory that Instruments has detected as being potentially leaked. It appears your application has leaked a number of string objects.

While it's handy to know that a memory leak is present, it's more useful to determine how the leak came about. With one of the memory leaks selected, display the Extended Details pane (Cmd-E) and you'll see a call stack showing where the memory originally became allocated.

As you switch between the various memory leaks, notice that most of the call stacks appear virtually the same. This is a good sign that you have a single bug to track down. You'll also notice that there's a line in the stack trace for `RootViewController's tableView:cellForRowAtIndexPath:` method. If you double-click it, you'll see the source code of the method with the following line highlighted:

```
cell.textLabel.text =  
    [[NSString stringWithFormat:@"Item %d", indexPath.row] retain];
```

Looking at this line and reviewing the object ownership rules discussed in chapter 9, you can see that this line is the source of the memory leak.

The line allocates a new `NSString` object, which is added to the autorelease pool (by virtue of using a class method named using the `classNameWithxxxx` naming convention). It then takes ownership of the string object by sending it a `retain` message, and finally assigns it to the `text` property of the `UITableViewCell`'s `textLabel`. Although the `text` property also takes ownership of the object, it's not the source of the memory leak, because when the `UITableViewCell` is deallocated, it'll take care of releasing ownership of the string.

The source of the memory leak is the explicit `retain` message sent to the object. By sending a `retain` message, the current code takes responsibility for the object, but it never decrements the reference count by sending a matching `release` or `autorelease` message. Therefore, string objects created by the line

```
cell.textLabel.text =  
    [[NSString stringWithFormat:@"Item %d", indexPath.row] retain];
```

are never returned to a reference count of zero and are never deallocated. This is the source of the memory leak. Replace the line with the following version, which correctly maintains the reference count:

```
cell.textLabel.text =  
    [NSString stringWithFormat:@"Item %d", indexPath.row];
```

With the memory leak resolved, rerun the application using Instruments to verify your analysis is correct and that the memory leak has been resolved. This time you should see that Instruments doesn't detect any leaks and that the total number of living objects, although fluctuating, stays fairly static. It no longer increases as you scroll through the tableview.

In this section we discussed how to debug a situation in which an object's retain count is accidentally incremented too much, in effect keeping the object alive for too long. It's also possible for the opposite scenario to occur: an object's retain count can decrement back to zero too quickly, resulting in the object being deallocated before you've finished using it.

This is potentially worse than a memory leak, because once an object is deallocated, any code attempting to access that object will cause a variety of more fatal outcomes depending on the luck of the draw. Your application could continue on without any apparent problem, it could crash entirely, or it could calculate an incorrect result or perform the wrong operation.

Let's purposely insert another type of error in your `DebugSample` application so you can see how to detect and resolve such bugs when they accidentally creep into your applications.

14.4 *Detecting zombies*

Once an object is deallocated, the Objective-C runtime is free to reuse the associated memory for other memory allocation requests. The object doesn't exist anymore. It's "dead" compared to a "live" object that has one or more active owners.

When you attempt to access a dead object, you may get lucky—the object may still be sitting in memory and you won't notice a difference (when an object is deallocated, the bookkeeping is updated, but what is in memory isn't overwritten until that memory is reused). But the more likely scenario is that the Objective-C runtime has re-assigned the associated memory for another task and attempting to access the dead object will cause an application crash.

Detecting object references to dead objects can be difficult, but Instruments and the Objective-C runtime provide a feature called NSZombies, which makes detecting them much easier and ensures they always fail in an identifiable manner. It turns dead objects into zombies. A zombie object is "the living dead." When the NSZombies feature is enabled, an object that should be deallocated (sent a dealloc message) is instead kept alive in memory. By using some of the dynamic features of Objective-C, the type of the object is modified to be `_NSZombie_xxx` where `xxx` is the original class name. The `NSZombie` class responds to any message sent to an object by logging a message to the console that something has attempted to communicate with an object that wouldn't exist if the `NSZombie` feature wasn't in use.

To demonstrate zombie object detection, add the code in the following listing to `RootViewController.m`.

Listing 14.3 Using NSZombies

```
static NSString *mymsg = nil;

- (NSString *)generateMessage:(int) x {
    NSString *newMessage;

    if (x > 1000)
        newMessage = @"X was a large value";
    else if (x < 100)
        newMessage = [NSString stringWithFormat:@"X was %d", x];

    return newMessage;
}

- (void)viewDidLoad {
    [super viewDidLoad];
    mymsg = [self generateMessage:10];
}

- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSLog(@"Your message has %d characters in it", mymsg.length);
}
```

With the code changed, run the application in Instruments by selecting Product > Profile > Allocations Trace Template. As soon as the application starts, stop Instruments

(and the application) by clicking the Stop button in the toolbar. Then click the (i) icon at the right-hand side of the Allocations instrument. Now you can configure the instrument. Tick the Enable NSZombie Detection and Record Reference Counts check boxes. Then restart the application by clicking the Record button in the toolbar.

A warning will be displayed that traditional memory leak detection won't work with NSZombies enabled. This is a natural side effect of this feature because enabling NSZombies means objects are never deallocated; they're all converted to zombies and kept in memory in case future code attempts to access them.

In the DebugSample application, tap on a cell in the tableview and notice that the application crashes and displays a pop-up message in Instruments similar to that shown in figure 14.5.

If you click the small arrow beside the second line of the zombie message, the bottom pane in Instruments will update to display the reference-count history of the object in question.

From the history listed in figure 14.5, you can see that the object was originally allocated (malloc'ed) some memory, then added to the autorelease pool by NSString's stringWithFormat: message (as detailed by the column titled Responsible Caller), and then released when the NSAutoreleasepool was released.

This series of events brought the object's reference count (listed in the RefCt column) back to zero, and the object was deallocated. But because the NSZombies feature was enabled, the object was instead converted into a zombie, and this object was then accessed by RootViewController's tableView:didSelectRowAtIndexPath: method.

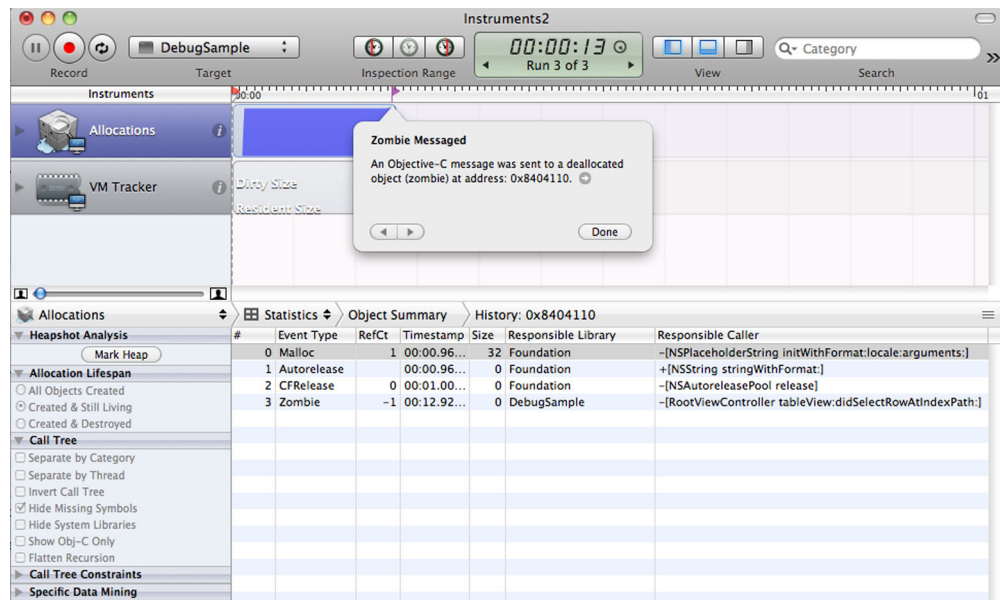


Figure 14.5 Instruments demonstrating a detected attempt to communicate with a zombie object. The bottom pane details the complete reference-count history of the object.

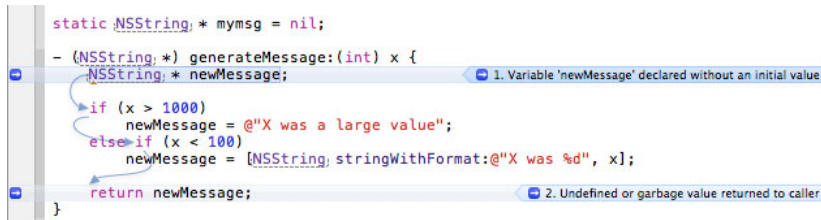


Figure 14.6 An error detected by the static code analysis tool can be displayed graphically in the Xcode IDE. Notice how the execution path required to trigger the error is visually demonstrated by arrows overlaid on the original source code.

Double-clicking anywhere on the row associated with `tableView:didSelectRowAtIndexPath:` will bring up the source code, indicating where you attempted to access an object that no longer existed.

In this case, you can see the error occurred because the string object pointed to by the `mymsg` variable in the `viewDidLoad` method had no explicit retain message. The following source code would resolve the problem:

```
mymsg = [[self generateMessage:10] retain];
```

This bug was engineered to be easy to detect; listing 14.3 contains other bugs. Luckily, Xcode is smart enough to detect some of them without your even running the application and waiting for it to crash! To get Xcode to perform a static analysis of your source code, select `Product > Analyze`. Notice that the `generateMessage:` method is highlighted with the message “Undefined or garbage value returned to caller.” This message indicates that in at least one scenario, the `generateMessage:` method may fail to return a valid string object and instead return some random “garbage” that’s likely to crash the application. To find how this is possible, click the error message on the right-hand side of the Code Editor window. The blue arrows on top of the source code guide you through interpreting the error, as shown in figure 14.6.

Notice in the figure that the variable `newMessage` isn’t assigned an initial value. Based on the value of `x`, the first `if` statement could evaluate to `false`, which would cause execution to jump to the second `if` statement. This statement could also evaluate to `false`, in which case execution would proceed to the `return` statement. At that point, an attempt to return the value of `newMessage`, the value of which isn’t explicitly set, will result in an error.

When the code analysis feature of Xcode detects errors such as this, it’s a cue for you to analyze your logic and determine if there’s a corner case you haven’t covered or if a variable has accidentally been left uninitialized.

14.5 Summary

Xcode provides an extensive range of debugging tools and integrates well with a number of open source tools. The technology behind Instruments is mostly open source,

and its ease of use and visual nature permitting exploration and interpretation of captured data is unrivaled.

Debugging applications that you've deployed via the iTunes App Store requires additional forethought if you want to maximize your ability to diagnose crashes that you yourself can't replicate. At a minimum, you should archive a copy of your source code and debugging symbols (*.dSYM files) for each update of the application you submit. Doing so increases your chances of being able to quickly zero in on probable causes of crashes that occur "out in the wild."

appendix A

Installing the iOS SDK

In this appendix we briefly walk you through the installation of the iOS software development kit (SDK).

A.1 *Installing the iOS SDK*

The first step is to select and sign up for the Apple developer program and download the SDK.

A.1.1 *Becoming an Apple developer and downloading the SDK*

Apple frequently changes the way you can download and install Xcode. At the time of this writing, you simply have to search for “Xcode” on the Mac App Store, where you can then download Xcode 4.1 for Mac OS X Lion for free. You should always be able to find the latest version and instructions at <http://developer.apple.com/xcode/>. To get access to valuable and up-to-date developer resources you should also set up a free developer account at <http://developer.apple.com/programs/register>. Click the blue Get Started button, and complete the registration steps. The free account also gives you access to the previous versions of Xcode for Mac OS X Snow Leopard and below.

However, simply downloading or purchasing Xcode through the Mac App Store will not enable you to test and run your apps on an actual iOS device. If you plan to test your applications on an actual device—which you should—or to submit them to the App Store, choose the \$99/year iOS developer program. You can enroll by going to <http://developer.apple.com/programs/ios/>, clicking the blue Enroll Now button, and completing the enrollment. That will enable you to test your apps on devices and submit them to the App Store.

A.1.2 *System requirements*

At the time of this writing, the requirements to install Xcode and the iOS SDK from the Mac App Store are an Intel-based Mac that runs Mac OS X Lion 10.7 or later and



Figure A.1 The Downloads section of the Apple Developer page

about 10 GB of free disk space. If you signed up for the free or paid developer program, you can also download older versions of Xcode for Mac OS X Snow Leopard.

A.1.3 *Downloading Xcode and the iOS SDK*

If you chose to purchase Xcode through the Mac App Store, Xcode won't be installed directly. Instead, an application to install Xcode will be downloaded and installed. If that's what you did, you can continue with the next step.

If instead you signed up for the iOS developer program, go to <http://developer.apple.com/xcode/index.php> and log in with your username and password. To start the download, click the big link in the Download Xcode 4 box as shown in figure A.1. It's a huge download, roughly 4.5 GB, so be patient.

A.1.4 *Installing Xcode and the iOS SDK*

If you've purchased Xcode from the Mac App Store, just double click on the "Install Xcode" app to start the installation process. If instead you've downloaded Xcode from Apple's developer website, the disk image (DMG) normally automatically opens in Finder once the download is complete (if not, double-click the .dmg file). Double-clicking on the Xcode and iOS SDK icon starts the installation process. Unless you have very specific needs and you know what you're doing, you should leave all the settings as they are and just click Continue until the installation starts. Once again, patience is needed because the installation takes a little while. After the installation finishes, you'll find Xcode in `/Developer/Applications/Xcode`.

A.2 *Preparing your iOS device for development*

When you develop applications for iOS devices, you naturally also want to test them on an actual device and not just in the simulator. Because iOS applications have to be digitally signed in order to run on a device, you have to get a digital certificate from Apple and install a Provisioning Profile on your device. Let's do that next.

A.2.1 *Creating a certificate*

First, launch an application on your Mac called Keychain Access (it's in the Applications folder inside the Utilities folder). In the menu bar, select Keychain Access > Preferences, and set both Online Certificate Status Protocol and Certificate Revocation List on the Certificates tab to Off. Next, select Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority..." from the menu bar. You should see the window depicted in figure A.2.

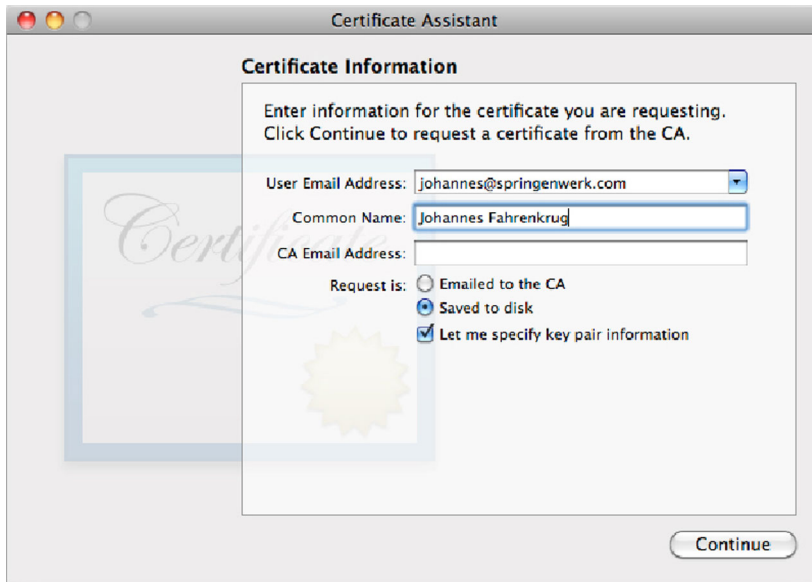


Figure A.2 Requesting a certificate using the Keychain Access application

Fill in your email address (the same one you used to sign up as an Apple developer) and your name, select Saved to Disk in the Request Is section, and check the Let Me Specify Key Pair Information check box. Click Continue, save the file, and remember where you saved it. You need to upload it to the Apple website in the next step. On the next screen, make sure the key size is 2048 bits and RSA is chosen as the algorithm. Click Continue and then Done.

Now head over to <http://developer.apple.com/ios>, log in, and select iOS Provisioning Portal at the top right. Once there, select Certificates from the left sidebar and click the Add Certificate button. On the bottom of the Certificates page, after a lot of text, you'll see a Choose File button. Click it and select the file you saved in the previous step. Click the Submit button to upload the Certificate Signing Request to Apple. It might take between a few seconds and a few minutes for the certificate to be created. When the Approve button is available next to your certificate in the Certificates section of the Provisioning Portal, click it, reload the page, and click Download. Double-click the downloaded file to install it in your keychain.

You also need to download and install the WWDC intermediate certificate. A link to it also appears on the Certificates page. After downloading it, double-click it to install it in your keychain.

A.2.2 *Provisioning a device using Xcode*

For generic development (for example, to run all the sample code from this book), Xcode can take care of setting up your device for development. But if you need to use

advanced features like in-app purchases or push notifications, you must set up your device manually on the website. Section A.2.3 explains how to do that.

