COCOA® PROGRAMMING FOR MAC® OS X

FOURTH EDITION

AARON HILLEGASS ADAM PREBLE

Cocoa® Programming for Mac® OS X

Fourth Edition

This page intentionally left blank

COCOA® PROGRAMMING FOR MAC® OS X

FOURTH EDITION

Aaron Hillegass
Adam Preble

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

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

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Hillegass, Aaron.
Cocoa programming for Mac OS X / Aaron Hillegass, Adam Preble.—4th ed.
p. cm.
Includes index.
ISBN 978-0-321-77408-8 (pbk. : alk. paper)
1. Cocoa (Application development environment) 2. Operating systems
(Computers) 3. Mac OS. 4. Macintosh (Computer)—Programming. I. Preble, Adam. II. Title.

QA76.76.O63H57145 2012
005.26'8—dc23

2011034459

Copyright © 2012 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-77408-8
ISBN-10: 0-321-77408-6

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, November 2011

For Aaron's sons, Walden and Otto

and

For Adam's daughter, Aimee

This page intentionally left blank

Contents

Preface . . . . **xix**

Acknowledgments . . . . **xxi**

Chapter 1 Cocoa: What Is It? **1**

A Little History 1

Tools 3

Language 4

Objects, Classes, Methods, and Messages 5

Frameworks 6

How to Read This Book 7

Typographical Conventions 7

Common Mistakes 8

How to Learn 8

Chapter 2 Let's Get Started **11**

In Xcode 11

Create a New Project 12

The main Function 15

In Interface Builder 15

The Utility Area 16

The Blank Window 16

Lay Out the Interface 17

The Dock 20

Create a Class 20

Create an Instance 22

Make Connections 22

A Look at Objective-C 25

Types and Constants in Objective-C 26

Look at the Header File 26
Edit the Implementation File 27
Build and Run 28
awakeFromNib 30
Documentation 31
What Have You Done? 31
Chronology of an Application 32

Chapter 3 Objective-C 35
Creating and Using Instances 35
Using Existing Classes 37
Sending Messages to nil 41
NSObject, NSArray, NSMutableArray, and NSString 43
“Inherits from” versus “Uses” or “Knows About” 48
Creating Your Own Classes 48
Creating the LotteryEntry Class 49
Changing main.m 51
Implementing a description Method 52
Writing Initializers 55
Initializers with Arguments 56
The Debugger 58
What Have You Done? 63
Meet the Static Analyzer 63
For the More Curious: How Does Messaging Work? 65
Challenge 66

Chapter 4 Memory Management 67
Living with Manual Reference Counting 69
Leak-Free Lottery 70
dealloc 72
Autoreleasing Objects 73
The Retain-Count Rules 76
Accessor Methods 77
Living with ARC 80
Strong References 81
Weak References 81
ARC Odds and Ends 82

Chapter 5 Target/Action 83
Some Commonly Used Subclasses of NSControl 85
NSButton 85
NSSlider 86
NSTextField 87
Start the SpeakLine Example 89
Lay Out the XIB File 90
Making Connections in Interface Builder 91
Implementing the SpeakLineAppDelegate Class 94
For the More Curious: Setting the Target Programmatically 96
Challenge 96
Debugging Hints 98

Chapter 6 Helper Objects 99
Delegates 100
The NSTableView and Its dataSource 104
SpeakLineAppDelegate Interface File 105
Lay Out the User Interface 107
Make Connections 109
Edit SpeakLineAppDelegate.m 110
Common Errors in Implementing a Delegate 112
Many Objects Have Delegates 112
For the More Curious: How Delegates Work 113
Challenge: Make a Delegate 114
Challenge: Make a Data Source 114

Chapter 7 Key-Value Coding and Key-Value Observing 117
Key-Value Coding 117
Bindings 119
Key-Value Observing 120
Making Keys Observable 121
Properties 124
Attributes of a Property 125
For the More Curious: Key Paths 126
For the More Curious: Key-Value Observing 127

Chapter 8 NSArrayController 129
Starting the RaiseMan Application 130
RMDocument.xib 134

Key-Value Coding and nil 139
Add Sorting 140
For the More Curious: Sorting without NSArrayController 141
Challenge 1 142
Challenge 2 142

Chapter 9 NSUndoManager 145
NSInvocation 145
How the NSUndoManager Works 146
Adding Undo to RaiseMan 148
Key-Value Coding and To-Many Relationships 148
Key-Value Observing 152
Undo for Edits 153
Begin Editing on Insert 156
For the More Curious: Windows and the Undo Manager 158

Chapter 10 Archiving 159
NSCoder and NSCoding 160
Encoding 160
Decoding 162
The Document Architecture 163
Info.plist and NSDocumentController 163
NSDocument 164
NSWindowController 167
Saving and NSKeyedArchiver 167
Loading and NSKeyedUnarchiver 168
Setting the Extension and Icon for the File Type 170
For the More Curious: Preventing Infinite Loops 172
For the More Curious: Creating a Protocol 173
For the More Curious: Automatic Document Saving 174
For the More Curious: Document-Based Applications without Undo . . . 175
Universal Type Identifiers 175

Chapter 11 Basic Core Data 177
NSManagedObjectModel 177
Interface 179
View-Based Table Views 180
Connections and Bindings 183
How Core Data Works 189

For the More Curious: View-Based versus Cell-Based Table Views 191
Challenge 191

Chapter 12 NIB Files and NSWindowController 193
NSPanel 193
Adding a Panel to the Application 194
Setting Up the Menu Item 197
AppController.m 198
Preferences.xib 198
PreferenceController.m 203
For the More Curious: NSBundle 204
Challenge 206