To provision your device with Xcode, launch Xcode and select Window > Organizer. The Organizer window is where you manage your devices, provisioning profiles, archived applications, and source control repositories. On the Devices tab, select Provisioning Profiles from the Library section on the left (figure A.3).

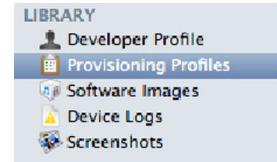


Figure A.3 The Provisioning Profiles section of the Xcode Organizer

On the bottom of the window, check the Automatic Device Provisioning check box (figure A.4).

Now plug in your iOS device using the USB cable. It'll show up in the Devices section. Select it and click the Use for Development button. A dialog will come up asking you for your Apple developer credentials. Enter them and click Submit Request in case Xcode asks you whether you want it to request a development certificate. Xcode will then log in to the Provisioning Portal, register your device ID, and create a provisioning profile called Team Provisioning Profile: * and install it on your device.



Figure A.4 Enabling Automatic Device Provisioning in the Xcode Organizer

Your device is now ready for development!

A.2.3 Provisioning a device manually

If you want to provision your device manually, you should still first follow the steps in the previous section to add your device to the developer portal. If you'd rather do this step manually as well, connect your device to your Mac, open the Xcode Organizer, select the device, and copy the long string of numbers and letters called Identifier. Then log in to the iOS Provisioning Portal and select Devices. Click the Add Devices button, enter a meaningful name, and paste the device ID into the second text field. Click Submit to save your device.

Applications that you want to submit to the Apple App Store require a unique App ID. An App ID consists of the reversed URL of your company and an application name, such as com.mygreatcompany.myamazingapp (this is why Xcode asks you to enter such an App ID when you create a new project). To add a new App ID, select App IDs from the sidebar and click the New App ID button. Enter a meaningful name and the bundle identifier (the reversed URL with the application name) into the second text field. This bundle identifier must match the one you specify in the settings of your Xcode project.

Finally, a provisioning profile is the combination of an App ID and a device ID, practically allowing a device (or a group of devices) to install and run an application with a certain App ID. You can create provisioning profiles for both development and

distribution. Section A.2.2 took care of creating one for development; now let's look at how to create one for distribution.

In the Provisioning Portal, select Provisioning and then the Distribution tab. Click the New Profile button. On the next page, you can select if you want to create a profile for submitting your application to the App Store or for giving it to beta testers via Ad Hoc distribution. The following steps are the same for App Store and Ad Hoc distribution; the only difference is that you don't have to specify devices for an App Store profile. For now, select Ad Hoc, enter a meaningful name (such as My Awesome App Ad Hoc), select the desired App ID from the Select box, and select the device(s) this provisioning profile should be valid for. Click Submit and go back to the Distribution tab and click the Download button next to your brand new Ad Hoc provisioning profile. Once the profile is downloaded, double-click it to install it. That's it.

A.2.4 *Running an application on a device*

To learn how to run an application on a device, you'll create a blank new project in Xcode and run it on your device. Fire up Xcode and select File > New > New Project. Select Tab Bar Application and click Next. Name the product "My Great App" and put your company's reversed URL as the Company Identifier. Click Next and save the project. Now plug in your iOS device and select your iOS device from the Scheme selector right next to the Run and Stop buttons in Xcode. Press the Run button: the project should compile and run on your device.

You can change which provisioning profile should be used (if you have multiple ones) by selecting the project in the Navigator view in Xcode, selecting the Build Settings tab, and scrolling down to the Code Signing section. There, under the Debug or Release configuration, you can change the provisioning profile and certificate by selecting a different one next to the Any iOS SDK entry (figure A.5).

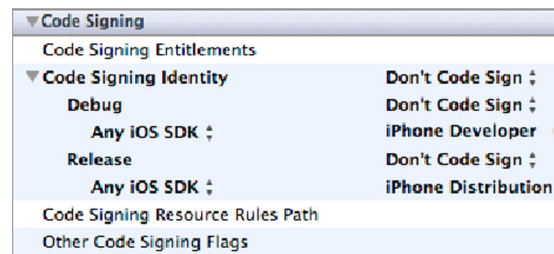


Figure A.5 Code Signing settings in Xcode

Now you're all set to run and test your applications on your device. To dig deeper into the complex topic of certificates and device provisioning, check out the elaborate tutorials and videos on Apple's Provisioning Portal website.

appendix B

The basics of C

Throughout this book we've assumed you understand the basic syntax of Objective-C. Even if the iPhone is your first exposure to Objective-C, you may have preexisting knowledge of another language that shares a similar lineage with the C language. As an example, the basic control structures in C, C++, Java, and C# are all essentially the same.

A lot of beginner Objective-C developers don't fully appreciate the C lineage of Objective-C, but Objective-C is essentially a layer over top of C. Every valid C program is also, by definition, a valid Objective-C program, albeit one that doesn't take advantage of Objective-C's object-oriented features.

The impact of this lineage is that most of the syntax in Objective-C for making decisions and controlling the flow of execution throughout an application is virtually identical to that of the C programming language. If you want to become a truly great Objective-C programmer, you must first become an ace at using C.

If you are new to programming or haven't written code in a C-inspired language before, this appendix is for you: it covers the basic control structures and details of programming in a C-style language. Let's begin our discussion by looking at how variables, messages, and classes should be named to avoid potential errors during compilation.

B.1 *Variable naming conventions*

All variables, methods, and classes in an application must be provided with a name to identify them, but what makes a valid variable name?

A variable name is one type of identifier. An identifier is simply a name that uniquely identifies an element such as a variable, custom data type, or function in the source code that makes up an application. In C, identifiers can consist of any sequence of letters, digits, or underscores. They can't start with a digit, however, and those that start with an underscore are reserved for internal use by the

compiler or support libraries. Identifiers can be of any length but are case sensitive, so `fooBar`, `FOOBAR`, and `FooBar` are all valid identifiers that refer to different items in an application.

Following are some examples of valid variable identifiers:

- `MyVariable`
- `variable123`
- `Another_Variable`

And here are some examples of invalid variable identifiers:

- `$Amount`
- `cost(in_cents)`
- `monthly rental`
- `10timesCost`
- `int`

The last invalid variable identifier, `int`, deserves additional comment. At first glance, it may appear to pass all the rules for valid identifiers. The keyword `int`, however, already has a special meaning in C source code. It identifies the integer primitive data type. Keywords that already have a special meaning in a C application are called *reserved words*. A reserved word is an identifier that has a special meaning to the C compiler and therefore can't be used as an identifier in your source code. Other examples of reserved words include `if`, `return`, and `else`, among others.

B.1.1 Hungarian notation

Although you're free to name your variables using any identifier that conforms to the rules previously specified, over the years a number of traditions and conflicting styles have developed regarding the best way to name identifiers. Many developers already familiar with C-based applications are aware of a convention called *Hungarian notation*.

Hungarian notation is the concept of encoding into your variable names a description of the data type associated with the variable. This is done via a prefix that represents the data type; for example, `iCount` indicates an integer count, and `chGender` indicates a variable capable of storing a character. Table B.1 lists some of the most common prefixes you may encounter.

Table B.1 Common Hungarian notation prefixes for variable names in C source code

Example variable identifiers	Description
<code>chFoo</code>	Character (<code>char</code>)
<code>iFoo</code> , <code>nBar</code> , <code>cApples</code>	Integer, number or count (<code>int</code>)
<code>bFlag</code> , <code>fBusy</code>	Boolean or flag (<code>BOOL</code>)
<code>pszName</code>	Pointer to null terminated string (<code>char *</code>)

As IDEs have become more powerful with features such as Code Sense (Apple's equivalent to IntelliSense), the use of Hungarian notation has quickly lost favor among developers. It's now quick enough to determine the data type associated with a variable without encoding it into the name. Most usage of Hungarian notation is now confined to C-based source code; it's uncommon to see this convention used in applications that embrace Objective-C. This is not to say that other naming conventions have also faded; for example, camel case is still popular among Objective-C developers.

B.1.2 Camel case

It's helpful to make identifiers as descriptive and self-documenting as possible. As an example, a variable named `CountOfPeople` is more meaningful than a variable simply named `n`.

Because identifiers can't contain spaces, one common technique for achieving this goal is to capitalize the first letter of each word, as seen in the variable name `CountOfPeople`. This technique is named *camel case* for the "humps" the uppercase letters spread throughout the identifier. Another alternative to camel case is the use of underscores to separate words: `Count_Of_People`.

B.1.3 Namespaces

When your applications start getting more complex or you start to use code developed by a third party, one problem you may come across is a clash between identifiers. This occurs when two separate parts of the application attempt to use the same identifier to represent different things. For example, your application may name a variable `Counter`, and a third-party library might name a class `Counter`. If you were to refer to `Counter` in your source code, the compiler would be unable to determine which item you were referring to.

Some programming languages resolve this problem via a mechanism called *namespaces*. A namespace allows you to group related identifiers into separate groups or containers. For example, all the code written by Apple could be placed inside one namespace, and all the code written by you could be placed inside another. Each time you reference a variable or function, you must provide not only the name of the identifier but also its containing namespace (although this may be implied rather than explicit). In this manner, if two regions of code attempt to reference something called `Counter`, the compiler can differentiate between the two interpretations and understand your intent.

The C programming language doesn't provide a concept equivalent to that of namespaces, but it's possible to simulate enough of their behavior to avoid collisions by using commonly agreed-upon prefixes for your identifiers.

A good example of this technique is the venerable `NSLog` function. This function can be thought of as having the name `log` in a namespace called `NS`. `NS` is the namespace prefix Apple uses for the Foundation Kit library; as such, it would be unwise to start the name of any of your variables or functions with the `NS` prefix because it would increase the likelihood of its clashing with a preexisting Apple function.

You may like to get into the habit of prefixing your own publicly visible identifiers with a prefix of your own choosing, perhaps based on the initials of your name or company.

With the proper naming of variables and other elements out of the way and no longer producing compile-time errors, let's explore how to use one or more variables to calculate new values or to answer questions.

B.2 Expressions

It's uncommon for an application to be developed without requiring at least one mathematical calculation, so C provides a rich set of operators for mathematical expressions.

B.2.1 Arithmetic operators

The most obvious operators are the arithmetic ones that perform the basic mathematical operations we're all familiar with. These operators are listed in table B.2.

Table B.2 Arithmetic operators available in C

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)

These arithmetic operators all have their expected behavior. For example, the expression $4 + 3$ results in the value 7 being calculated. The modulus operator returns the remainder of an integer division operation, so $9 \% 2$ equals 1 because 2 goes into 9 four times with 1 left over.

B.2.2 Comparison operators

Once you've calculated one value, you may want to compare it against another. Relational operators allow this comparison, which results in a Boolean value being calculated that indicates the truth of the statement. Table B.3 outlines the available relational operators.

As an example, the expression $x \geq 5 * 10$ determines if the value of variable x is greater than or equal to 50 (the value of the expression on the right side of the \geq operator).

Sometimes comparing a single value isn't enough. For example, you may want to check if a person's age is greater than 25 years and weight is less than 180 pounds. In such scenarios, you must use compound conditional statements.

Table B.3 Relational operators available in C

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

A compound conditional statement is two Boolean expressions joined by a logical operator. Common logical operators are shown in table B.4.

Table B.4 Logical operators available in C

Operator	Description
&&	And
	Or
!	Not

As an example, the previously discussed condition could be expressed as `age > 25 && weight < 180`. The `!` is a special logical operator in that it requires only one Boolean value and it reverses it (true becomes false, false becomes true).

Although it makes little difference to the expression just shown, it's important to note that C implements short-circuit Boolean evaluation: an expression will stop being evaluated as soon as the end result can be determined without any level of doubt. The following logical expression involving the `&&` operator demonstrates how this feature can make the difference between your application running correctly or randomly crashing:

```
struct box *p = get_some_pointer_value();
BOOL result = (p != NULL) && (p->width == 20);
```

The expression on the second line evaluates to `true` if `p` is a non-NULL pointer and the width of the box it points to has a width of 20.

Without short-circuit Boolean evaluation, the expression would crash whenever `p` was NULL. This would occur because after evaluating the left side of the `&&` operator (`p != NULL`), execution would then evaluate the right side (`p->width == 20`) and crash while attempting to dereference the NULL pointer. The right side is evaluated even though the final result of the statement is known to be false because `false && x` for any value of `x` will always be false.

With C implementing short-circuit Boolean evaluation, when `p` is `NULL`, the right side of the expression isn't evaluated, safely avoiding the potential `NULL` reference exception.

B.2.3 Bitwise operators

A confusing thing when you first look at the operators available in C is the apparent doubling up of the logical operators. For example, `or` (`|`, `||`), and `&` (`&`, `&&`), and `not` (`~`, `!`) all appear to have two versions. There's method to this madness: the singular forms, called *bitwise operators*, have a slightly different function than the equivalent *logical operators*. The bitwise operators are listed in table B.5.

Table B.5 Bitwise operators available in C

Operator	Description
<code>&</code>	And
<code> </code>	Or
<code>^</code>	Xor (exclusive or)
<code>~</code>	One's complement (0s become 1s, 1s become 0s)
<code><<</code>	Left shift
<code>>></code>	Right shift

The byte is the smallest-sized variable that a C program can operate on. There's no bit type capable of storing a single on or off state. Bitwise operators, however, allow you to test, toggle, or shift the individual bits that make up an integer value. As an example of the difference between logical and bitwise operators, compare the output of the following code snippet:

```
int x = 8 && 1;
int y = 8 & 1;
NSLog(@"x = %d, y = %d", x, y);
```

The output will indicate that `x` has the value 1 and `y` has the value 0. Clearly, the logical and bitwise and operators have performed different operations. The first expression uses the logical and operator (`&&`) and evaluates to `true` because both 8 and 1 are nonzero and hence treated as representing true.

The second assignment uses the bitwise and operator (`&`) and performs the and operation on each bit in the two integer variables separately. Converting the values 8 and 1 into binary, you obtain 00001000 and 00000001. ANDing bit 0 from the first value with bit 0 from the second value to produce bit 0 of the result, and so on, for all bits in each value will produce the result 00000000. This occurs because there's no bit position in which both values have a 1.

B.2.4 Assignment operators

So far, we've used the assignment operator (=) to store the value of an expression in a variable. But this isn't the only form of assignment operator available to Objective-C developers. The following code snippet also introduces the += assignment operator:

```
int x = 5, y = 5;

x = x + 4;
y += 4;

NSLog(@"x = %d, y = %d", x, y);
```

This code snippet initializes two variables, `x` and `y`, to the value 5. Variable `x` is then assigned the value of the expression `x + 4`, which increases its existing value by 4. Variable `y`, on the other hand, also has its value incremented by 4, but this time via the more compact syntax afforded by the += operator.

The += operator in effect says to “increment the variable on the left side by the value of the expression on the right side.” Similar shorthand syntax can be used for most of the arithmetic and bitwise operators, such as +, -, *, /, %, &, |, ^, >>, and <<, to perform a mathematical operation and assignment in one step.

Another often overlooked feature of the assignment operator (=) is that it can be nested in a larger expression. The value of the assignment operator is the value it will assign to the variable on its left side. This can lead to statements such as the following:

```
int x, y;

x = (y = 5 + 2) + 4;

NSLog(@"X is %d, Y is %d", x, y);
```

The expression `(y = 5 + 2) + 4` can be thought of as consisting of two individual steps. Working from the inside outward, the first step adds 5 and 2 together and assigns the value 7 to variable `y`. Then 4 is added to the result of the inner subexpression, resulting in the value 11 being assigned to variable `x`.

Although it's uncommon to see the assignment operator used in the manner just demonstrated, the benefit of this aspect of the assignment operator becomes clearer once we cover conditional looping constructs later in this appendix.

Now that we've covered how assignment operators can be used in larger expressions, let's look at C's pre- and post-increment and pre- and post-decrement operators. These operators can be considered as a shorthand way to increment or decrement a variable by 1; they're represented by two ++ or -- symbols, as demonstrated here:

```
int x, y, z;

x = 5;
y = ++x + 10;
z = x++ + 10;

NSLog(@"X is %d, Y is %d, Z is %d", x, y, z);
```

This code results in the variable `x` having the value 7 and `y` and `z` both having the value 16. The ++`x` and `x`++ operators each contributed an increase of 1 to the value of

x but each clearly had a different impact on the larger expression they're contained in because both y and z evaluate to the value 16 even though the value of x changes value after each statement is executed.

The expression $y = ++x + 10$ can be rewritten as $y = (x = x + 1) + 10$, which is similar in nature to the expressions discussed in the previous section. Variable x (which starts off as the value 5) is assigned a new value of $x + 1$, resulting in x equaling 6. This updated value is then added to 10 to produce the result of 16, which is then stored in variable y.

The next expression, $y = x++ + 10$, performs the same set of operations but in a different order. In this case, the current value of x (which at the start of executing this line is 6) has 10 added to it. This results in variable z being assigned the value 16, and it's only after this assignment that variable x is increased by 1.

The pre-increment operator ($++x$) increases the value of the variable and then uses the result in the larger expression, while the post-increment operator ($x++$) uses the current value of the variable in the larger expression and then increments the variable after the fact. Similar $--x$ and $x--$ operators perform pre- and post-decrement operations.

B.2.5 Operator precedence

Some expressions are ambiguous and can be interpreted in multiple ways. For example, should the following statement result in variable x having the value 20 or 14?

```
int x = 2 + 3 * 4;  
int y = (2 + 3) * 4;  
int z = 2 + (3 * 4);
```

C implements a set of precedence rules that specify the order in which expressions should be calculated (multiplication and division before addition and subtraction, for example). These rules mean that variable x will have the value 14.

If you need to override the precedence rules, you can use parentheses to explicitly control the order in which operations are calculated, as demonstrated by variables y and z.

This concludes our look at expressions and how to specify them in a C-based application. One type of expression that deserves greater attention is the conditional expression. Expressions of this form enable you to check the state of the data stored in an application and come up with a single Boolean (true or false) value that represents the truth of a potentially complex situation. What we haven't covered is how to use such a value to change the behavior and flow of execution in an application. C provides the answer in a set of statements collectively called the *conditional statements*.

B.3 Conditional statements

Applications that execute a fixed sequence of commands are not the most exciting or practical. Invariably, an application of any complexity must make conditional decisions

based on the value of variables or actions of the user and change their behavior or flow of execution to suit.

Making conditional decisions is a fundamental part of any programming language, and C is no different.

B.3.1 The if-else statement

The if-else statement is the simplest of the conditional statements. It evaluates a single Boolean expression and executes one set of statements if it evaluates to true and another set of statements if it evaluates to false. The general form is as follows:

```
if (condition) {
    statement block 1
} else {
    statement block 2
}
```

The condition can be any Boolean expression. If the expression evaluates to true (any value other than zero), statement block 1 is executed; otherwise, statement block 2 is executed. No matter which path is selected, execution then proceeds to the first statement immediately after the if statement.

The following if statement checks to see if the value of variable *x* holds a value greater than 10 and prints a different result depending on the result of this comparison.

```
int x = 45;
if (x > 10) {
    NSLog(@"x is greater than 10");
} else {
    NSLog(@"x is less than or equal to 10.");
}
```

The else keyword and associated statement block are optional and don't need to be included if you have no statements you wish to execute whenever the specified condition evaluates to false. The following if statement prints a message only if variable *x* is greater than 10.

```
int x = 45;
if (x > 10) {
    NSLog(@"x is greater than 10");
}
```

In this case where only one statement is required to be executed in a statement block, it's not necessary to wrap the statement in curly braces. The semicolon at the end of the statement is still required, though, because C uses the semicolon to terminate the current statement rather than a separator between multiple statements, as in some other languages. The following if statement is identical in function to the previous one:

```
int x = 45;
if (x > 10)
    NSLog(@"x is greater than 10");
```

Omitting the curly braces is subject to a stylistic debate: many developers advocate always using curly braces around the statements conditionally executed by an `if` statement as a way to add clarity and avoid easy-to-make yet often hard-to-spot mistakes, such as adding a second statement immediately under the current call to `NSLog` and expecting it to be covered by the `if` statement.

Many conditions don't fall into a simple true/false type of comparison. For example, your application may ask for the user's age and want to perform different tasks if the user is deemed to be a child, teenager, adult, or senior citizen. By stringing together multiple `if-else` statements, it's possible to check for a range of related conditions. As an example of this approach, the following listing demonstrates using multiple `if-else` statements to compare the current heading of a digital compass.

Listing B.1 Chaining multiple `if-else` statements together

```
enum direction currentHeading = North;

if (currentHeading == North)
    NSLog(@"We are heading north");
else if (currentHeading == South)
    NSLog(@"We are heading south");
else if (currentHeading == East)
    NSLog(@"We are heading east");
else if (currentHeading == West)
    NSLog(@"We are heading west");
else
    NSLog(@"We are heading in an unknown direction");
```

Although it may look like we have a slightly different statement structure, there's nothing new in this code sample. We have a series of `if` statements where the `else` clause consists of another `if-else` statement specified without curly braces.

B.3.2 *The conditional operator*

The conditional operator, represented by a question mark (?), can be considered a shorthand syntax for a common form of `if-else` statement. Consider an `if-else` statement used to conditionally assign a variable a new value based on a specific condition:

```
int x;
if (y > 50)
    x = 10;
else
    x = 92;
```

At the end of execution, this statement variable `x` will store the value 10 if the expression `y > 50` evaluates to true or the value 92 if it evaluates to false. Because this type of conditional assignment is common in many applications, C provides an alternative syntax designed to make such cases more concise. As an example, the following statement is equivalent in behavior to the previous one, yet requires only a single line of source code:

```
int x = (y > 50) ? 10 : 92;
```

The general format of the conditional operator statement is as follows:

```
condition_expression ? expression1 : expression2
```

When a conditional operator is detected, the condition expression is evaluated. If the result of this expression evaluates to `true`, `expression1` is then evaluated and becomes the result of the conditional operator. If the condition expression evaluates to `false`, `expression2` is evaluated and becomes the result of the conditional operator.

B.3.3 The switch statement

Chaining `if-else` statements together, as shown in listing B.1, is such a common practice that C provides a special statement, the `switch` statement, to simplify their specification and construction. The `switch` statement is especially handy when you're using enumerated data types and want to perform a different action for each possible value. Its general form is shown in the following listing.

Listing B.2 General syntax of a switch statement

```
switch (expression)
{
    case value1:
        statements;
        break;

    case value2:
        statements;
        break;

    default:
        statements;
        break;
}
```

The integer expression in parentheses is evaluated, and the resulting value is compared sequentially against the constant values provided by one or more case blocks. Once a match is found, the statements that follow are executed until the `break` keyword is found. This keyword signals the end of a particular case, and execution proceeds immediately to the first statement after the end of the entire `switch` statement.

If no case matches the current value of the expression, execution can proceed to a special catch-all case identified by the keyword `default`. The use of a default clause is optional.

As an example, listing B.1 could be rewritten to make use of a `switch` statement as shown in the following listing.

Listing B.3 Using the switch statement

```
enum direction currentHeading = North;

switch (currentHeading) {
    case North:
        NSLog(@"We are heading north");
        break;
```

```

case South:
    NSLog(@"We are heading south");
    break;

case East:
    NSLog(@"We are heading east");
    break;

case West:
    NSLog(@"We are heading west");
    break;

default:
    NSLog(@"We are heading in an unknown direction");
}

```

Comparing listing B.1 to listing B.3, you can see that the different syntax and the amount of whitespace makes a clear impact on the readability and maintainability of such a statement. One thing worth noticing with `switch` statements is that they're an exception to the general rule that multiple statements must be enclosed in a set of curly braces. In each case block, it's possible to simply list multiple statements one after another, and they'll all be executed in sequence if their case value is a match.

In listing B.3 the end of each case block is punctuated by the `break` keyword, which signals the end of a particular case block and causes execution to jump to the end of the entire `switch` statement. The `break` keyword is optional; if it isn't present, execution follows down into the next statement even if that statement is part of another case block. This feature is commonly used if you want to handle multiple potential values in the same or a similar manner. As a somewhat contrived example, take a look at the following listing.

Listing B.4 `switch` statement with case statements falling into each other

```

enum { Horse, Pony, Cat, Kitten } pet = Kitten;

switch (pet)
{
    case Kitten:
        NSLog(@"This is a young cat so turn on the fire");
        // Falls through
    case Cat:
        NSLog(@"Cats can sleep inthe living room");
        break;

    case Horse:
    case Pony:
        NSLog(@"These pets don't belong inside the house!");
        break;
}

```

The code is straightforward in the case of the `pet` variable storing the value `Cat` or `Pony`. For both values, the `switch` statement executes the matching calls to `NSLog` and then jumps to the end of the `switch` statement via the `break` keyword.

Not as straightforward is the `Kitten` case, which doesn't end in a `break` statement. If the `switch` statement is executed while the `pet` variable stores the value `Kitten`, the first call to `NSLog` is executed, and execution proceeds to the next statement. Because the `Kitten` case has no additional statements, execution "falls through" into the `Cat` case block, and the second call to `NSLog` is executed.

It's not necessary for a case block to have any statements, as in `Horse` case: notice it immediately falls through into the case for `Pony`. This feature is handy when multiple values should be treated identically.

With your newfound knowledge of conditional statements, you can alter the sequence of operations your applications perform. And the C developer's toolbox has even more tricks. As in real life, the path an iPhone application executes is not straight; it can contain many twists and turns. You may want to repeat a set of statements or continue doing something until a certain condition becomes true, such as a user pressing a button or an array running out of data. Looping statements with keywords such as `for` and `while` are a natural progression from conditional statements and enable an application to repeat a set of statements.

B.4 Looping statements

In your application logic you may come across the need to repeat a particular statement a fixed number of times. In chapter 1 you printed "Hello, World!" to the debug console with the following statement:

```
NSLog(@"Hello, World!");
```

If you wanted to say "Hello, World!" three times in a row, you might be tempted to add two calls to `NSLog`:

```
NSLog(@"Hello, World!");  
NSLog(@"Hello, World!");  
NSLog(@"Hello, World!");
```

Using an `if` statement, it's possible to extend this approach to make the number of times you print "Hello, World!" conditional on the value stored in a variable. For example, the code in the following listing uses the value of variable `n` to determine how many times to print "Hello, World!" to the console (between 0 and 3 times).

Listing B.5 Controlling how many times a message is logged based on program state

```
int n = 2;  
  
if (n > 0)  
    NSLog(@"Hello, World!");  
if (n > 1)  
    NSLog(@"Hello, World!");  
if (n > 2)  
    NSLog(@"Hello, World!");
```

While this technique is perfectly acceptable, it may not be the best or most scalable approach. As an example, try to extend this technique to print "Hello, World!"

anywhere between 0 and 1000 times. You'll quickly find that you must write a large amount of duplicative code, and the code is fairly hard to maintain, presenting a lot of opportunity to introduce errors. It's also hard to make changes, such as changing the code to print "Hello, Objective-C!" rather than "Hello, World!".

Luckily, C provides a better alternative to such scenarios with a set of statements that allow you to programmatically loop a specified number of times or while a certain condition continues to hold true.

The first such statement we investigate is the `while` loop.

B.4.1 *The while statement*

The `while` statement allows you to execute a set of statements as long as a specific condition continues to hold true. The general form of a `while` statement is as follows:

```
while (condition) {  
    statement block  
}
```

When this statement is executed, the condition is evaluated. If the expression evaluates to `false`, execution skips the statements in the block and proceeds to the first statement after the `while` loop. If the condition evaluates to `true`, the statements in the block are executed before execution returns to the top of the loop, and the condition is reevaluated to determine if another pass through the loop should be made.

As an example, the `while` loop in the following listing causes "Hello, World!" to be printed 15 times.

Listing B.6 A flexible way to log an arbitrary number of messages

```
int x = 0;  
while (x < 15)  
{  
    NSLog(@"Hello, World!");  
    x = x + 1;  
}
```

Execution through this loop is controlled by variable `x`, which is initialized to the value 0. The `while` loop condition is evaluated to determine if `x` is less than 15, which at this point it is, so the condition initially evaluates to `true`, and the statements in the `while` loop are executed.

Each time through the loop, a call to `NSLog` is made to emit another copy of the string "Hello, World!", and the value of `x` is incremented by 1. Execution then returns to the top of the `while` loop, and the loop condition is reevaluated to determine if another loop through the statement block is required.

After the 15th time through the `while` loop, the condition `x < 15` no longer holds true, so the looping stops and execution proceeds to the first statement after the `while` loop.

One important thing to notice with `while` loops is that because the condition is evaluated at the beginning of the loop, it's possible for the statements in the braces to never be executed. This occurs if the looping condition evaluates to `false` on the initial check.

Using a `while` loop, it's now relatively easy to change the number of times the "Hello, World!" string is printed without resorting to copy-and-paste coding. You can modify the value of variable `x` to control the number of times through the loop.

You don't have to restrict your `while` loops to a fixed number of loops. It's possible to use any condition you like to control execution of the `while` loop. The following listing demonstrates a more complex looping condition that loops until a sum of numbers reaches a specific target.

Listing B.7 A `while` loop doesn't need to loop a fixed number of times

```
int numbers[11] = {1, 10, 0, 2, 9, 3, 8, 4, 7, 5, 6};
int n = 0, sum = 0;

while (sum < 15)
{
    sum = sum + numbers[n];
    n = n + 1;
}

NSLog(@"It takes the first %d numbers to get the sum %d", n, sum);
```

This `while` loop is designed to add numbers from the `numbers` array in the first line until the sum exceeds the value of 15, as specified by the `while` loop condition. It then prints to the debug console the actual sum and how many numbers were required to reach it.

B.4.2 The `do-while` statement

A slight variation of the `while` loop is the `do-while` loop, which moves the loop condition to the end of the loop instead of the beginning. The general form of this statement is as follows:

```
do
{
    Statement block
} while (condition);
```

The `do` loop holds the distinction of being the only loop construct in C that will perform at least one iteration through the statements in the loop no matter what the value of the condition is. This is because the condition is tested at the end of the loop. One full execution through the loop must be performed in order to reach the condition at the end. The following listing demonstrates converting listing B.6 into a `do-while` loop.

Listing B.8 Using a `do-while` loop to log an arbitrary number of messages

```
int x = 0;
do {
    NSLog(@"Hello, World!");
    x = n x 1;
} while (x < 15);
```

Notice that listings B.8 and B.6 are similar in style and behavior. In this example, the only difference is in the behavior that occurs when the value of variable `x` is modified so that it initially has a value of 15 or higher. If `x` is initially set to the value 16, for example, the `do-while` loop version emits one "Hello, World!", whereas the `while` loop version does not.

B.4.3 *The for statement*

Although `while` and `do` loops are flexible, they aren't the most concise of statements to understand at a glance. To determine how many times a given `while` loop may loop or under what conditions it will exit, you must look for logic potentially spread across a wide number of individual statements on a number of lines of source code.

A `for` loop is designed to make the common scenario of looping a specific number of times easier, yet still provide a flexible construct capable of more complex looping conditions.

Looking at the various code listings containing `while` and `do` loops, notice that they generally have three things in common:

- A statement that initializes a variable to an initial value
- A condition that checks the variable to determine if the loop should continue looping
- A statement to update the value of the variable each time through the loop

A `for` loop can specify all three components in a single, concise statement. The general form of a `for` loop is as follows:

```
for (initialization_expr; loop_condition; increment_expr)
{
    statement_block;
}
```

The three expressions enclosed in parentheses specify the three components previously outlined and determine how many times the statements in the `for` loop will be executed. The flexibility of three individual expressions allows for a number of variations, but the most common arrangement is to configure a `for` loop to loop a fixed number of times.

In this scenario, the initialization expression is executed once to set the initial value of a loop counter. The second component specifies the condition that must continue to hold true in order for the loop to continue executing, and the last component is an expression that's evaluated at the end of each cycle through the loop to update the value of the loop counter.

As in a do loop, the condition expression is evaluated before the first execution of the loop. This means the contents of a for statement aren't executed at all if the condition initially evaluates to false.

The following is an example of how a for loop could be used to print "Hello, World!" 10 times.

```
int t;
for (t = 0; t < 10; t = t + 1) {
    NSLog(@"Hello, World!");
}
```

You can see the initialization expression setting the loop counter (*t*) to 0, the condition expression checking if the loop should continue to loop (until *t* is greater than or equal to 10), and the increment expression increasing the value of the loop counter by 1 for each time around the loop.

As mentioned previously, the three expressions of a for statement provide great flexibility. It's not necessary for all three expressions to be related to one another, and this enables some interesting looping conditions to be configured. For example, you can rewrite the code in listing B.7, which summed numbers until the sum became larger than 15, to use a for loop:

```
int numbers[11] = {1, 10, 0, 2, 9, 3, 8, 4, 7, 5, 6};
int n, sum = 0;

for (n = 0; sum < 15; sum += numbers[n++])
    ;

NSLog(@"It takes the first %d numbers to get the sum %d", n, sum);
```

Notice how the initialization, condition, and increment expressions don't all refer to the same variable. The initialization statement sets the loop counter *n* to 0, yet the condition that determines if the loop should continue executing checks if *sum* is less than 15. Finally, each time through the loop, the increment expression adds the current number to the sum and, as a side effect, also increments the loop counter *n* to index the next number.

All of the required behavior is specified in the three expressions that make up the for loop, so no statements need to be executed each time through the for loop; the sole semicolon represents an empty, or "do nothing," statement.