Chapter 13 User Defaults 207
NSDictionary and NSMutableDictionary 208
NSDictionary 209
NSMutableDictionary 209
NSUserDefaults 210
Precedence of Types of Defaults 211
Setting Defaults 212
The Identifier for the Application 212
Create Keys for the Names of the Defaults 212
Register Defaults 213
Letting the User Edit the Defaults 213
Using the Defaults 215
Suppressing the Creation of Untitled Documents 215
Setting the Background Color on the Table View 216
For the More Curious: NSUserDefaultsController 217
For the More Curious: Reading and Writing Defaults from the Command Line 217
Challenge 219

Chapter 14 Using Notifications 221
What Notifications Are and Are Not 221
What Notifications Are Not 222
NSNotification 222
NSNotificationCenter 222
Posting a Notification 224
Registering as an Observer 225

Handling the Notification When It Arrives 226
The userInfo Dictionary 226
For the More Curious: Delegates and Notifications 227
Challenge 228

Chapter 15 Using Alert Panels 229
Make the User Confirm the Deletion 230
Challenge 232

Chapter 16 Localization 233
Localizing a NIB File 234
String Tables 236
Creating String Tables 237
Using the String Table 238
For the More Curious: ibtool 239
For the More Curious: Explicit Ordering of Tokens in Format Strings 240

Chapter 17 Custom Views 241
The View Hierarchy 241
Get a View to Draw Itself 243
Create an Instance of a View Subclass 243
Size Inspector 244
drawRect 246
Drawing with NSBezierPath 248
NSScrollView 250
Creating Views Programmatically 252
For the More Curious: Cells 253
For the More Curious: isFlipped 255
Challenge 255

Chapter 18 Images and Mouse Events 257
NSResponder 257
NSEvent 257
Getting Mouse Events 259
Using NSOpenPanel 259
Change the XIB File 260
Edit the Code 263

Composite an Image onto Your View 264
The View's Coordinate System 266
Autoscrolling 268
For the More Curious: NSImage 269
Challenge 270

Chapter 19 Keyboard Events 271
NSResponder 273
NSEvent 273
Create a New Project with a Custom View 274
Lay Out the Interface 274
Make Connections 276
Write the Code 278
For the More Curious: Rollovers 282
The Fuzzy Blue Box 284

Chapter 20 Drawing Text with Attributes 285
NSFont 285
NSAttributedString 286
Drawing Strings and Attributed Strings 289
Making Letters Appear 289
Getting Your View to Generate PDF Data 291
For the More Curious: NSFontManager 293
Challenge 1 293
Challenge 2 294

Chapter 21 Pasteboards and Nil-Targeted Actions 295
NSPasteboard 296
Add Cut, Copy, and Paste to BigLetterView 298
Nil-Targeted Actions 300
Looking at the XIB File 301
For the More Curious: Which Object Sends the Action Message? 303
For the More Curious: UTIs and the Pasteboard 303
Custom UTIs 303
For the More Curious: Lazy Copying 304
Challenge 1 305
Challenge 2 305

Chapter 22 Categories 307
Add a Method to NSString 307
For the More Curious: Declaring Private Methods 309

Chapter 23 Drag-and-Drop 311
Make BigLetterView a Drag Source 312
Starting a Drag 312
After the Drop 314
Make BigLetterView a Drag Destination 315
registerForDraggedTypes: 316
Add Highlighting 316
Implement the Dragging Destination Methods 317
Add a Second BigLetterView 318
For the More Curious: Operation Mask 319

Chapter 24 NSTimer 321
Lay Out the Interface 323
Make Connections 325
Add Code to TutorController 326
For the More Curious: NSRunLoop 328
Challenge 328

Chapter 25 Sheets 329
Adding a Sheet 330
Add Outlets and Actions 331
Lay Out the Interface 331
Add Code 334
For the More Curious: contextInfo 335
For the More Curious: Modal Windows 336

Chapter 26 Creating NSFormatters 339
A Basic Formatter 341
Create ColorFormatter.h 341
Edit the XIB File 342
NSColorList 344
Searching Strings for Substrings 344
Implement the Basic Formatter Methods 345
The Delegate of the NSControl Class 347

Checking Partial Strings 348
Formatters That Return Attributed Strings 350
For the More Curious: NSValueTransformer 351

Chapter 27 Printing 353
Dealing with Pagination 353
For the More Curious: Are you Drawing to the Screen? 358
Challenge 358

Chapter 28 Web Services 359
RanchForecast Project 360
NSURLConnection 361
Add XML Parsing to ScheduleFetcher 363
Lay Out the Interface 366
Write Controller Code 368
Opening URLs 368
Challenge: Add a WebView 369

Chapter 29 Blocks 371
Block Syntax 373
Memory and Objects within Blocks 375
Availability of Blocks 376
RanchForecast: Going Asynchronous 376
Receiving the Asynchronous Response 377
Challenge: Design a Delegate 381

Chapter 30 Developing for iOS 383
Porting RanchForecast to iOS 383
ScheduleFetcher 386
RootViewController 386
Add a Navigation Controller 388
ScheduleViewController 391
UITableViewController 392
Pushing View Controllers 393
Challenge 395

Chapter 31 View Swapping 397
Get Started 398
Create the ManagedViewController Class 399
Create ViewControllers and their XIB files 400

Add View Swapping to MyDocument 401
Resizing the Window 403

Chapter 32 Core Data Relationships 407
Edit the Model 407
Create Custom NSManagedObject Classes 409
Employee 409
Department 410
Lay Out the Interface 411
EmployeeView.xib 413
Events and nextResponder 414

Chapter 33 Core Animation 417
Scattered 417
Implicit Animation and Actions 423
More on CALayer 425
Challenge 1 425
Challenge 2 425

Chapter 34 Concurrency 427
Multithreading 427
A Deep Chasm Opens Before You 428
Simple Cocoa Background Threads 429
Improving Scattered: Time Profiling in Instruments 431
Introducing Instruments 431
NSOperationQueue 435
Multithreaded Scattered 435
Thread Synchronization 437
For the More Curious: Faster Scattered 438
Challenge 439