B.4.4 Controlling loops

The flow of execution through all the loop constructs demonstrated so far is controlled by the loop condition that occurs at the beginning or end of each iteration through the loop. Sometimes, jumping out of a loop early or immediately reevaluating the loop condition is advantageous. C provides two statements, `break` and `continue`, to achieve these goals.

BREAK

The break statement can be used to exit a loop immediately, jumping over any additional statements in the loop body and not reevaluating the loop condition. Execution jumps to the first statement after the while or for loop. As an example, the following listing is a modification of listing B.7; it contains an if statement that causes the while loop to break out early if a 0 is found in the array of numbers.

Listing B.9 Breaking out of a while loop early

```
int numbers[11] = {1, 10, 0, 2, 9, 3, 8, 4, 7, 5, 6};
int n = 0, sum = 0;

while (sum < 15)
{
    if (numbers[n] == 0)
        break;

    sum = sum + numbers[n];
    n = n + 1;
}

NSLog(@"It takes the first %d numbers to get the sum %d", n, sum);
```

The break statement immediately exits the while loop whenever the value 0 is detected. Execution proceeds directly to the call to NSLog, skipping the additional statements in the while loop.

This example is a little contrived: you could have also rewritten the while loop without the break statement by using a more complex looping condition, as demonstrated by the following listing.

Listing B.10 Reworking the loop condition to remove the break statement

```
int numbers[11] = {1, 10, 0, 2, 9, 3, 8, 4, 7, 5, 6};
int n = 0, sum = 0;

while (sum < 15 && numbers[n] != 0)
{
    sum = sum + numbers[n];
    n = n + 1;
}

NSLog(@"It takes the first %d numbers to get the sum %d", n, sum);
```

In this version, the if and break statements are replaced with a compound loop condition that continues looping only if the expressions on both sides of the && operator hold true. Comparing this logic to that of listing B.9, you'll see they're identical in behavior: as soon as the sum becomes greater than or equal to 15 or a 0 is detected, the loop stops.

The break statement is helpful in more complex code when you want to perform additional processing before exiting the loop early or when it's possible to determine you need to exit the loop only after performing complex calculations that don't fit suitably into a single expression.

CONTINUE

The `continue` statement can be used as a shortcut to cause the current iteration of the loop to be prematurely ended, in effect causing the loop condition to immediately be reevaluated. For example, the following listing demonstrates a loop that skips over any numbers greater than 5.

Listing B.11 Stopping the current execution of a loop early with `continue`

```
int numbers[11] = {1, 10, 0, 2, 9, 3, 8, 4, 7, 5, 6};
int n = 0, sum = 0;

do{
    if (numbers[n] > 6)
        continue;

    sum = sum + numbers[n];
} while (n++ < 11 && sum < 15);

NSLog(@"It takes the first %d numbers to get the sum %d", n, sum);
```

The loop condition causes the loop to repeat while `n` is less than 11 (indicating the last number in the `numbers` array hasn't been passed) and the sum is currently smaller than 15.

Each iteration through the loop adds the next number to the running sum. But the number is first checked if it's greater than 6, and if so, the `continue` statement skips the addition and immediately processes the next number in the `numbers` array.

Notice the increment of the loop counter variable, `n`, was moved out of the loop and into the loop condition expression. If it hadn't been moved but instead was incremented similarly to previous code samples, it would've become stuck after the first number greater than 6 was found. In this situation, the number would've caused the `continue` statement to be executed, skipping the rest of the statements in the loop, but nothing would've incremented the value of `n` to cause the next iteration through the loop to look at the next number in the array.

It's important to also note that the `break` and `continue` statements can exit only out of the immediate `while`, `do`, or `for` loop in which they're contained. It's not possible to break out of multiple loops if more than one loop is nested inside of another.

B.5 Summary

Because Objective-C is a strict superset of C, it's important to have a strong understanding of the principles behind C before branching out to learn about the additions Objective-C brings to the table. This appendix offers you a firm foundation on which to start your Objective-C learning. It's in no way a complete coverage of C-based programming, though, and many details have been left out.

It's clear that Objective-C owes a great debt to the C programming language. With C-based knowledge, you can perform calculations, make decisions, and alter the flow of execution throughout your applications. Objective-C expands upon the C foundations with its own flavor of object-oriented extensions.

appendix C

Alternatives to Objective-C

Steve Jobs, one of the original founders of Apple Inc., has overseen a number of innovations and exciting products in the computer industry since Apple was founded in 1976.

Two of the most recent of these are undoubtedly the iPhone and iPad. There's no denying that the iPhone has made an impact on the smartphone marketplace. Software developers, device manufacturers, and telecommunication carriers can attribute to the iPhone's presence at least some kind of impact on or change in their industries, whether it's an increased interest in innovative and visually appealing UIs or the increased use of cellular data services and downloadable content.

If the iPhone is your first foray into a platform developed by Apple, the required development tools and languages are likely to feel foreign and perhaps even esoteric or antiquated compared to your current platform because of their rather different origins.

Much as the iPhone hardware can trace its roots to a long line of prior iPod devices, Objective-C and Cocoa Touch can follow their long lineage and history back more than 25 years. The iPhone is as much a culmination and refinement of existing technologies as it is a breakthrough design.

In this appendix we discuss some of the alternatives to developing iOS applications in Objective-C, but first, it's important to understand the origins of Objective-C.

C.1 A short history of Objective-C

The late 1970s and early 1980s saw a lot of experimentation in improving software developers' productivity and the reliability of the systems they were producing. One train of thought was that gains could be found with a shift from procedural-based programming languages to languages incorporating object-oriented design principles.

Smalltalk, a language developed at Xerox PARC in the 1970s, was the first to introduce the term *object-oriented programming* to the general developer community.

The language was first widely distributed as Smalltalk-80 in 1980, and ever since, Smalltalk has left its mark on a number of more recent programming languages.

C.1.1 The origins of Objective-C

In the early 1980s, Dr. Brad Cox and his company, Stepstone Corporation, experimented with adding Smalltalk-80-style object-oriented features to the preexisting and popular C programming language. They soon had a working prototype, which they called Object-Oriented Programming in C (OOPC).

This language continued to evolve, and in 1986, Brad Cox published *Object-Oriented Programming: An Evolutionary Approach*, a book that outlines the original description of what had by then become Objective-C. This was a clean layer of object-oriented features on top of the C programming language. Early implementations consisted of a preprocessor that turned Objective-C code into C source code, which was then processed by a standard C compiler. Being a true superset of C, Objective-C had the advantage of being familiar to existing C developers, making it possible to compile any valid C program with an Objective-C compiler. This enabled a high level of code reuse with existing C-based libraries.

C.1.2 Popularization via NeXT Inc.

In 1988, NeXT Inc. (founded by Steve Jobs when he left Apple in 1985) licensed Objective-C from Stepstone Corporation and developed its own compiler and runtime library for the language. The compiler and runtime library were used as the foundations of the development environment for the NeXTStep operating system, which powered its new, innovative, and some would say well-ahead-of-its-time, high-end workstations, such as the NeXT Computer and NeXTCube.

The NeXTStep operating system has a revered position in computer history. It garnered widespread respect as an innovative platform, although it was also criticized for its expensive price point compared to commodity IBM PCs of the day. But many innovations in a number of fields are attributed to users of NeXTStep-based computers. For example, Sir Tim Berners-Lee developed the first web browser on a NeXT computer and had the following to say about the development experience:

I wrote the program using a NeXT computer. This had the advantage that there were some great tools available—it was a great computing environment in general. In fact, I could do in a couple of months what would take more like a year on other platforms, because on the NeXT, a lot of it was done for me already.

“The WorldWideWeb Browser,”
(www.w3.org/People/Berners-Lee/WorldWideWeb)

Even the now classic DOOM and Quake games from id Software have elements of their development history intertwined with Objective-C and NeXTStep hardware. John Romero reflected recently on why he’s still passionate about NeXT computers these many years later:

Because we at id Software developed the groundbreaking titles DOOM and Quake on the NeXTSTEP 3.3 OS running on a variety of hardware for about 4 years. I still remember the wonderful time I had coding DoomEd and QuakeEd in Objective-C; there was nothing like it before and there still is no environment quite like it even today.

“Apple-NeXT Merger Birthday,” 2006 (<http://rome.ro/labels/apple%20next%20doom%20quake.html>)

C.1.3 Adoption and evolution by Apple Inc.

In 1996 NeXT Inc. was acquired by Apple Inc., and Steve Jobs returned to the helm of Apple. A lot of the NeXTStep technologies that Tim Berners-Lee and John Romero were so enamored of eventually found their way into Mac OS X, first released to the public in 2001.

As well as inheriting the Objective-C-based programming model for application development, Mac OS X incorporated numerous NeXTStep GUI concepts. Looking at the OPENSTEP screenshot in figure C.1, for example, you’ll see that a number of iconic Mac OS X features, such as the dock, had their origins in NeXTStep.



Figure C.1 Screenshot of the OPENSTEP 4.2 Desktop, showing many Mac OS X-like features

C.2 Alternatives to Objective-C and Cocoa

Out of the box, Xcode and the iPhone software development kit (SDK) support development of applications in C, C++, and their Objective-C-based variants, but this doesn't mean they're the only options open to developers. A number of third parties offer alternative development tools to suit the needs and backgrounds of almost any developer.

Of these alternatives, a lot of attention has been concentrated on those that enable applications to be developed in scripting languages such as Lua or Ruby or that use technologies such as HTML, Cascading Style Sheets (CSS), and JavaScript, which are more familiar to client-side web developers. It's commonly perceived that these types of tools can offer a quicker and perhaps more productive work environment, much in the way that Objective-C and other Smalltalk-inspired languages were considered an improvement over C and C++.

C.2.1 Close to home: Objective-C++ and plain old C or C++

If you've developed for another mobile platform, you likely have some background in C or C++. A lot of third-party support libraries for a magnitude of purposes, such as speech synthesis, physics engines, communication, and image analysis, to name a few, are also developed in these languages.

Xcode can compile C (.c) or C++ (.cpp) source code in an iPhone project just as easily as it can compile Objective-C (.m) files. Simply add the files to your project and allow Xcode to build your project as normal. A number of the key frameworks in iOS, such as Core Graphics, are C-based APIs, so you're already familiar with integrating code developed in C or C++ with an Objective-C project.

Unlike Mac OS X, though, iOS doesn't provide a C-based API for creating GUIs (there's no equivalent of Carbon). This means that unless you're interested in developing a completely custom UI with OpenGL (another C-based API), your use of C or C++ source code will probably be restricted to behind-the-scenes logic, which will then be interfaced to a UI written in Objective-C. This isn't a bad approach if you want to share a core set of logic across a number of platforms that support development in C or C++, such as iOS, Android, Symbian, and Windows Mobile. You can develop your core logic with C or C++ and then wrap a platform-specific UI layer around it for each platform you want to support.

Objective-C++ is interesting because it's, quite literally, the object-oriented features of Objective-C applied on top of the C++, not C, programming language. This combination leads to some interesting side effects. As an example, it isn't possible to derive a C++ class from an Objective-C class, because no attempt was made to unify their distinct type systems.

C.3 The iPhone SDK: Safari

When the original iPhone was released, the only way to extend the platform was by deploying web-based applications. As an Apple press release of the time noted:



Figure C.2 Spot the difference: the Facebook web application on the left and the Facebook native application. The web application lacks certain features, such as access to the camera and photo library.

Developers can create Web 2.0 applications which look and behave just like the applications built into iPhone, and which can seamlessly access iPhone’s services, including making a phone call, sending an email and displaying a location in Google Maps.

“iPhone to Support Third-Party Web 2.0 Applications,”
(www.apple.com/pr/library/2007/06/11iphone.html)

From this comment, it’s hard to imagine why you’d learn Objective-C—a technology that finds little use outside of Apple-based platforms. It appears that reusable skills such as HTML, CSS, and JavaScript are a better approach. An iPhone web application can be as simple as a website rendered by the iPhone’s Safari web browser. By altering layout, CSS styling, and so on, a web application can achieve fairly good parity with the look and feel of native applications, as shown in figure C.2.

One advantage of web-based applications is their ability to automatically and instantaneously update to a newer version. When you update the source code hosted on your web server, all users are immediately upgraded; in fact, the user has no option to decide if or when they upgrade. On the negative side, however, the iTunes App Store provides a simple, economic model to allow you to monetize and charge for your applications. Finding a way to monetize web applications, which can’t be hosted in the iTunes App Store, is left as an exercise for the developer.

C.3.1 *HTML5, CSS3, and other modern standards*

Web-based applications require a connection to the internet so their source code can be downloaded from the web server. This appears to rule out their use in scenarios such as airplane flights or subway travel, where radio transmitting devices are either prohibited or

have potentially patchy (or no) coverage. Mobile Safari offers support for an HTML5 feature called the Offline Application Cache that offers a solution to this limitation. It allows a manifest file to be written that outlines the files (HTML, JavaScript, CSS, and image files) a web application requires. Safari downloads all files listed in the manifest and ensures they're available while offline. Provided your web application doesn't depend on server-side functionality, your application will happily run while the device is offline.

The Offline Application Cache is only one of many advanced technologies implemented by Safari. Other examples include extensions to CSS to support rounded edges, shadows, transforms, animations, and transitions; rich canvas and SVG support for vector-based client-side drawing; and even APIs to store data on the client side.

Mobile Safari is a world-class mobile browser designed to enable the development of serious web-based applications. As evidence of some of the powerful UI elements that can be crafted with pure HTML, CSS, and JavaScript, take a look at the various demos available on Matteo Spinelli's blog at <http://cubiq.org>. As seen in figure C.3, many of the UI elements seen in Apple and third-party iOS applications can be replicated easily in a web-based environment.

An intrinsic advantage of developing using standardized web-based technologies is their inherent ability to support more than one platform. With tweaks to your HTML and CSS, it's often possible to develop a web application that provides fairly good native look and feel across a wide range of mobile platforms, such as Microsoft's Windows Mobile (now called Windows Phone), Research In Motion's Blackberry, Palm's WebOS, and Nokia's various platforms. With Mobile Safari leading the charge, most if not all of these alternative platforms have equally rich web browser support. This isn't to say you're left with a mediocre application that can't access device-specific functionality. An iPhone web application has access to a number of iPhone-specific device capabilities and features.

C.3.2 iPhone OS integration

iPhone integration for web applications covers not only access to hardware features such as the GPS, accelerometer, and compass, but also aspects that help make web applications feel more like a natural part of the operating system. For example, by adding the following line of HTML to your main page, you can enable a static image to be displayed as a splash screen while your web application HTML and JavaScript source code is downloaded, parsed, and rendered behind the splash screen: `<link rel="apple-touch-startup-image" href="img/splash.png" />`.

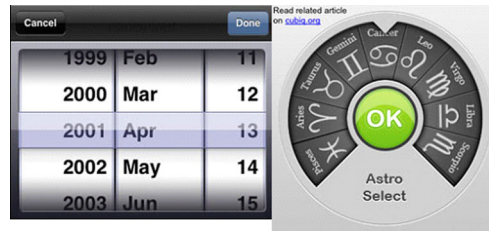


Figure C.3 Examples from <http://cubiq.org>, the blog of Matteo Spinelli, demonstrating the use of CSS3 transitions, animations, transformations, JavaScript, and HTML to build web-based UIs that approach the behavior and look and feel of native iPhone applications

Similar tags can be used to enable your application to display full screen without any window chrome from the Safari web browser and to alter other aspects of your web page's presentation, such as its ability to zoom in on the content, something commonly needed for web pages originally designed for desktop-sized screens but typically unexpected and unnecessary for content specifically designed for mobile devices.

Another tag worth mentioning with respect to iOS integration of web applications is `<link rel="apple-touch-icon" href="img/icon.png"/>`. This tag is an important one to set on mobile web applications. The image referenced by this tag will appear on the device's home screen if the user navigates to the web application by typing in its URL and then using the Add To Home Screen option. Tapping the icon will launch the web application without the user needing to type in a URL, just like a native application. This feature combined with others, such as the Offline Application Cache, enables web applications to essentially look and feel like a native application—no browser, window chrome, bookmarks, or URLs in sight.

On the device hardware access side of things, extensions to HTML, CSS, and JavaScript allow access to multitouch, accelerometer, and location functionality, which have become common to most iPhone applications. Listing C.1 demonstrates a simple web application that uses the W3C Geolocation API (www.w3.org/TR/geolocation-API/) to display where the device viewing the web page is currently located. This would be ideal as a starting point for a web application that provides location-specific search results.