Chapter 35 Cocoa and OpenGL 441
A Simple Cocoa/OpenGL Application 442
Lay Out the Interface 442
Write Code 446

Chapter 36 NSTask 451
ZIPspector 451
Asynchronous Reads 456
iPing 456
Challenge: .tar and .tgz files 460

Chapter 37 Distributing Your App 461
Build Configurations 461
Preprocessor Macros and Using Build Configurations to Change Behavior 462
Creating a Release Build 464
Application Sandboxing 466
Entitlements 467
Mediated File Access and Powerbox 468
The Mac App Store 468

Chapter 38 The End 471

Index 473

This page intentionally left blank

Preface

If you are developing applications for the Mac, or are hoping to do so, this book is just the resource you need. Does it cover everything you will ever want to know about programming for the Mac? Of course not. But it does cover probably 80% of what you need to know. You can find the remaining 20%—the 20% that is unique to you—in Apple's online documentation.

This book, then, acts as a foundation. It covers the Objective-C language and the major design patterns of Cocoa. It will also get you started with the two most commonly used developer tools: Xcode and Instruments. After reading this book, you will be able to understand and utilize Apple's online documentation.

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

This fourth edition includes technologies introduced in Mac OS X 10.6 and 10.7. These include Xcode 4, ARC, blocks, view-based table views, and the Mac App Store. We have also devoted one chapter to the basics of iOS development.

This book is written for programmers who already know some C programming and something about objects. If you don't know C or objects, you should first read *Objective-C Programming: The Big Nerd Ranch Guide*. You are not expected to have any experience with Mac programming. This hands-on book assumes that you have access to Mac OS X and the developer tools. Xcode 4.2, Apple's IDE, is available for free. If you are a member of the paid Mac or iOS Developer Programs, Xcode can also be downloaded from the Apple Developer Connection Web site (http://developer.apple.com/). Enrollment in these programs enables you to submit your applications to the Mac and iOS App Stores, respectively.

We have tried to make this book as useful for you as possible, if not indispensable. That said, we'd love to hear from you at cocoabook@bignerdranch.com if you have any suggestions for improving it.

—Aaron Hillegass and Adam Preble

This page intentionally left blank

Acknowledgments

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

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

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

Thank you to all the instructors at the Ranch, who made great additions and caught many of our most egregious errors.

A final shout out to the people at Addison-Wesley, who took our manuscript and made it into a book. They put the book on trucks and convinced bookstores to put it on the shelves. Without their help, it would still be just a stack of paper.

This page intentionally left blank

Chapter 3
OBJECTIVE-C

Once upon a time, a man named Brad Cox decided that it was time for the world to move toward a more modular programming style. C was a popular and powerful language. Smalltalk was an elegant untyped object-oriented language. Starting with C, Brad Cox added Smalltalk-like classes and message-sending mechanisms. He called the result *Objective-C*. Objective-C is a very simple extension of the C language. In fact, it was originally just a C preprocessor and a library.

Objective-C is not a proprietary language. Rather, it is an open standard that has been included in the Free Software Foundation's GNU C compiler (gcc) for many years. More recently, Apple has become heavily involved in the clang/LLVM (Low Level Virtual Machine) open source compiler projects, which are much faster and more versatile than gcc. In Xcode projects, LLVM is the default compiler.

Cocoa was developed using Objective-C, and most Cocoa programming is done in Objective-C. Teaching C and basic object-oriented concepts could consume an entire book. This chapter assumes that you already know a little C and something about objects and introduces you to the basics of Objective-C. If you fit the profile, you will find learning Objective-C to be easy. If you do not, our own *Objective-C Programming: The Big Nerd Ranch Guide* or Apple's *The Objective-C Language* offer more gentle introductions.

Creating and Using Instances

Chapter 1 mentioned that classes are used to create objects, that the objects have methods, and that you can send messages to the objects to trigger these methods. In this section, you will learn how to create an object and send messages to it.

As an example, we will use the class **NSMutableArray**. You can create a new instance of **NSMutableArray** by sending the message **alloc** to the **NSMutableArray** class like this:

```
[NSMutableArray alloc];
```

This method returns a pointer to the space that was allocated for the object. You could hold onto that pointer in a variable like this:

```
NSMutableArray *foo;
foo = [NSMutableArray alloc];
```

While working with Objective-C, it is important to remember that `foo` is just a pointer. In this case, it points to an object.

Before using the object that `foo` points to, you would need to make sure that it is fully initialized. The **`init`** method will handle this task, so you might write code like this:

```
NSMutableArray *foo;
foo = [NSMutableArray alloc];
[foo init];
```

Take a long look at the last line; it sends the message **`init`** to the object that `foo` points to. We would say, "`foo` is the receiver of the message **`init`**." Note that a message send consists of a receiver (the object `foo` points to) and a message (**`init`**) wrapped in brackets. You can also send messages to *classes*, as demonstrated by sending the message **`alloc`** to the class **`NSMutableArray`**.

The method **`init`** returns the newly initialized object. As a consequence, you will always nest the message sends like this:

```
NSMutableArray *foo;
foo = [[NSMutableArray alloc] init];
```

What about destroying the object when we no longer need it? We will talk about this in the next chapter.

Some methods take arguments. If a method takes an argument, the method name (called a *selector*) will end with a colon. For example, to add objects to the end of the array, you use the **`addObject:`** method (assume that `bar` is a pointer to another object):

```
[foo addObject:bar];
```

If you have multiple arguments, the selector will have multiple parts. For example, to add an object at a particular index, you could use the following:

```
[foo insertObject:bar atIndex:5];
```

Note that **`insertObject:atIndex:`** is one selector, not two. It will trigger one method with two arguments. This outcome seems strange to most C and Java programmers but should be familiar to Smalltalk programmers. The syntax also makes your code easier to read. For example, it is not uncommon to see a C++ method call like this:

```
if (x.intersectsArc(35.0, 19.0, 23.0, 90.0, 120.0))
```

It is much easier to guess the meaning of the following code:

```
if ([x intersectsArcWithRadius:35.0
                  centeredAtX:19.0
                            Y:23.0
                    fromAngle:90.0
                      toAngle:120.0])
```

If it seems odd right now, just use it for a while. Most programmers grow to appreciate the Objective-C messaging syntax.

You are now at a point where you can read simple Objective-C code, so it is time to write a program that will create an instance of **`NSMutableArray`** and fill it with ten instances of **`NSNumber`**.

Using Existing Classes

If it isn't running, start Xcode. Close any projects that you were working on. Under the File menu, choose New -> New Project…. When the panel pops up, choose to create a Command Line Tool (Figure 3.1).

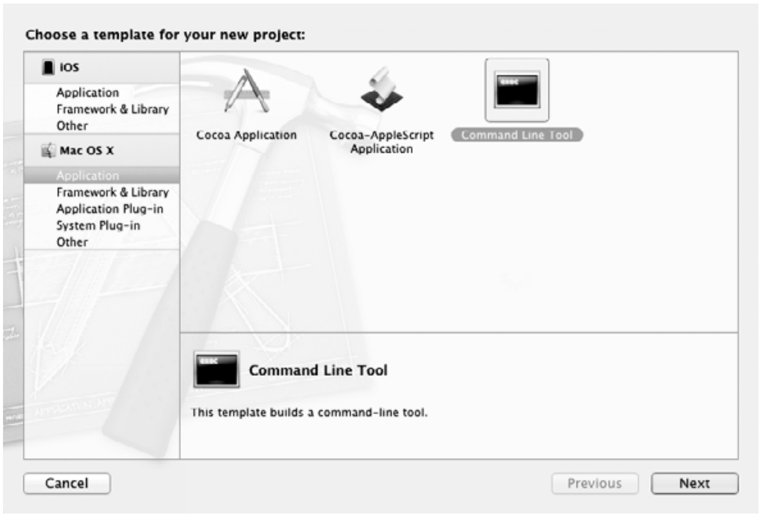
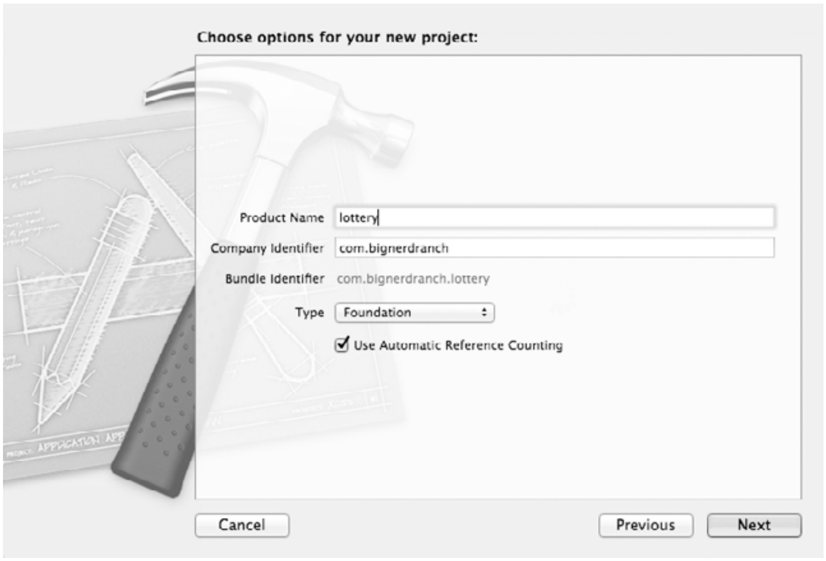


Figure 3.1 Choose Project Type

A *command-line tool* has no graphical user interface and typically runs on the command line or in the background as a daemon. Unlike in an application project, you will always alter the **main** function of a command-line tool.

Name the project lottery (Figure 3.2). Unlike the names of applications, most tool names are lowercase. Set the Type to Foundation.


Choose options for your new project:

Product Name: lottery
Company Identifier: com.bignerdranch
Bundle Identifier: com.bignerdranch.lottery
Type: Foundation
✓ Use Automatic Reference Counting

Figure 3.2 Name Project

When the new project appears, select `main.m` in the lottery group. Edit `main.m` to look like this:

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {

        NSMutableArray *array;
        array = [[NSMutableArray alloc] init];
        int i;
        for (i = 0; i < 10; i++) {
            NSNumber *newNumber =
                       [[NSNumber alloc] initWithInt:(i * 3)];
            [array addObject:newNumber];
        }
        for ( i = 0; i < 10; i++) {
            NSNumber *numberToPrint = [array objectAtIndex:i];
            NSLog(@"The number at index %d is %@",  i, numberToPrint);
        }

    }
    return 0;
}
```

Here is the play-by-play for the code:

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

You are including the headers for all the classes in the Foundation framework. The headers are precompiled, so this approach is not as computationally intensive as it sounds.

```
int main (int argc, const char *argv[])
```

The **main** function is declared just as it would be in any Unix C program.

```
@autoreleasepool {
```

This code defines an autorelease pool for the code enclosed by the braces. We will discuss the importance of autorelease pools in the next chapter.

```
NSMutableArray *array;
```

One variable is declared here: `array` is a pointer to an instance of **NSMutableArray**. Note that no array exists yet. You have simply declared a pointer that will refer to the array once it is created.

```
array = [[NSMutableArray alloc] init];
```

Here, you are creating the instance of **NSMutableArray** and making the `array` variable point to it.