Listing C.1 Web application using the W3C Geolocation API

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Example of Geoposition API</title>
  </head>
  <body>
    <script language="javascript" type="text/javascript">
      function locationFound(position) {
        alert('You are currently located at '
          + position.coords.latitude + ", " + position.coords.longitude);
      }

      function errorOccurred(error) {
        alert('An error occurred trying to determine your location. '
          + error.message);
      }

      if (navigator.geolocation)
        navigator.geolocation.getCurrentPosition(locationFound, errorOccurred);
      else
        alert('Your device does not support the geolocation API');
    </script>
  </body>
</html>
```

It's important to note that web applications currently—and probably always will—have less access to hardware features and operating system services than do Objective-C applications. Examples of features currently accessible only from Objective-C include the user's address book, camera photo library, and iPod music. Constant improvements are being made in this regard; for example, iPhone OS 3.0 introduced the capability for web applications to access GPS location information, and iOS 4.2 added access to accelerometer data. But the web application platform will always present the developer with a subset of the features accessible to native developers, if only because Apple engineers must expend additional effort to produce JavaScript-accessible wrappers for any feature they want to expose to web applications, and additional security and privacy concerns arise with features that can be accessed by any website.

C.3.3 PhoneGap and other extensible cross-platform solutions

Developers need not abandon HTML, CSS, and JavaScript as their development platforms if they find the built-in services of the Safari web browser and web-based applications inadequate for their application needs.

A number of solutions exist to enable applications written in web-based technologies to run outside of the traditional Safari browser-based environment and, in the process, to gain greater access to system hardware. One example of this approach is a development tool called PhoneGap (www.phonegap.com). PhoneGap is an open source product that enables you to build applications in HTML and JavaScript. Unlike a web application running in Safari, however, your JavaScript source code also has access to the PhoneGap SDK APIs, which provide access to additional features of the mobile platform, such as Geolocation, Vibration, Accelerometer, Contacts, and Sound.

This accessibility is achievable because a PhoneGap application is a native iOS application, built using Objective-C and Xcode, just as we've discussed throughout this book. Most of the application is a predeveloped Objective-C component (which for the most part is hidden and automatically inserted by the project template) that sets up a `UIWebView` control to display the HTML and JavaScript content embedded in your Xcode project. The magic occurs because the Objective-C component exposes additional services that your JavaScript can hook into. All your web application must do is include a `<script/>` element that imports the main PhoneGap JavaScript file.

Appearing as a standard Objective-C-based iOS application, at least from the outside, means PhoneGap applications can be distributed via the iTunes App Store, which provides an easy monetization and consumer discovery/marketing approach.

Because the core of your PhoneGap application is written in HTML, CSS, and JavaScript, it's fairly portable across most mobile platforms, so it shouldn't surprise you that the native part of PhoneGap is available for a number of platforms, including Android, Windows Mobile, iPhone, and Blackberry. By combining your platform-agnostic JavaScript with a device-specific shell that deals with setting up a web browser control, then wrapping the device's features into a consistent PhoneGap JavaScript-based API, you can have your cake and eat it too—developing iPhone

applications in HTML, CSS, and JavaScript without having to give up too much cross-platform portability.

A similar solution is Titanium Mobile (www.appcelerator.com/products/titanium-mobile-application-development/). A unique feature of Titanium is that it provides access to the entire iPhone or Android UI feature set: table views, scroll views, native buttons, switches, tabs, and popovers can all be accessed from the JavaScript in a Titanium Mobile–based application.

With both solutions, because source code is available, if you find a device feature not exposed to JavaScript, and you're comfortable with writing a little Objective-C or C, you can easily provide a JavaScript callable wrapper for the feature in question and produce your own application-specific variant of the solution.

Ultimately, the decision of web versus native iPhone development may come down to your own personal experience and comfort levels. If you're a web developer, learning how to optimize your existing web pages for viewing on the iPhone may be the quickest route forward. If you're a long-time C, C++, C#, or even J2ME developer, taking the time to learn Objective-C and Cocoa to build native applications may be more rewarding and open the possibility of code reuse from previous projects.

C.4 Scripting languages: Lua and Ruby

If an Objective-C-based application can be developed to host content written in HTML, CSS, and JavaScript, you may wonder if it's possible to develop similar shells to enable development of applications in other popular scripting languages such as Python, Ruby, or Lua. The resounding answer is, yes—although the path to App Store acceptance hasn't always been clear or straightforward in such cases.

One such product is the Corona SDK product from Anscamobile (www.anscamobile.com). Corona enables a developer to write an application in the Lua scripting language and place it in a predeveloped native application that performs a task similar to what PhoneGap did for JavaScript.

Although PhoneGap is more focused on enabling web-based content, the focus of Corona is arguably in gaming, as OpenGL-ES and similar game technologies are used heavily in the APIs the Corona SDK exposes to your Lua script.

The Corona website offers the contents of listing C.2 as an example of the Objective-C source code required to draw an image onto the screen using OpenGL-ES. Although the source code is arguably worse than it needs to be, it's hard to deny the potential productivity gains once you compare it to the one line of Lua source code required in a Corona application to perform the same task: `display.newImage("myImage.jpg", 0, 0)`.

Listing C.2 Drawing an image onscreen with OpenGL and Objective-C

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"myImage"
                ofType:@"jpg"];
NSData *texData = [[NSData alloc] initWithContentsOfFile:path];
UIImage *image = [[UIImage alloc] initWithData:texData];
```

```

if (image == nil)
    NSLog(@"Do real error checking here");

GLuint width = CGImageGetWidth(image.CGImage);
GLuint height = CGImageGetHeight(image.CGImage);
CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
void *imageData = malloc(height * width * 4);
CGContextRef context = CGContextCreate(imageData, width, height,
    8, 4 * width, colorSpace,
    kCGImageAlphaPremultipliedLast | kCGBitmapByteOrder32Big);
CGColorSpaceRelease(colorSpace);
CGRect imageRect= CGRectMake(0, 0, width, height);
CGContextClearRect(context, imageRect);
CGContextTranslateCTM(context, 0, height - height);
CGContextDrawImage(context, imageRect, image.CGImage);

glTexImage2D(GL_TEXTURE_2D 0, GL_RGBA, width, height, 0,
    GL_RGBA, GL_UNSIGNED_BYTE, imageData);

CGContextRelease(context);

free(imageData);
[image release];
[textureData release];

```

Technologies such as the Corona SDK are opening iPhone application development to a wider range of people; development isn't necessarily restricted to those with a strong programming background. A key premise to this type of technology is that the higher-level scripting languages make development more accessible and forgiving than in languages in which hard-to-detect-and-diagnose errors can occur (such as Objective-C, which allows you to get closer to the hardware). In this case, the trade-off of not having immediate access to all hardware of the operating system features can be deemed to be acceptable. If Lua isn't your cup of tea, chances are, with a little research online, you'll be able to find an equivalent solution for your language of choice. For example, Ruby developers can use an open source framework called Rhodes (<http://rhomobile.com/products/rhodes/>) to develop native applications that work across a number of mobile platforms (iPhone, Windows Mobile, RIM, Symbian, Android, MeeGo, and Windows Phone 7) while working with device capabilities such as GPS, PIM information, camera, mapping, signature capture, and Bluetooth.

The trade-off in using a platform such as Corona is that you may be restricted to a particular subset of device features supported by the product. Using a similar argument to that used for web applications, scripting languages will potentially always lag behind, if only because they require a wrapper to be developed with every API the developer wants to be accessible from the scripting language. This is especially true when you consider that many of these platforms also attempt to be cross-platform solutions so, in some cases, they purposely don't expose a feature of a specific platform if that feature would be hard or impossible to provide on another supported device type. You, in effect, obtain a lowest-common-denominator development platform.

Scripting languages also have to be careful not to run afoul of the iPhone OS SDK license agreement terms, which every developer must accept in order to develop and

submit applications to the iTunes App Store. In particular, clause 3.3.2 has been raised as a potential concern:

3.3.2 — An Application may not itself install or launch other executable code by any means, including without limitation through the use of a plug-in architecture, calling other frameworks, other APIs or otherwise. No interpreted code may be downloaded or used in an Application except for code that is interpreted and run by Apple’s Documented APIs and built-in interpreter(s).

The iPhone OS SDK doesn’t ship with a Lua or Ruby interpreter, so this clause would appear to rule out development of an application using Corona or Rhodes because it wouldn’t be possible for these products to execute your code via an Apple Documented API or built-in interpreter.

Apple appears to have a rather loose interpretation of clause 3.3.2, however, as evidenced by the wide number of applications approved in the App Store that are scripted with Lua and other languages. The general gist is that scripting is okay as long as the functionality isn’t exposed to the user, and the behavior of your application can’t change without going through the standard App Store application process. We come back to this topic later in this appendix.

C.5 The 10,000-pound gorilla in the room: Adobe Flash

Once you get your head around a solution such as Corona, you may wonder if a similar approach could be used to enable Adobe Flash content to be wrapped up and played as a standalone application on an iOS-powered device. The answer is a resounding yes, but the road has been particularly rocky for Adobe to get to this stage.

Recently rereleased, the Packager for iPhone (available as part of the Adobe Flash Professional CS5 product) enables Flash developers to convert an ActionScript 3–based project into a native iOS application that’s suitable for deployment to an iOS device as well as for submission to the iTunes App Store.

The Packager for iPhone works by compiling the Flash content into Advanced RISC Machine (ARM) code, which is then bundled with a Flash runtime. Xcode isn’t required; in fact, the tool even works on a Windows-based machine. This approach of cross-compilation potentially hit a significant roadblock when Apple updated and greatly expanded section 3.3.1 of the iPhone OS 4.0 SDK license agreement to read as follows:

3.3.1 — Applications may only use Documented APIs in the manner prescribed by Apple and must not use or call any private APIs. Applications must be originally written in Objective-C, C, C++, or JavaScript as executed by the iPhone OS WebKit engine, and only code written in C, C++, and Objective-C may compile and directly link against the Documented APIs (e.g., Applications that link to Documented APIs through an intermediary translation or compatibility layer or tool are prohibited).

This clause was widely interpreted as a shot across the bow of Adobe, because Apple had expressed its dislike of Adobe Flash and its disinterest in having Flash as part of

the iOS platform. For a while, this caused Adobe to discontinue the Packager for iPhone component of CS5, and the future remained uncertain for a number of alternative development tools (such as Corona and Rhode). All of these tools were arguably creating applications that “link to Documented APIs through an intermediary translation or compatibility layer or tool” and, for the most part, weren’t written in C, C++, or Objective-C.

The whole war of words between Adobe and Apple can perhaps be best summed up with the following excerpt from an Apple press release dated September 9, 2010 (www.apple.com/pr/library/2010/09/09statement.html), in which Apple sought to clarify the terms of the license agreement and its intent behind the changes:

We are continually trying to make the App Store even better. We have listened to our developers and taken much of their feedback to heart. Based on their input, today we are making some important changes to our iOS Developer Program license in sections 3.3.1, 3.3.2 and 3.3.9 to relax some restrictions we put in place earlier this year.

In particular, we are relaxing all restrictions on the development tools used to create iOS apps, as long as the resulting apps do not download any code. This should give developers the flexibility they want, while preserving the security we need.

One interesting side note about the Packager for iPhone is that it’s designed to work on a Windows-based PC, meaning users don’t need to purchase a Mac to develop iPhone applications. The Flash toolchain can produce iPhone applications ready for deployment to a device or submission to the iTunes App Store. Developers still are required to purchase a yearly subscription to the iPhone Developer program in order to obtain the code-signing certificates required.

C.6 Mono (.NET)

Developers familiar with the Microsoft .NET development platform may have heard of the Mono open source project, produced by Xamarin (previously Novell). The Mono project aims to deliver a compatible .NET CLR runtime environment and development tools for environments not supported by Microsoft’s own implementation (primarily various Linux- and UNIX-based systems, such as Mac OS X).

By reutilizing a number of key Mono technologies, Xamarin has created a .NET runtime environment that can run .NET applications on iOS-powered devices. This means that a developer can use Visual Studio (or Xamarin’s MonoDevelop IDE) to develop C# applications that can be deployed to iPhones. You still enjoy most of the comforts of .NET, such as a garbage collector and an extensive set of base class libraries.

MonoTouch works in a fashion similar to Adobe’s Flash solution. Because iOS doesn’t support runtime environments that require just-in-time (JIT) compilation, MonoTouch contains a compiler that’s capable of ahead-of-time (AOT) compilation. This turns Common Intermediate Language (CIL)-based assemblies into native ARM code at compile time rather than performing this process each time the application starts. One limitation is that a few base class library APIs, such as `System.Reflection.Emit`

and `Assembly.LoadFrom`, which dynamically generate or load code from external sources, aren't supported in MonoTouch.

MonoTouch also doesn't provide an implementation of any traditional .NET UI frameworks, such as Silverlight, WPF, or Winforms. Instead, through an Objective-C-to-.NET bridging technology, C# developers can interface to UIKit to generate UIs that have a 100% native look and feel, because they use the same controls and classes an Objective-C developer uses. This bridging technology also enables C# applications to access all functionality exposed by the iPhone SDK, such as address book, GPS, and accelerometer.

A number of successful applications published via the iTunes App Store were developed in .NET-based languages and internally rely on MonoTouch-based technology. An example is Raptor Copter developed by Flashbang Studios, shown in figure C.4.

MonoTouch is a commercial product that can be purchased online at <http://ios.xamarin.com/>. A free trial edition that's restricted to deploying applications to the iPhone and iPad Simulators is also available. Unlike the Adobe solution for Flash, MonoTouch still requires Xcode behind the scenes and hence is a Mac-only solution.



Figure C.4 iTunes App Store showing Raptor Copter, one of the first iPhone games developed using MonoTouch-based .NET technologies via the Unity3D engine

Xamarin announced a follow-on product from MonoTouch called Mono for Android, which brings the ability to develop .NET applications to Android-powered devices. The combination of Visual Studio, MonoTouch, and Mono for Android suddenly became a potential cross-platform play for those developers interested in supporting more than one platform. With Microsoft's Windows Phone 7 platform being .NET-based, using all three tools would enable the core of an application's logic to be developed with standard .NET-based APIs. This core logic could then be shared across all three platforms, and a thin device-specific UI layer could be crafted for each platform. The result would be three applications that look and feel totally integrated into the platforms they run on, yet without three times the development effort. As an added bonus, all development would be in a single language rather than potentially spread across three (Objective-C, Java, and C#).

C.7 Summary

With iOS becoming a more mature development platform, a wide range of development tools are available to develop your next iTunes App Store masterpiece. There's a tool to suit the needs of virtually every developer.

For someone starting with prior Mac OS X development experience, the Objective-C, Cocoa Touch, and Xcode platform will be familiar territory. It provides full and timely access to all iOS hardware and software features and is well supported by Apple. This makes Objective-C a strong default position for developing iPhone applications. Even if you select an alternative programming language to develop the core of your application, chances are you'll eventually need to reach out to Xcode and Objective-C to configure or extend the capabilities of your chosen environment.

With the mobile computing space heating up and a number of competing platforms becoming available, the concept of cross-platform support and running your application on one or more platforms becomes more important. Xcode and Objective-C fail in this sense because they find little applicability outside of the iPhone or Mac OS X. One alternative is to develop your core application logic in C or C++ and interface to Objective-C only to provide a UIKit-based UI. But increasingly, mobile platforms of importance are exposing a virtual machine-based development story that doesn't support development in C or C++ (at least not with a traditional compiler). In this regard, development on the .NET platform is worthy of further investigation. Products such as Xamarin's MonoTouch and Mono for Android provide .NET support for the iPhone and Android platforms respectively and help provide a consistent development experience across multiple platforms.

At the end of the day, your choice of development tools will come down to your personal preferences and comfort levels as a developer. The ease of rapid prototyping and protection from difficult-to-diagnose bugs offered by scripting languages such as Lua and Ruby has to be traded off against the risk that your third-party development tools may become unsupported or may not be updated to support new platform features (such as the iPhone 4 high-resolution screen or the iPad's larger size) at some

stage in the future. If you invest the farm in a technology, you must feel confident that you can either spend the resources to redevelop your application a second time around or that your development tools will survive long enough to outlast the usable lifetime of your application.