```
for (i = 0; i < 10; i++) {
    NSNumber *newNumber = [[NSNumber alloc] initWithInt:(i*3)];
    [array addObject:newNumber];
}
```

Inside the `for` loop, you have created a local variable called `newNumber` and set it to point to a new instance of **NSNumber**. Then you have added that object to the array.

The array does not make copies of the **NSNumber** objects. Instead, it simply keeps a list of pointers to the **NSNumber** objects. Objective-C programmers make very few copies of objects, because it is seldom necessary.

```
for ( i = 0; i < 10; i++) {
    NSNumber *numberToPrint = [array objectAtIndex:i];
    NSLog(@"The number at index %d is %@", i, numberToPrint);
}
```

Here, you are printing the contents of the array to the console. **NSLog** is a function much like the C function **printf()**; it takes a format string and a comma-separated list of variables to be substituted into the format string. When displaying the string, **NSLog** prefixes the generated string with the name of the application and a time stamp.

In **printf**, for example, you would use %x to display an integer in hexadecimal form. With **NSLog**, we have all the tokens from **printf** and the token %@ to display an object. The object gets sent the message **description**, and the string it returns replaces %@ in the string. We will discuss the **description** method in detail soon.

All the tokens recognized by **NSLog()** are listed in Table 3.1.

Table 3.1 Possible Tokens in Objective-C Format Strings

Symbol	Displays
%@	id
%d, %D, %i	long
%u, %U	unsigned long
%hi	short
%hu	unsigned short
%qi	long long
%qu	unsigned long long
%x, %X	unsigned long printed as hexadecimal
%o, %O	unsigned long printed as octal
%f, %e, %E, %g, %G	double
%c	unsigned char as ASCII character
%C	unichar as Unicode character
%s	char * (a null-terminated C string of ASCII characters)
%S	unichar * (a null-terminated C string of Unicode characters)
%p	void * (an address printed in hexadecimal with a leading 0x)
%%	a % character

Note: If the @ symbol before the quotes in @"The number at index %d is %@" looks a little strange, remember that Objective-C is the C language with a couple of extensions. One of the extensions is that strings are instances of the class **NSString**. In C, strings are just pointers to a buffer of characters that ends in the null character. Both C strings and instances of **NSString** can be used in the same file. To differentiate between constant C strings and constant **NSString**s, you must put @ before the opening quote of a constant **NSString**.

```
// C string
char *foo;
// NSString
NSString *bar;
foo = "this is a C string";
bar = @"this is an NSString";
```

You will use mostly **NSString** in Cocoa programming. Wherever a string is needed, the classes in the frameworks expect an **NSString**. However, if you already have a bunch of C functions that expect C strings, you will find yourself using `char *` frequently.

You can convert between C strings and **NSString**s:

```
const char *foo = "Blah blah";
NSString *bar;
// Create an NSString from a C string
bar = [NSString stringWithUTF8String:foo];

// Create a C string from an NSString
foo = [bar UTF8String];
```

Because **NSString** can hold Unicode strings, you will need to deal with the multibyte characters correctly in your C strings, and this can be quite difficult and time consuming. (Besides the multibyte problem, you will have to wrestle with the fact that some languages read from right to left.) Whenever possible, you should use **NSString** instead of C strings.

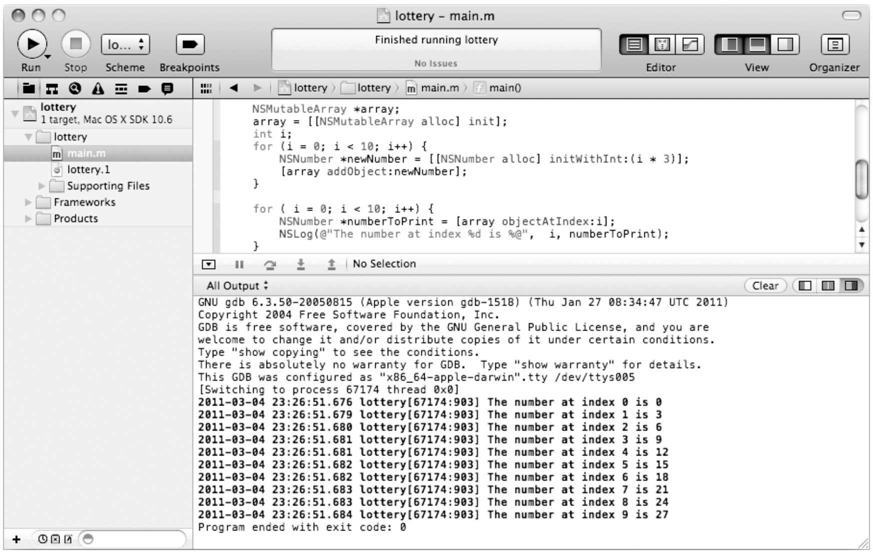
Our **main()** function ends by returning 0, indiciating that no error occurred:

```
    return 0;
}
```

Run the completed command-line tool (Figure 3.3). (If your console doesn't appear, use the View -> Show Debug Area menu item and ensure that the console, the right half, is enabled.)

Sending Messages to nil

In most object-oriented languages, your program will crash if you send a message to `null`. In applications written in those languages, you will see many



```
NSMutableArray *array;
array = [[NSMutableArray alloc] init];
int i;
for (i = 0; i < 10; i++) {
    NSNumber *newNumber = [[NSNumber alloc] initWithInt:(i * 3)];
    [array addObject:newNumber];
}

for ( i = 0; i < 10; i++) {
    NSNumber *numberToPrint = [array objectAtIndex:i];
    NSLog(@"The number at index %d is %@",  i, numberToPrint);
}
```

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1518) (Thu Jan 27 08:34:47 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".tty /dev/ttys005
[Switching to process 67174 thread 0x0]
2011-03-04 23:26:51.676 lottery[67174:903] The number at index 0 is 0
2011-03-04 23:26:51.679 lottery[67174:903] The number at index 1 is 3
2011-03-04 23:26:51.680 lottery[67174:903] The number at index 2 is 6
2011-03-04 23:26:51.681 lottery[67174:903] The number at index 3 is 9
2011-03-04 23:26:51.681 lottery[67174:903] The number at index 4 is 12
2011-03-04 23:26:51.682 lottery[67174:903] The number at index 5 is 15
2011-03-04 23:26:51.682 lottery[67174:903] The number at index 6 is 18
2011-03-04 23:26:51.683 lottery[67174:903] The number at index 7 is 21
2011-03-04 23:26:51.683 lottery[67174:903] The number at index 8 is 24
2011-03-04 23:26:51.684 lottery[67174:903] The number at index 9 is 27
Program ended with exit code: 0
```

Figure 3.3 Completed Execution

checks for `null` before sending a message. In Java, for example, you frequently see the following:

```
if (foo != null) {
    foo.doThatThingYouDo();
}
```

In Objective-C, it is okay to send a message to `nil`. The message is simply discarded, which eliminates the need for these sorts of checks. For example, this code will build and run without an error:

```
id foo;
foo = nil;
int bar = [foo count];
```

This approach is different from how most languages work, but you will get used to it.

You may find yourself asking over and over, "Argg! Why isn't this method getting called?" Chances are that the pointer you are using, convinced that it is not `nil`, is in fact `nil`.

In the preceding example, what is `bar` set to? Zero. If `bar` were a pointer, it would be set to `nil` (`zero` for pointers). For other types, the value is less predictable.

NSObject, NSArray, NSMutableArray, and NSString

You have now used these standard Cocoa objects: **`NSObject`**, **`NSMutableArray`**, and **`NSString`**. (All classes that come with Cocoa have names with the `NS` prefix. Classes that you will create will *not* start with `NS`.) These classes are all part of the Foundation framework. Figure 3.4 shows an inheritance diagram for these classes.

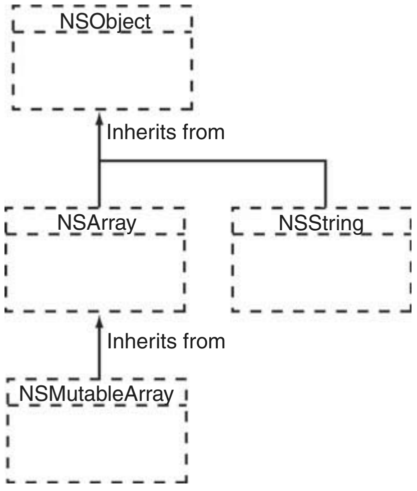


Figure 3.4 Inheritance Diagram

Let's go through a few of the commonly used methods on these classes. For a complete listing, you can access the online documentation in Xcode's `Help` menu.

NSObject

`NSObject` is the root of the entire Objective-C class hierarchy. Some commonly used methods on **`NSObject`** are described next.

```
- (id)init
```

Initializes the receiver after memory for it has been allocated. An **`init`** message is generally coupled with an **`alloc`** message in the same line of code:

```
TheClass *newObject = [[TheClass alloc] init];
```

```
- (NSString *)description
```

Returns an **NSString** that describes the receiver. The debugger's print object command ("po") invokes this method. A good **description** method will often make debugging easier. Also, if you use %@ in a format string, the object that should be substituted in is sent the message **description**. The value returned by the **description** method is put into the log string. For example, the line in your main function

```
NSLog(@"The number at index %d is %@", i, numberToPrint);
```

is equivalent to

```
NSLog(@"The number at index %d is %@", i,
                            [numberToPrint description]);
```

```
- (BOOL)isEqual:(id)anObject
```

Returns YES if the receiver and anObject are equal and NO otherwise. You might use it like this:

```
if ([myObject isEqual:anotherObject]) {
    NSLog(@"They are equal.");
}
```

But what does equal really mean? In **NSObject**, this method is defined to return YES if and only if the receiver and anObject are the same object—that is, if both are pointers to the same memory location.

Clearly, this is not always the "equal" that you would hope for, so this method is overridden by many classes to implement a more appropriate idea of equality. For example, **NSString** overrides the method to compare the characters in the receiver and anObject. If the two strings have the same characters in the same order, they are considered equal.

Thus, if x and y are **NSStrings**, there is a big difference between these two expressions:

```
x == y
```

and

```
[x isEqual:y]
```

The first expression compares the two pointers. The second expression compares the characters in the strings. Note, however, that if x and y are instances of a class that has not overridden **NSObject**'s **isEqual:** method, the two expressions are equivalent.

NSArray

An **`NSArray`** is a list of pointers to other objects. It is indexed by integers. Thus, if there are *n* objects in the array, the objects are indexed by the integers 0 through *n* – 1. You cannot put a `nil` in an **`NSArray`**. (This means that there are no "holes" in an **`NSArray`**, which may confuse some programmers who are used to Java's `Object[]`.) **`NSArray`** inherits from **`NSObject`**.

An **`NSArray`** is created with all the objects that will ever be in it. You can neither add nor remove objects from an instance of **`NSArray`**. We say that **`NSArray`** is *immutable*. (Its mutable subclass, **`NSMutableArray`**, will be discussed next.) Immutability is nice in some cases. Because it is immutable, a horde of objects can share one **`NSArray`** without worrying that one object in the horde might change it. **`NSString`** and **`NSNumber`** are also immutable. Instead of changing a string or number, you will simply create another one with the new value. (In the case of **`NSString`**, there is also the class **`NSMutableString`** that allows its instances to be altered.)

A single array can hold objects of many different classes. Arrays cannot, however, hold C primitive types, such as `int` or `float`.