Symbols

- ^ operator 298
- operator 296
- ! operator 39, 297
- != operator 39, 220, 297
- (++x) pre-increment operator 300
- (=) assignment operator 299
- (=) operator 299
- (i) icon 285
- (x++) post-increment operator 300
- @avg operator 216
- @catch blocks 211
- @class 99
- @count operator 216
- @distinctUnionOfObjects operator 216–217
- @encode statement 90
- @finally block 211
- @implementation directive 99, 106
- @implementation section 102, 112–113, 117
- @interface declaration 100
- @interface directive 99, 117
- @interface section 106, 109
- @max operator 216
- @min operator 216
- @optional messages 171
- @private attribute 101–102
- @property attributes 111
- @property declarations 14
- @property directive 109, 112–113
- @property statement 110, 114
- @property syntax 109–112
 - method naming 111
 - setter method semantics 111
 - thread safety 111–112
 - writeability 111
- @selector(...) directive 168
- @sum operator 216
- @synthesize 214
- @synthesize directive 112–113
- @throw directive 208, 210
- @try block 211
- * character 59–60
- * operator 296
- *.h file extension 12
- *.m file extension 12
- *.xib files 15
- / operator 296
- & operator 59, 298
- && operator 39, 297
- #import statements 122
- #include statements 122
- % character 40, 42
- % operator 296
- %@ placeholders 132
- %d placeholders 132
- + button 21
- + operator 69, 296
- += assignment operator 299
- < > operator 220
- < operator 39, 221, 297
- << operator 298
- <= operator 39, 221, 297
- = operator 220
- =< operator 221
- == operator 39, 61, 69, 91, 220, 297
- => operator 221
- > operator 61, 107
- > operator 39, 220, 297
- >= operator 39, 221, 297
- >> operator 298
- | operator 298

|| operator 39, 297
 ~ operator 298

A

abort() function 254
 abstract factory design 134
 action methods, <UITableViewDelegate> 155–157
 ActionScript 3-based project 322
 actionSheet 157
 actionSheetCancel: method 158
 Ad Hoc distribution 292
 Add Devices button 291
 addEntriesFromDictionary: message 85
 addObject: message 81
 addObserver: selector 198–199
 address value 100
 addresses, obtaining for variables 59–60
 address-of (&) operator 59
 addTask method 249, 251
 addTasksObject: method 251
 addWidget 104
 Adobe Flash 322–323
 Advanced RISC Machine code. *See* ARM
 advert key 217
 advert property 217–218
 age variable 128
 aggregating values 216–217
 ahead-of-time compilation. *See* AOT
 alertView:clickedButtonAtIndex: method 227
 ALL operator 223
 allKeys method 87–88
 alloc class method 115–116, 119
 alloc message 66, 184, 190
 alloc method 127, 163, 165, 182, 192–193
 allocation, combining with initialization 118
 Allocations Trace Template 284
 alloc-based object creation process 67
 allocWithZone: message 190–192
 allProperties object 220, 227
 alternatives 315

- Adobe Flash 322–323
- C and C++ 315
- iPhone SDK 315–320
 - HTML5, CSS3, and other standards 316–317
 - integration with 317–319
 - PhoneGap 319–320
- Lua 320–322
- Mono (.NET) 323
- Objective-C++ 315
- Ruby 320–322

 anError pointer 208
 animate method 147, 150
 animationDidStopSelector 148
 animationNotification method 146

animationNotification protocol 146, 148–149
 animationStopped methods 148
 animationWillStartSelector 148
 ANY operator 223
 AOT (ahead-of-time) compilation 323
 APIs (application programming interfaces) 264–265
 App Store application 322
 App Store profile 292
 appendString method 161
 AppKit framework 5
 Apple Core Data Programming Guide 253
 Apple developer, process for becoming 288
 Apple Inc., adoption of Objective-C by 314
 Apple Operating System. *See* iOS
 Apple Provisioning Portal website 292
 Apple, block-based APIs in 264–265
 Apple-based platforms 316
 application data 228–256

- changing data model 251–253
 - Core Data framework
 - history of 229–231
 - objects 231–232
 - resources 232–234
- error handling and validation 253–256
- performance 253
- PocketTasks application 234–251
 - adding and deleting people 243–246
 - data model 235
 - defining relationships 236
 - managing tasks 246–250
 - master TableView class 240–243
 - model objects 249–251
 - Person entities in pure code 237–240
 - Xcode Core Data template 234–235

 application programming interfaces. *See* APIs
 Application section 9
 application: didFinishLaunchingWithOptions 238–239, 242
 applicationDidFinishLaunching method 133, 138, 150
 applicationDidFinishLaunching: withOptions method 133, 138
 applicationDidReceiveMemoryWarning 193–194
 applications 238–239, 242

- demo, subclassing in 138–143
- iOS. *See* iOS (Apple Operating System) applications
- Rental Manager
 - developing 29–32
 - making data driven 91–94
 - running on iOS device 292
 - sample object 69–73
 - states of, inspecting with breakpoints 23–24
 - with bugs, creating 277

 Applications subfolder 8

- application-specific query 220
- application-specific variant 320
- Approve button 290
- arguments 62
- arithmetic operators 296
- ARM (Advanced RISC Machine) code 322
- array factory method 81
- arrays 48–50, 61, 75–82
 - adding items to 80–82
 - constructing 75–76
 - elements of
 - accessing 76–77
 - searching for 77–78
 - initializing 49
 - iterating through 79–80
 - fast enumeration 80
 - NSEnumerator class 79–80
 - vs. simple types 50
- arrayWithCapacity: factory method 81
- arrayWithContentsOfURL: message 75, 83
- arrayWithObject: factory method 75
- ASCII character chart 37
- aSimpleDynamicMethod selector 174
- Assembly.LoadFrom 324
- Assign attribute 111
- assignment operators 299–300
- asterisk, in variable name 208
- asynchronous task performance 265–275
 - GCD fundamentals 266–267
 - image loading 273–275
 - image search 271–272
 - introduction to GCD 266
 - RealEstateViewer application 267–271
- atIndexPath method 81
- atIndexPath method 242, 249
- attributes Core Data 233–234
- Attributes Inspector option 15
- Attributes Inspector pane 15
- Author element 159
- authors, parsing with NSXMLParser
 - delegate 159–162
- autoboxing 88
- Automatic Device Provisioning check box 291
- autorelease messages 177, 180, 185–187, 192–193
- autorelease object 190
- autorelease pools 184–190
 - adding objects to 185
 - creating new 185–187
 - limitations of 187–190
 - releasing objects in 187

B

- beginAnimations: context: method 147
- BEGINSWITH operator 222
- BETWEEN operator 221
- bFlag variable identifiers 294
- binding, dynamic 166
- bitwise operators 298
- block literal 259–260, 263–264
- __block storage type 261–263, 274
- blocks, syntax of 257–265
 - block-based APIs in iOS frameworks 264–265
 - blocks and memory management 262–264
 - closures 260–261
- Book elements 159
- BOOL data type 39–40
- Boolean truths 39–40
- boxing 88–91
 - nil vs. NULL value vs. NSNull class 90–91
 - NSNumber class 89
 - NSNumber class 90
- boxView 147–148
- Brautaset, Stig 268
- break keyword 304
- break statement 309–311
- breakpoints, inspecting application states with 23–24
- buffer overrun 38
- bugs, creating application with 277
- Build option 21
- Build Phases tab 21
- Build Settings tab 292
- building process 21
- bundle identifier 9
- buttonPressed 172

C

- C and C++ 315
- C libraries 5
- C++ libraries 5
- CABasicAnimation objects 14, 21
- callers, protocol method 147–148
- callHeads method 13–14, 16, 18
- callTails method 13–14, 16, 18
- CAMediaTimingFunction object 21
- Camel case 295
- cApples variable identifiers 294
- caret
 - in block literal 259
 - in syntax of blocks 258
- Cascading Style Sheets. *See* CSS
- categories
 - considerations when using 138
 - extending classes without subclassing 136
- C-based APIs 5, 315
- C-based application 300
- C-based libraries 313
- cellForRowAtIndexPath: method 30–32, 53–54

- Certificate Revocation List 289
- certificates 289–290
- Certificates page 290
- Certificates tab 289
- char * data type 38–39
- char data type 37–38
- characterAtIndex: message 67
- characters, extracting from strings 67–68
- chFoo variable identifiers 294
- chGender 294
- CIL (Common Intermediate Language)-based assemblies 323
- cityMapping dictionary 93
- cityMappings dictionary 197
- CityMappings.plist file 91, 93
- class clusters 57, 134
- class methods, vs. instance methods 104–105
- class_addMethod 173–174
- classes 97–143
 - adding new methods to at runtime 173–174
 - categories
 - considerations when using 138
 - extending classes without subclassing 136
 - clusters 134–136
 - multiple public 135–136
 - reasons for using 134–135
 - custom, adding new class to project 98–99
 - declared properties 109–115
 - @property syntax 109–112
 - dot syntax 113–115
 - synthesizing property getter and setter methods 112–113
 - declaring interface of 99–106
 - header file for CTREntalProperty class 105–106
 - ivars 100–101
 - method declarations 101–105
 - definitions of 57
 - in Rental Manager application 120–123
 - instance variables
 - accessing existing 129–131
 - adding new 127–129
 - making conform to protocol 148–150
 - objects 115–120
 - combining allocation and initialization 118
 - creating and initializing 115–116
 - destroying 119–120
 - init method 116–118
 - overriding methods 131–134
 - providing implementation for 106–109
 - accessing instance variables 106–107
 - method file for CTREntalProperty class 108–109
 - methods 106
 - sending messages to self 107–108
 - sending messages to 63
 - subclassing
 - in demo application 138–143
 - overview of 124–127
- Classes folder 268
- classnameWithxxxx naming convention 283
- clickedButtonAtIndex: method 227
- CLLocation.h header file 51
- closures, blocks as 260–261
- clusters 134–136
 - multiple public 135–136
 - reasons for using 134–135
- cmd parameter 167, 173
- Cocoa API (Application Programming Interface) 204–206
- Cocoa exceptions 210
- Cocoa frameworks 4–5
- Cocoa Touch support library 3–4
- code
 - pure, Person entities in 237–239
 - silent flaws in 116
- Code Editor window 286
- code reuse 57
- Code Sense 295
- Code Signing section 292
- code-signing certificates 323
- Coin Toss game
 - compiling 21–22
 - developing with Xcode tool 7–15
 - creating projects with 9–12
 - description of 8
 - launching 8
 - writing source code 12–15
 - hooking up user interface 15–20
 - test run 21–27
 - controlling debugger 25–27
 - inspecting application state with breakpoints 23–24
 - running CoinToss game in iPhone simulator 24–25
 - selecting destination 22–23
- coinLandedOnHeads variable 14, 27
- CoinTossViewController class 12, 196
 - reviewing connections made to and from 20
 - visually forming connection between button control and 17–19
- CoinTossViewController header file 12, 16
- CoinTossViewController.m file 13
- CoinTossViewController.xib file 15
- collating values 216–217
- collection-based data structures 215
- collections 74–94
 - arrays 75–82
 - adding items to 80–82
 - constructing 75–76

- collections (*continued*)
 - elements of 76–78
 - iterating through 79–80
 - boxing 88–91
 - nil vs. NULL value vs. NSNull class 90–91
 - NSNumber class 89
 - NSValue class 90
 - dictionaries 82–88
 - accessing entries in 84–85
 - adding key/value pairs 85–86
 - constructing 82–84
 - enumerating all keys and values 86–88
 - filtering 220
 - making Rental Manager application data driven 91–94
 - colon character 62–63
 - commitAnimations method 148
 - commitEditingStyle:forRowAtIndexPath method 246
 - Common Intermediate Language assemblies. *See* CIL
 - communicating, with objects 62–66
 - comparison operators 39, 296–298
 - compile-time errors 296
 - compile-time type 164
 - compiling, Coin Toss game 21–22
 - compliance, with KVC 213–217
 - accessing properties via 214
 - key paths 215
 - values 215–217
 - computing power, adjusting expectations for 5–7
 - hardware specifications 6
 - unreliable internet connections 7
 - conditional operator 302–303
 - conditional statements 300–305
 - conditional operator 302–303
 - if-else statement 301–302
 - switch statement 303–305
 - conditions, predicate
 - complex 221–222
 - expressing 220–221
 - configureCell: atIndexPath method 242, 249
 - Console section 278
 - constraints, for integral numbers in real world 33–34
 - CONTAINS operator 222
 - containsObject: message 78
 - context-sensitive editor pane 11
 - Continue button, Xcode debugger window 25, 27
 - continue statement 309, 311
 - controls, adding to view 15–16
 - copy attribute 111, 120
 - copy message 262
 - copyWithZone message 190
 - Core Data attributes 233–234
 - Core Data entities 229–230, 232–234, 236
 - Core Data framework
 - error domains 204
 - history of 229–231
 - objects 231–232
 - managed 231–232
 - persistent store coordinator 231
 - resources 232–234
 - attributes 233–234
 - entities 232–233
 - relationships 234
 - Core Data relationships 234
 - Core Data stack 231–232, 237
 - Core Data templates, Xcode 234–235
 - Core Data-based projects 231
 - CoreData.framework 231–232
 - Corona SDK product 320–321
 - count message 75–76, 79
 - Counter class 295
 - CountOfPeople variable 295
 - createBlock function 261
 - CreateMessageForPerson method 184–185
 - createSampleData method 238–239
 - cross-platform play 325
 - CSS (Cascading Style Sheets) 315
 - CSS3 316–317
 - C-style array 75
 - CTFixedLease class 141–143
 - CTFixedTermLease 140
 - CTFontCreateWithName() method 176
 - CTLease class 139–140
 - CTPeriodicLease object 140–142
 - CTPeriodicLease subclass 140–141
 - CTPerson objects 215
 - CTRentalProperty class 164–166, 173–174, 217–219
 - header files for 105–106
 - method files for 108–109
 - curly braces, in block literal 259
 - custom subclasses 231
- D**
-
- Data Model Inspector 235–236
 - Data Modeling tool 235
 - data models 235, 251–253
 - data source 145
 - data types 28–54
 - additional 34–35
 - basic 32–40
 - Boolean truths 39–40
 - integral numbers 32–35
 - char 37–38
 - custom 44–52
 - arrays 48–50

- data types (*continued*)
 - descriptive names for 50–52
 - enumerations 44–46
 - structures 46–48
- displaying and converting values 40–44
 - NSLog function and format specifiers 40–42
 - type casts and type conversions 43–44
- id 58–59
- Rental Manager application 29–32
 - completing 52–54
 - developing 29–32
- database table 234
- data-driven applications, Rental Manager 91–94
- DBController classes 230
- Deactivate Breakpoints option 25
- dealloc message 180, 284
- dealloc methods 14, 93, 119–120, 234, 242
- DEBUG preprocessor symbol 280
- Debugger Console window 31
- debuggers, controlling 25–27
- debugging 276–287
 - controlling memory leaks with Instruments application 281–283
 - creating application with bugs 277
 - detecting zombie objects 284–287
 - NSLog function 278–281
- DebugSample application 278, 285
- DebugSample_Prefix.pch file 279
- declarations, method 101–105
- declared properties 109–115
 - @property syntax 109–112
 - method naming 111
 - setter method semantics 111
 - thread safety 111–112
 - writeability 111
 - dot syntax 113–115
 - synthesizing property getter and setter methods 112–113
- Declared Properties feature 109
- decreaseRentalByPercent: withMinimum method 103–104
- default priority queue 272
- delegate parameter 146, 158
- delegation 166
- dequeueReusableCellWithIdentifier method 152–153
- dereferencing operation 60
- description method 132–134, 136
- desktop-sized screens 318
- destinations, selecting 22–23
- details variable 71
- deterministic behavior 180
- Developer folder 8–9
- Developer/Applications folder 8
- developers, Apple 288
- development tools 4–5
- development, preparing iOS devices for 289–292
- Devices section 291
- Devices tab 291
- devices, iOS
 - preparing for development 289, 292
 - running applications on 292
- dictionaries 82–88
 - accessing entries in 84–85
 - adding key/value pairs 85–86
 - constructing 82–84
 - enumerating all keys and values 86–88
- dictionaryWithContentsOfURL: message 83
- dictionaryWithObjects: forKey method 83
- dictionaryWithObjectsAndKeys: message 83
- dictionaryWithValuesForKey: method 214
- didFinishLaunchingWithOptions: method 238–239, 242
- didPresentActionSheet: method 157
- didReceiveMemoryWarning message 193–196
- didReceiveMemoryWarning method 194–197
- didSelectRowAtIndexPath: method 249, 285–286
- dispatch_async function 267, 272
- dispatch_get_global_queue function 267
- dispatch_queue_create function 267
- distinctUnionOfObjects aggregate function 217
- Distribution tab 292
- do loop 308, 311
- do while statement 280
- Document Type Definition. *See* DTD
- doesNotRecognizeSelector 169
- DoomEd coding 314
- dot syntax 113–115
- double data type 35–36
- do-while loop 307
- Download button 292
- DTD (Document Type Definition) 158
- DTrace 8
- dumpDataToConsole method 239, 242
- dynamic binding 166
- dynamic typing 163–176
 - dynamic binding 166
 - messaging 166–171
 - handling unknown selectors 169–170
 - methods, selectors, and implementations 167–168
 - sending message to nil 170–171
 - runtime type information 171–174
 - adding new methods to class at runtime 173–174
 - determining if message will respond 171
 - practical uses of 174–176
 - sending messages generated at runtime 171–172
 - static typing vs. 164–165