Here are some commonly used methods implemented by **`NSArray`**:

```
- (unsigned)count
```

Returns the number of objects currently in the array.

```
- (id)objectAtIndex:(unsigned)i
```

Returns the object located at index `i`. If `i` is beyond the end of the array, you will get an error at runtime.

```
- (id)lastObject
```

Returns the object in the array with the highest index value. If the array is empty, `nil` is returned.

```
- (BOOL)containsObject:(id)anObject
```

Returns `YES` if `anObject` is present in the array. This method determines whether an object is present in the array by sending an **`isEqual:`** message to each of the array's objects and passing `anObject` as the parameter.

```
- (unsigned)indexOfObject:(id)anObject
```

Searches the receiver for `anObject` and returns the lowest index whose corresponding array value is equal to `anObject`. Objects are considered equal if **`isEqual:`** returns `YES`. If none of the objects in the array are equal to `anObject`, **`indexOfObject:`** returns `NSNotFound`.

NSMutableArray

`NSMutableArray` inherits from **`NSArray`** but extends it with the ability to add and remove objects. To create a mutable array from an immutable one, use **`NSArray`**'s **`mutableCopy`** method.

Here are some commonly used methods implemented by **`NSMutableArray`**:

```
- (void)addObject:(id)anObject
```

Inserts `anObject` at the end of the receiver. You are not allowed to add `nil` to the array.

```
- (void)addObjectsFromArray:(NSArray *)otherArray
```

Adds the objects contained in `otherArray` to the end of the receiver's array of objects.

```
- (void)insertObject:(id)anObject atIndex:(unsigned)index
```

Inserts `anObject` into the receiver at `index`, which cannot be greater than the number of elements in the array. If `index` is already occupied, the objects at `index` and beyond are shifted up one slot to make room. You will get an error if `anObject` is `nil` or if `index` is greater than the number of elements in the array.

```
- (void)removeAllObjects
```

Empties the receiver of all its elements.

```
- (void)removeObject:(id)anObject
```

Removes all occurrences of `anObject` in the array. Matches are determined on the basis of `anObject`'s response to the **`isEqual:`** message.

```
- (void)removeObjectAtIndex:(unsigned)index
```

Removes the object at `index` and moves all elements beyond `index` down one slot to fill the gap. You will get an error if `index` is beyond the end of the array.

As mentioned earlier, you cannot add `nil` to an array. Sometimes, you will want to put an object into an array to represent nothingness. The **`NSNull`** class exists for exactly this purpose. There is exactly one instance of **`NSNull`**, so if you want to put a placeholder for nothing into an array, use **`NSNull`** like this:

```
[myArray addObject:[NSNull null]];
```

NSString

An **`NSString`** is a buffer of Unicode characters. In Cocoa, all manipulations involving character strings are done with **`NSString`**. As a convenience, the Objective-C language also supports the `@"…"` construct to create a string object constant from a 7-bit ASCII encoding:

```
NSString *temp = @"this is a constant string";
```

`NSString` inherits from **`NSObject`**. Here are some commonly used methods implemented by **`NSString`**:

```
- (id)initWithFormat:(NSString *)format, ...
```

Works like **`sprintf`**. Here, `format` is a string containing tokens, such as `%d`. The additional arguments are substituted for the tokens:

```
int x = 5;
char *y = "abc";
id z = @"123";
NSString *aString = [[NSString alloc] initWithFormat:
            @"The int %d, the C String %s, and the NSString %@",
            x, y, z];
```

```
- (NSUInteger)length
```

Returns the number of characters in the receiver.

```
- (NSString *)stringByAppendingString:(NSString *)aString
```

Returns a string object made by appending `aString` to the receiver. The following code snippet, for example, would produce the string "Error: unable to read file."

```
NSString *errorTag = @"Error: ";
NSString *errorString = @"unable to read file.";
NSString *errorMessage;
errorMessage = [errorTag stringByAppendingString:errorString];
```

```
- (NSComparisonResult)compare:(NSString *)otherString
```

Compares the receiver and `otherString` and returns `NSOrderedAscending` if the receiver is alphabetically prior to `otherString`, `NSOrderedDescending` if `otherString` is comes before the receiver, or `NSOrderedSame` if the receiver and `otherString` are equal.

```
- (NSComparisonResult)caseInsensitiveCompare:(NSString *)
otherString
```

Like **`compare:`**, except the comparison ignores letter case.

"Inherits from" versus "Uses" or "Knows About"

Beginning Cocoa programmers are often eager to create subclasses of **`NSString`** and **`NSMutableArray`**. Don't. Stylish Objective-C programmers almost never do. Instead, they use **`NSString`** and **`NSMutableArray`** as parts of larger objects, a technique known as composition. For example, a **`BankAccount`** class *could* be a subclass of **`NSMutableArray`**. After all, isn't a bank account simply a collection of transactions? The beginner would follow this path. In contrast, the old hand would create a class **`BankAccount`** that inherited from **`NSObject`** and has an instance variable called `transactions` that would point to an **`NSMutableArray`**.

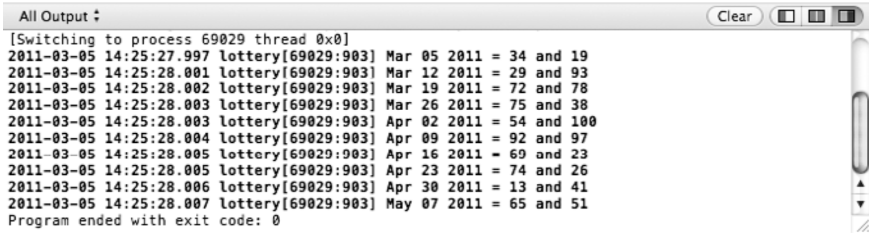
It is important to keep track of the difference between "uses" and "is a subclass of." The beginner would say, "**`BankAccount`** inherits from **`NSMutableArray`**." The old hand would say, "**`BankAccount`** uses **`NSMutableArray`**." In the common idioms of Objective-C, "uses" is much more common than "is a subclass of."

You will find it much easier to use a class than to subclass one. Subclassing involves more code and requires a deeper understanding of the superclass. By using composition instead of inheritance, Cocoa developers can take advantage of very powerful classes without really understanding how they work.