E

-
- e character 35
 - editButtonItem 246
 - Editor Style button 235
 - element variable 160
 - elements, of arrays
 - accessing 76–77
 - searching for 77–78
 - else keyword 301
 - emulator term 22
 - ENDSWITH operator 222
 - entities, Core Data 232–233
 - enum keyword 45–46
 - enumerateObjectsUsingBlock block-based API 264
 - enumerating
 - all keys and values in dictionaries 86–88
 - fast enumeration 80
 - enumerations 44–46
 - enumerators 79
 - error codes 204
 - error domains 204
 - error handling, of application data 253–256
 - error object 204
 - error variable 204
 - errors, handling 203–210
 - in Cocoa API 204–206
 - in RentalManagerAPI project 209–210
 - NSError objects 206–210
 - silent flaws in code 116
 - escape sequences 37–38
 - evaluateWithObject: message 220
 - exceptions 210–211
 - catching 211
 - throwing 210
 - explicit type conversion 43
 - exponential notation 35
 - expressions
 - operators in 296–300
 - arithmetic operators 296
 - assignment operators 299–300
 - bitwise operators 298
 - comparison operators 296–298
 - precedence of 300
 - predicate
 - parameterizing and templating 223–224
 - using key paths in 222
 - Extended Details pane (Cmd-E) 282
-
- Facebook web application 316
 - factory methods 67
 - failure, of objects to initialize 116
 - FALSEPREDICATE operator 221
 - fancyAddress 170
 - fBusy variable identifiers 294
 - FIFO (first in, first out) 266
 - File menu 9, 29
 - files, for CTRentalProperty class
 - header files 105–106
 - method files 108–109
 - filteredProperties 227
 - filtering, with predicates 219–224
 - complex conditions 221–222
 - evaluating predicate 219–220
 - expressing predicate condition 220–221
 - filtering collection 220
 - predicate expressions 222–224
 - filterUsingPredicate: message 220
 - Finder window 8, 70
 - first in, first out. *See* FIFO
 - FirstViewController class 196
 - Flash toolchain 323
 - Flashbang Studios 324
 - float data type 35–36
 - floating-point numbers 35–36
 - foo property 111
 - FooBar identifiers 294
 - for loop 49, 77, 308, 311
 - for statement 309
 - forKeys, message 83, 85–86, 214, 218
 - format specifiers 40–42
 - forObject, message 86
 - forRowAtIndexPath, method 246
 - forUndefinedKey, message 217–218
 - forwardingTargetForSelector 169
 - forwardInvocation 169–170
 - Foundation framework
 - error codes 205
 - error domains 204
 - Foundation Kit framework 4
 - Foundation.framework 232
 - foundCharacters, method 161
 - Frameworks section, Xcode main window 12, 21
-
- G**
 - garbage collection 119
 - GCC (GNU compiler collection) 8
 - GCD (Grand Central Dispatch) 265–275
 - GCD fundamentals 266–267
 - image loading 273–275
 - image search 271–272
 - introduction to GCD 266
 - RealEstateViewer application 267–271
 - GDB (GNU debugger) 8
 - gender instance variables 130
 - generateMessage method 286
 - getKey method 213

getRentalPrice 103
 getter methods 112–113, 131
 getValue message 90
 global queue, running block on 266
 GNU compiler collection. *See* GCC
 GNU debugger. *See* GDB
 Google, Image Search API 268
 Grand Central Dispatch. *See* GCD
 graphical plist file editor 91–92

H

handleComplaint method 108
 hard-to-detect-and-diagnose errors 321
 hardware, specifications for iOS applications 6
 header (*.h) file 13
 header files, for CTRentalProperty class 105–106
 Heads button 17
 heightForRowAtIndexPath: method 154
 Hide button 25
 HIG (Human Interface Guidelines) 17
 history of Objective-C 312–314
 adoption by Apple Inc. 314
 origins 313
 popularization via NeXT Inc. 313–314
 house object 174
 HTML5 316–317
 Human Interface Guidelines. *See* HIG
 Hungarian notation 294–295

I

IBAction keyword 20
 IBOutlet keyword 20
 iCount 294
 id data type 58–59, 64, 165
 IEEE 754 Standard format 36
 if statements 27, 77, 117, 152, 286, 310
 if-else statements 301–303
 iFoo variable identifiers 294
 ILP32 programming model 34
 Image Search API, Google 268
 images
 asynchronous loading of 273–275
 asynchronous searches for 271–272
 ImageTableViewController class 267–268
 ImageTableViewController.h header file 269
 ImageTableViewController.m file 269
 immutable array 80
 immutable objects 68
 implementations 167–168
 accessing instance variables 106–107
 method file for CTRentalProperty class 108–109
 methods 106
 sending messages to self 107–108

in keyword 80
 Include Unit Tests check box 10
 increaseRentalByPercent: withMaximum
 method 106, 108, 110
 indexes, in arrays 48
 indexOfObject: message 78
 indexPath.row property 54
 inheritance 57–58
 init methods 116–118, 127–128, 247
 init-based object creation process 67
 initialization, combining allocation with 118
 initWithContentsOfFile: method 93
 initWithPerson method 247
 initWithString: message 66
 initWithURL: method 116
 initWithXYZ message 192
 initWithXYZ: method 117
 initWithZone 191
 inManagedObjectContext: method 239
 insertObject: atIndex method 81
 instance methods 104–105, 131
 instance variables. *See* ivars
 Instruments application, controlling memory leaks
 with 281–283
 int data type 28, 32, 35
 int keyword 294
 integral numbers 32–35
 additional data types 34–35
 char data type 37–38
 constraints in real world 33–34
 floating-point numbers 35–36
 strings 38–39
 Interface Builder, Xcode 4 4
 interfaces, of classes 99–106
 header file for CTRentalProperty class
 105–106
 ivars 100–101
 method declarations 101–105
 internet, unreliable connections 7
 intValue message 89
 iOS (Apple Operating System) applications 3–27
 adjusting computing power and resource
 expectations 5–7
 hardware specifications 6
 unreliable internet connections 7
 block-based APIs in 264–265
 Coin Toss game
 compiling 21–22
 developing with Xcode tool 7–15
 hooking up user interface 15–20
 test run 21–27
 development tools 4–5
 SDK 288–292
 installing 288–289
 preparing device for development 289–292

- iOS Developer account 9
- iOS developer program 289
- iOS Developer Program license 323
- iOS project templates 10
- iOS Provisioning Portal 291
- iOS Simulator 22–23
- iOS-based templates 10
- iOS-powered device 322
- iPad, hardware specifications 6
- iPhone
 - capabilities 5
 - hardware specifications 6
 - screen 6
- iPhone application, running CoinToss game
 - in 24–25
- iPhone Developer program 323
- iPhone Packer 322
- iPhone Safari web browser 316
- iPhone SDK 315–320
 - HTML5, CSS3, and other standards 316–317
 - integration with 317–319
 - PhoneGap 319–320
- isDone attribute 235
- isEqual: message 69
- isKey instance variable 214
- isKey method 213
- isKey variable 213
- isKindOfClass 171
- isMultitaskingSupported property 175
- iterating, through arrays 79–80
 - fast enumeration 80
 - NSEnumerator class 79–80
- ivars (instance variables) 98, 100–101
 - accessing 106–107, 129–131
 - adding new 127–129

J

- java.lang.Object 100
- JavaScript source code 317
- JavaScript-accessible wrappers 319
- JIT (just-in-time) compilation 323
- JSON format 268
- JSON framework 268

K

- kCAMediaTimingFunctionEaseInEaseOut
 - object 21
- key Mono technologies 323
- key paths, using in predicate expressions 215, 222
- _key variable 213–214
- key/value pairs, adding to dictionaries 85–86

- keyEnumerator message 87
- keys
 - enumerating, in dictionaries 86–88
 - in dictionaries 82
 - unknown 217–218
- keys array 83, 85
- Key-Value Coding Programming Guide 215
- Key-Value Coding. *See* KVC
- Kitten case 305
- KVC (Key-Value Coding) 212–227
 - and NSPredicate class
 - filtering and matching with predicates 219–224
 - sample application 224–227
 - compliance with 213–217
 - accessing properties via KVC 214
 - key paths 215
 - values 215–217
 - handling special cases 217–219
 - nil values 218–219
 - unknown keys 217–218

L

- label control 19
- languages, procedural-based 56
- lastObject message 77
- length message 67, 171
- length property 215
- Let Me Specify Key Pair Information check
 - box 290
- Libraries section 21
- Library section 291
- Library window 15
- lightweight migrations 252
- LIKE operator 222
- Link Binary option 21
- LLVM (Low-Level Virtual Machine) 8
- localizedDescription method 205
- localizedFailureReason method 205
- localizedRecoveryOptions method 205
- localizedRecoverySuggestion method 205
- log 295
- LogAlways 280–281
- LogDebug macro 280
- logical operators 39, 298
- long qualifier 34
- looping statements 305–311
 - controlling 309–311
 - break statement 310
 - continue statement 311
 - do-while statement 307–308
 - for statement 308–309
 - while statement 306–307
- Low-Level Virtual Machine. *See* LLVM

low-memory warnings, responding to 193–200
 overriding `didReceiveMemoryWarning`
 method 194–197
 UIApplicationDelegate protocol 193–194
 UIApplicationDidReceiveMemoryWarningNoti-
 fication notification 197–200
 LP64 programming model 34
 Lua 320–322

M

Mac App Store 289
 main queue 272–273
 main thread 271, 273
 mainBundle method 64
 makeBlock method 263
 malloc_destroy_zone method 192
 managed objects
 context 231
 models 232
 managedObjectContext method 235, 239
 managedObjectModel method 235
 master TableView class 240–243
 MATCHES operator 222
 matching, with predicates 219–224
 complex conditions 221–222
 evaluating predicate 219–220
 expressing predicate condition 220–221
 filtering collection 220
 predicate expressions 222–224
 memory 177–200
 autorelease pools 184–190
 adding objects to 185
 creating new 185–187
 limitations of 187–190
 releasing objects in 187
 controlling leaks with Instruments
 application 281–283
 object ownership 178–179, 192–193
 reference counting 179–184
 determining current retain count 182–184
 releasing object 180–182
 responding to low-memory warnings 193–200
 overriding `didReceiveMemoryWarning`
 method 194–197
 UIApplicationDelegate protocol 193–194
 UIApplicationDidReceiveMemoryWarning-
 Notification notification 197–200
 zones 190–192
 memory fragmentation 190
 memory leak 178
 memory management, blocks and 262–264
 memory maps 59
 memory zones 190
 message forwarding 166, 169

messages
 nonexistent 64–65
 sending
 to classes 63
 to nil 65–66
 to objects 62–63
 to self 107–108
 messaging 166–171
 determining if message will respond 171
 handling unknown selectors 169–170
 methods, selectors, and implementations
 167–168
 sending messages generated at runtime
 171–172
 to nil 170–171
 method callers protocol 147–148
 method declarations 101–105
 method files, for CTRentalProperty class 108–109
 method implementations 106
 method naming category 110
 method swizzling 174
 method_exchangeImplementations 174
 methods 167–168
 <UITABLEVIEWDATASOURCE> 153
 <UITableViewDataSource> 151–153
 <UITableViewDelegate> action 155–157
 <UITableViewDelegate> setter 154–155
 adding new to class at runtime 173–174
 class, vs. instance methods 104–105
 getter
 manual approach to 130–131
 synthesizing 112–113
 naming 111
 overriding 131–134
 setter
 manual approach to 130–131
 semantics 111
 synthesizing 112–113
 Microsoft .NET development platform 323
 Microsoft's Windows Mobile 317
 Minimal overhead 180
 <MKMapViewDelegate> protocol 153
 mobile devices, adapting Cocoa frameworks
 for 4–5
 model objects 249–251
 models
 data 235, 251–253
 managed object 232
 model-view-controller. *See* MVC
 Mono (.NET) 323
 MonoDroid 325
 MonoTouch 324
 msg object 182
 multiple public clusters 135–136
 mutable objects 68

mutableCopyWithZone 190
 MVC (model-view-controller) 229
 MyBlockTest class 263
 myObject object 172
 myProtocol project 146
 myProtocolAppDelegate.m file 149
 myView class 146–147, 150
 myView delegate 150
 myView object 150
 myView.h file 146

N

name attribute 235
 name object 198–199
 name variable 128
 namespaces 295–296
 naming, for variables 293–296
 Camel case 295
 Hungarian notation 294–295
 namespaces 295–296
 native ARM code 323
 Navigation-based Application template 10, 29–30
 nBar variable identifiers 294
 needsConfiguration 115
 .NET applications 323, 325
 .NET CLR runtime environment 323
 .NET-based languages 324
 new addTask method 251
 New App ID button 291
 New File dialog 91, 98
 New File menu option 98
 New Project dialog 9, 29
 New Project option 9, 29
 newMessage variable 286
 NeXT Inc., popularization of Objective-C by 313–314
 nextObject method 79
 NeXTStep GUI concepts 314
 NeXTStep operating system 313
 NeXTStep technologies 314
 .nib files 21
 nil
 sending messages to 65–66, 170–171
 vs. NULL value vs. NSNull class 90–91
 nil constant 62
 nil receiver 170
 nil values 218–219
 nonatomic attribute 112
 NONE operator 223
 nonexistent messages, sending 64–65
 non-NULL pointer 297
 notFoundMarker argument 85
 notFoundMarker message 84–85
 notification handler, using block as 265
 Novell's MonoDevelop IDE 323
 NS namespace 295
 NS prefix 295
 NSArray class 75, 80, 82, 165
 NSArray's filteredArrayUsingPredicate message 220
 NSAutoreleasepool 285
 NSAutoreleasePool class 185–189
 NSAutoreleasePool instances 186
 NSCocoaErrorDomain error domain 204
 NSCreateZone function 190–191
 NSData object 204
 NSDefaultMallocZone function 191
 NSDictionary class 82, 84, 87, 91–92, 208
 NSEnumerator class 79–80
 NSError objects 203–210, 253
 RentalManagerAPI project 206–208
 userInfo dictionary 205–206
 NSExcption class 210
 NSFastEnumeration protocol 80
 NSFetchedResultsController class 228, 241–242, 246, 253
 NSFetchRequest class 240, 242, 253
 NSInvalidArgumentException 223
 NSInvocation class 169–170
 NSLog function 40–42, 278–281
 NSLog message 174
 NSLog operation 131
 NSLog situation 281
 NSLog statement 182, 190
 NSLog-style format string 67
 NSMachErrorDomain error domain 204
 NSManagedObject 232, 247, 249–251
 NSManagedObjectContext 235, 243
 NSManagedObjectModel 235
 NSMutableArray class 76, 80–81, 135
 NSMutableDictionary class 82–85, 218
 NSMutableString element 159–161
 NSMutableString object 180–181
 NSNotFound value 78
 NSNotification object 199
 NSNotificationCenter 197–199
 NSNull class, vs. nil vs. NULL value 90–91
 [NSNull null] statement 91
 NSNumber class 89, 135, 230
 NSNumber wrapper 135
 NSObject class 124, 131–132, 139–141
 NSObject implements 148
 NSObject method 141
 NSObject object 58
 NSObject version 133
 NSObjects 232
 NSOSStatusErrorDomain error domain 204
 NSPersistentStoreCoordinator 235
 NSPOSIXErrorDomain error domain 204

NSPredicate class 213, 219–220, 222–223
 filtering and matching with predicates
 219–224
 sample application 224–227
 NSPredicate-based expressions 222–223
 NSPredicate-based filtering 224
 NSRecycleZone 192
 NSSelectorFromString 168, 172
 NSSet 219
 NSSortDescriptors 239
 NSSQLiteStoreType 232
 NSString class 64–66, 136–138
 NSString object 190, 214–215, 283
 NSString stringWithFormat 223
 NSStringFromSelector method 168
 NSUnderlyingError key 205
 NSValue class 90
 <NSXMLParser> protocol 158
 NSXMLParser class 158–162
 NSXMLParser delegate methods 158–160, 162
 NSZombie Detection 285
 NSZombies feature 284–285
 NULL character 38
 NULL constant 62
 NULL object 170
 NULL reference exception 65, 181, 298
 NULL value, vs. nil vs. NSNull class 90–91
 numberOfCharacters 171
 numberOfComplaints instance variable 108
 numberOfItems variable 77
 numberOfRowsInSection, method 30–31, 53–54,
 277
 numberOfSectionsInTableView 242
 numbers array 311
 numberWithInt 89
 numberWithRentalPropertyDetail method 89

O

objc_msgSend 166–167
 objc_object struct 58
 objcType, message 90
 Object Library option 15
 object message 198–199
 object ownership 178–179, 192–193
 objectAtIndex message 77–79, 84
 objectAtIndex method 123
 objectEnumerator message 79–80, 87
 objectForKey message 84–85, 87, 93
 Objective-C classes 232, 234
 Objective-C developers 276
 Objective-C objects 214, 229, 233
 Objective-C programming language 3–4
 Objective-C statement 219
 Objective-C syntax 62, 281

Objective-C++ 315
 Objective-C-based APIs 5
 Objective-C-based iOS application 319
 object-orientated programming model 4
 object-oriented programming. *See* OOP
 objects 55–73, 115–120
 combining allocation and initialization 118
 communicating with 62–66
 creating and initializing 115–116
 definitions of 56
 destroying 119–120
 id data type 58–59
 init method 116–118
 OOP concepts 56–58
 definitions 56–57
 inheritance and polymorphism 57–58
 vs. procedural-based languages 56
 pointers 59–61
 comparing values of 61
 following 60–61
 memory maps 59
 obtaining address of variable 59–60
 sample application 69, 73
 sending messages to 62–63
 strings 66–69
 comparing 69
 constructing 66–67
 extracting characters from 67–68
 modifying 68–69
 objectsForKeys 84–85
 objectsPassingTest block-based API 264
 observer argument 198
 Offline Application Cache 317
 one-to-many relationship 234
 one-to-one relationship 234
 Online Certificate Status Protocol 289
 OOP (object-oriented programming) 56–58
 definitions
 of classes 57
 of objects 56
 inheritance and polymorphism 57–58
 vs. procedural-based languages 56
 opacity property 14
 OpenGL 315
 OpenGL ES Application template 10
 OPENSTEP 4.2 Desktop 314
 operators, in expressions 296–300
 arithmetic operators 296
 assignment operators 299–300
 bitwise operators 298
 comparison operators 296–298
 precedence of 300
 optional methods,
 <UITABLEVIEWDATASource> 153
 order object 234

Organizer window 291
 overriding methods 131–134
 ownership, of objects 178–179, 192–193

P

parallel thread 271
 parameterizing, predicate expressions 223–224
 parentheses in syntax of blocks 258, 260
 parse method 160
 Parser subclass 159
 Parser_ProjectAppDelegate.m 161
 parsing author with NSXMLParser delegate 159–162
 Pause button 25
 people, adding and deleting 243–246
 PeopleViewController class 240, 246
 performance of application data 253
 performSelector 171–172
 periodicLease method 140
 persistent store coordinators 231
 persistentStoreCoordinator method 235, 252
 Person class 125–126, 128, 130, 132, 250–251
 Person entities, in pure code
 creating 237–239
 fetching 239–240
 Person objects 230
 person.firstName 249
 Person.h file 130, 251
 PersonDetailViewController 243, 254
 pet variable 304
 PhoneGap 319–320
 plist (Property List) schema 76
 PocketTasks.xcdatamodel file 252
 PocketTasks application 234–251
 adding and deleting people 243–246
 data model 235
 defining relationships 236
 managing tasks 246–250
 master TableView class 240–243
 model objects 249–251
 Person entities in pure code
 creating 237–239
 fetching 239–240
 Xcode Core Data template 234–235
 PocketTasks.xcdatamodel 234–235
 PocketTasksAppDelegate.h file 237
 PocketTasksAppDelegate.m 234, 242
 pointers 59–61
 comparing values of 61
 following 60–61
 memory maps 59
 obtaining address of variable 59–60
 polymorphism 57–58

post-decrement operations 300
 precedence of operators 300
 predicate conditions 213, 219
 predicates, filtering and matching with 219–224
 complex conditions 221–222
 evaluating predicate 219–220
 expressing predicate condition 220–221
 filtering collection 220
 predicate expressions 222–224
 predicateWithFormat 223
 primitive data types 32
 procedural-based languages, OOP vs. 56
 Product menu
 Build option 21
 Deactivate Breakpoints option 25
 Run option 24
 Project Navigator pane 11, 21
 projects
 adding new class to 98–99
 creating with Xcode tool 9–12
 properties
 accessing via KVC 214
 declared 109–115
 @property syntax 109–112
 dot syntax 113–115
 synthesizing property getter and setter methods 112–113
 properties array 53–54, 75
 Property List file 92
 property.address 114
 PropertyType enumeration 52, 105
 propertyType property 222
 propertyType value 100
 protocol method callers 147–148
 protocols 144–162
 definition of 145–146
 implementing 146–150
 making class conform to protocol 148–150
 protocol method callers 147–148
 important 150–162
 <UIActionSheetDelegate> protocol 157–158
 <UITableViewDataSource> protocol 150–153
 <UITableViewDelegate> protocol 153–157
 NSXMLParser class 158–162
 provisioning
 manually 291–292
 using Xcode 290–291
 Provisioning Portal website 292
 public clusters, multiple 135–136
 publishAd:error: method 208–209
 pure code, Person entities in
 creating 237–239
 fetching 239–240

Q

QuakeEd coding 314
 QuartzCore framework 21

R

raise method 210
 rangeOfString method 72
 Raptor Copter 324
 readonly attribute 111
 readonly property 111
 readonly attribute 111
 RealEstateViewer application 267–271
 RealEstateViewerAppDelegate.h header file 268
 RealEstateViewerAppDelegate.m file 268
 Record Reference Counts check boxes 285
 reference counting 179–184

- determining current retain count 182–184
- releasing object 180–182

 relationships

- Core Data 234
- defining 236

 Release configuration 292
 release message 262
 releasing objects 180–182, 187
 removeAllObjects method 86
 removeObjectAtIndex message 82
 removeObjectForKey message 85
 removeObjectForKey message 85
 removeObserver 199
 Rental Manager application

- classes in 120–123
- completing 52–54
- developing 29–32
- making data driven 91–94

 RentalManagerAPI project 206–210
 RentalManagerAPI.h header file 207
 RentalManagerAPI.m file 207
 RentalManagerAppDelegate class 193
 rentalPerWeek 113
 rentalPrice property 102–103, 107, 109, 218–219
 rentalPrice value 100, 219
 rentalPrice variable 114
 RentalProperty data type 52
 RentalProperty structure 97, 100, 120
 rentalPropertyOfType 118
 removeObjectAtIndex 82
 replaceOccurrencesOfString 137
 required methods,

- <UITableViewDataSource> 151–153

 reserved words 294
 resolveInstanceMethod 173–174
 Resource section, New File dialog 92

resources

- adjusting expectations for 5–7
- hardware specifications 6
- unreliable internet connections 7

 of Core Data framework 232–234

- attributes 233–234
- entities 232–233
- relationships 234

 respondsToSelector method 148, 171, 175
 result property 19
 retain attribute 111, 120
 retain count 179, 182–184
 retain message 262, 283, 286
 retainCount message 182–183
 return statement 259, 286
 reverseObjectEnumerator message 80
 RootViewController class 92, 194–195, 277, 284
 RootViewController.h file 52–53, 92
 RootViewController.m file 30, 53, 71, 93
 Ruby 320–322
 Run button 292
 run loop 186
 Run option 24
 runMemoryTest method 263
 runtime type

- information about 171–174
 - adding new methods to class at runtime 173–174
 - determining if message will respond 171
 - practical uses of 174–176
 - sending messages generated at runtime 171–172
- making assumptions about 164–165

S

Safari browser-based environment 319
 safety, of threads 111–112
 saveAndDismiss method 243
 scientific notation 35
 <script/> element 319
 SDK (software development kit) 229, 315
 second tab 196
 sectionNameKeyPath 242
 security, of threads 111–112
 SEL data type 168
 selector argument 198–199
 selector name 198–199
 selectors 167–170
 self parameter 107
 [self setStatus:nil] 196
 self.rentalPrice 218
 semantics, of setter methods 111
 serial dispatch queue, creating your own 267
 setAddress 109, 166, 168

- setFlag 164
- setNilValueForKey 219
- setObject 85–86
- setRentalPrice method 103, 106, 109, 114
- setSortDescriptors method 239
- setter methods
 - <UITableViewDelegate> 154–155
 - semantics 111
 - synthesizing 112–113
- Setter semantics category 110
- setValue
 - forKey 86, 214, 218
 - forObject 86
 - forUndefinedKey 217–218
- setValuesForKeysWithDictionary method 214
- short qualifier 34
- shortcuts
 - Alt-Cmd-4 15
 - Cmd-4 21
 - Cmd-B 21
 - Cmd-Option-P 27
 - Cmd-R 29, 31, 54
 - Cmd-Y 25
 - Control-Option-Cmd-3 15
 - Shift-Cmd-N 9
 - Shift-Cmd-Y 31
- signed qualifier 32–33, 37
- simple types, arrays vs. 50
- simulateCoinToss method 14, 24, 26
- simulator term 23
- simulators, running CoinToss game in 24–25
- Singleton design pattern 91
- software development kit. *See* SDK
- software runtime environment 5
- someString object 171
- source code
 - connecting controls to 17–20
 - writing with Xcode tool 12–15
- special catch-all case 303
- specialization 57
- Split View-based Application template 10
- SQL code 229
- SQL SELECT statement 240
- SQLite 229
- square brackets 62
- states, inspecting with breakpoints 23–24
- static typing, vs. dynamic typing 164–165
- status property 19
- status variable 196
- Step Into button 25
- Step Out button 25
- Step Over button 25–26
- store coordinators, persistent 231
- strcat function 38
- strcpy function 38
- stringByAppendingString message 69
- stringByDestroyingVowels method 137–138
- stringByReplacingOccurrencesOfString 63, 68
- strings 38–39, 66–69
 - comparing 69
 - constructing 66–67
 - extracting characters from 67–68
 - modifying 68–69
- stringWithFormat message 67, 285
- stringWithFormat method 104, 118
- stringWithObject 192
- stringWithString 192
- strlen function 38
- struct box data type 50–51
- struct keyword 47–48, 52
- structures 46–48
- Student classes 126, 128
- subclassing
 - in demo application 138–143
 - overview 124–127
- substringFromIndex method 72
- substringToIndex method 72
- substringWithRange message 67–68
- Supporting Files group 70
- switch statement 303–305
- syntax 61
- synthesizing getter methods 112–113
- system requirements, for installing iOS SDK
 - 288–289
- System.Reflection.Emit 323

T

- Tab Bar Application 292
- Tab Bar Application template 10
- tableView 123, 156–157, 242
 - cellForRowAtIndexPath 71, 93, 273, 277, 280, 282
 - commitEditingStyle 246
 - didSelectRowAtIndexPath 249, 285–286
 - numberOfRowsInSection 30–31, 53–54, 153
- tableView 277
- TableView class 240–243
- Tails button 18
- Target-Action design pattern 172
- Task class 250
- Task entity 235
- tasks, managing 246–250
- TasksViewController 246, 249
- Teacher classes 126, 129
- Teacher init method 129
- Teacher object 126
- templates, Xcode Core Data 234–235
- templating, predicate expressions 223–224
- Tenants property 215

test runs, Coin Toss game 21–27
 controlling debugger 25–27
 inspecting application state with
 breakpoints 23–24
 running in iPhone simulator 24–25
 selecting destination 22–23

third-party iOS applications 317

thisObject 166

threads, safety of 111–112

Titanium Mobile–based application 320

toolbar buttons 25

tools, development. *See* development tools

transform.rotation property 14

TRUEPREDICATE operator 221

type casts 43–44

type conversions 43–44

type definition 50

type information, runtime 171–174
 adding new methods to class at runtime
 173–174
 determining if message will respond 171
 practical uses of 174–176
 sending messages generated at runtime
 171–172

typedef keyword 50–52, 259

U

UI_USER_INTERFACE_IDIOM 175

<UIActionSheetDelegate> protocol 157–158

UIAlertView class 209, 254

UIApplicationDelegate class 193–194

UIApplicationDelegate protocol 149–150,
 193–194

UIApplicationDidReceiveMemoryWarning-
 Notification 193, 197–200

UIButton class 172, 197

UIDevice class 175

UIImageView class 155, 273

UIKit elements 147, 151

UIKit framework 4–5

UIKit.framework 232

UILabel controls 12–14, 16, 27, 196

UIPrintInteractionController class 175–176

UISlider class 172

UISlider control 223

UITableView class 151, 157, 197, 277, 281–282

UITableView control 30–32, 54

UITableViewCell class 151–153, 155–156, 283

UITableViewCellAccessoryCheckmark 156

UITableViewCellAccessoryDetailDisclosureButton
 156

UITableViewCellAccessoryDisclosureIndicator
 155

UITableViewCellAccessoryNone 155

UITableViewCellAccessoryType parameter 156

UITableViewCellAccessoryTypes 155

UITableViewCells 151, 153, 155

UITableViewCellStyleDefault 31, 151

UITableViewCellStyleSubtitle 54, 152

UITableViewCellStyleValue1 152

UITableViewCellStyleValue2 152

UITableViewController class 240, 246, 267

<UITableViewDataSource> protocol 150–153
 <UITableViewDataSource> optional
 methods 153
 <UITableViewDataSource> required
 method 151–153

UITableViewDataSource protocol 145, 150–151

UITableViewDataSource section 154

<UITableViewDelegate> protocol 153, 155–157
 <UITableViewDelegate> action methods
 155–157
 <UITableViewDelegate> setter methods 154–155

UITableViewStyleGrouped 151

UITableViewStylePlain 151

UITextField IBOutlets 243

UIView objects 172

UIView subclass 145–149, 155

UIViewController class 186, 193–197, 243, 246

UIViews class 194

UIWebView control 319

underscore (_) prefix 214

Unity3D engine 324

unknown keys 217–218

unsigned qualifier 32, 34

Use for Development button 291

user interface, for Coin Toss game 15–20

userCalledHeads parameter 14, 26

userInfo dictionary 205–206, 208, 210

Utility Application template 10

V

validation, of application data 253–256

valueForKey method 213–214

valueForKeyPath message 215

valueForUndefinedKey message 217–218

values
 aggregating and collating 216–217
 displaying and converting 40–44
 NSLog function and format specifiers 40–42
 type casts and type conversions 43–44
 enumerating, in dictionaries 86–88
 key/value pairs, adding to dictionaries 85–86
 nil 218–219
 of pointers, comparing 61
 returning multiple 215

values array 83, 85

valueWithBytes argument 90

vardic method 75
 variables
 instance
 accessing 106–107
 accessing existing 129–131
 adding new 127–129
 naming conditions for 293–296
 Camel case 295
 Hungarian notation 294–295
 namespaces 295–296
 obtaining address of 59–60
 View menu 15
 view, adding controls to 15–16
 View-based Application template 9–10, 12
 viewDidAppear message 197
 viewDidLoad method 93, 173, 196–197, 249, 286
 viewDidLoadUnloaded method 195–197
 Visual Studio 323
 VowelDestroyer interface 137

W

W3C Geolocation API 318
 warnings, low-memory 193–200
 weak-linking support 176
 web-based applications 315–316, 319
 while loop 79, 306–308, 310–311
 white rectangle 15
 willPresentActionSheet method 157
 willSelect 156
 Window-based Application template 10
 Windows-based PC 323
 wireframe box 15
 wireless connectivity 7
 withMinimum method 104

withObject message 82
 withString
 method 63, 68
 options 137
 world-class mobile browser 317
 writeability 111
 Writeability category 110
 WWDC intermediate certificate 290

X

Xcode 4 4
 Xcode Core Data template 234–235
 Xcode Data Modeling tool 235
 Xcode debugger window 25, 27
 Xcode Organizer 291
 Xcode Organizer window 278
 Xcode tool, developing Coin Toss game with 7–15
 creating projects 9–12
 description of Xcode tool 8
 launching 8
 writing source code 12–15
 Xcode toolset
 and iOS SDK
 downloading 289
 installing 289
 provisioning using 290–291
 Xcode window 235
 .xib files 21

Z

zombie objects, detecting 284–287
 zones, memory 190–192

Objective-C Fundamentals

Fairbairn • Fahrenkrug • Ruffenach



Objective-C Fundamentals guides you gradually from your first line of Objective-C code through the process of building native apps for the iPhone. Starting with chapter one, you'll dive into iPhone development by building a simple game that you can run immediately. You'll use tools like Xcode 4 and the debugger that will help you become a more efficient programmer. By working through numerous easy-to-follow examples, you'll learn practical techniques and patterns you can use to create solid and stable apps. And you'll find out how to avoid the most common pitfalls.

What's Inside

- Objective-C from the ground up
- Developing with Xcode 4
- Examples work unmodified on iPhone

No iOS or mobile experience is required to benefit from this book but familiarity with programming in general is helpful.

Christopher Fairbairn, Johannes Fahrenkrug, and Collin Ruffenach are professional mobile app developers, each with over a decade of experience using different systems including iOS, Palm, Windows Mobile, and Java.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/ObjectiveCFundamentals

“A handy and complete reference.”

—Glenn Stokol
Oracle Corporation.

“The essential iOS programming how-to guide.”

—Dave Bales, Whitescape

“A tour-de-force of Objective-C...I want to grok this stuff!”

—Dave Mateer, Mateer IT

“A superb introduction to essential iPhone application development tools.”

—Carl Douglas, NZX

“Become a hot commodity on the market... with this book.”

—Ted Neward, Principal,
Neward & Associates