In a strongly typed language, such as C++, inheritance is crucial. In an untyped language, such as Objective-C, inheritance is just a hack that saves the developer some typing. There are only two inheritance diagrams in this entire book. All the other diagrams are object diagrams that indicate which objects know about which other objects. This is much more important information to a Cocoa programmer.

Creating Your Own Classes

Where I live, the state government has decided that the uneducated have entirely too much money: You can play the lottery every week. Let's imagine that a lottery entry has two numbers between 1 and 100, inclusive. You will write a program that will make up lottery entries for the next ten weeks. Each **`LotteryEntry`** object will have a date and two random integers (Figure 3.5).



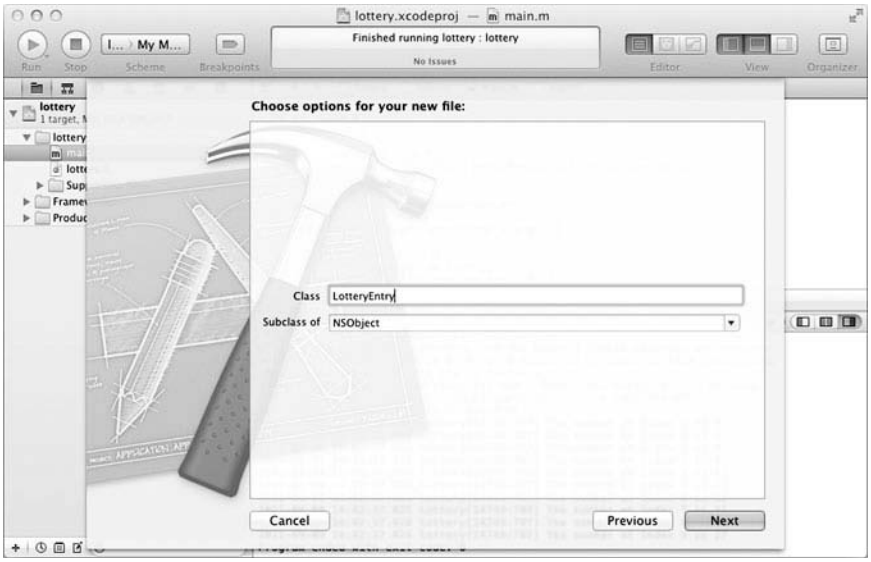
```
[Switching to process 69029 thread 0x0]
2011-03-05 14:25:27.997 lottery[69029:903] Mar 05 2011 = 34 and 19
2011-03-05 14:25:28.001 lottery[69029:903] Mar 12 2011 = 29 and 93
2011-03-05 14:25:28.002 lottery[69029:903] Mar 19 2011 = 72 and 78
2011-03-05 14:25:28.003 lottery[69029:903] Mar 26 2011 = 75 and 38
2011-03-05 14:25:28.003 lottery[69029:903] Apr 02 2011 = 54 and 100
2011-03-05 14:25:28.004 lottery[69029:903] Apr 09 2011 = 92 and 97
2011-03-05 14:25:28.005 lottery[69029:903] Apr 16 2011 = 69 and 23
2011-03-05 14:25:28.005 lottery[69029:903] Apr 23 2011 = 74 and 26
2011-03-05 14:25:28.006 lottery[69029:903] Apr 30 2011 = 13 and 41
2011-03-05 14:25:28.007 lottery[69029:903] May 07 2011 = 65 and 51
Program ended with exit code: 0
```

Figure 3.5 Completed Program

Besides learning how to create classes, you will build a tool that will certainly make you fabulously wealthy.

Creating the LotteryEntry Class

In Xcode, create a new file. Select Objective-C class as the type. Name the class **`LotteryEntry`**, and set it to be a subclass of **`NSObject`** (Figure 3.6).



Choose options for your new file:

Class LotteryEntry

Subclass of NSObject

Figure 3.6 New LotteryEntry Class

Note that you are also causing `LotteryEntry.h` to be created. Drag both files into the lottery group if they are not already there.

LotteryEntry.h

Edit the `LotteryEntry.h` file to look like this:

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

@interface LotteryEntry : NSObject {
    NSDate *entryDate;
    int firstNumber;
    int secondNumber;
}
```

```
- (void)prepareRandomNumbers;
- (void)setEntryDate:(NSDate *)date;
- (NSDate *)entryDate;
- (int)firstNumber;
- (int)secondNumber;
@end
```

You have created a header file for a new class called **`LotteryEntry`** that inherits from **`NSObject`**. It has three instance variables:

- `entryDate` is an **`NSDate`**.
- `firstNumber` and `secondNumber` are both `int`s.

You have declared five methods in the new class:

- **`prepareRandomNumbers`** will set `firstNumber` and `secondNumber` to random values between 1 and 100. It takes no arguments and returns nothing.
- **`entryDate`** and **`setEntryDate:`** will allow other objects to read and set the variable `entryDate`. The method **`entryDate`** will return the value stored in the `entryDate` variable. The method **`setEntryDate:`** will allow the value of the `entryDate` variable to be set. Methods that allow variables to be read and set are called *accessor methods*.
- You have also declared accessor methods for reading `firstNumber` and `secondNumber`. (You have not declared accessors for setting these variables; you are going to set them directly in **`prepareRandomNumbers`**.)

LotteryEntry.m

Edit `LotteryEntry.m` to look like this:

```
#import "LotteryEntry.h"

@implementation LotteryEntry

- (void)prepareRandomNumbers
{
    firstNumber = ((int)random() % 100) + 1;
    secondNumber = ((int)random() % 100) + 1;
}

- (void)setEntryDate:(NSDate *)date
{
    entryDate = date;
}
```

```
- (NSDate *)entryDate
{
    return entryDate;
}

- (int)firstNumber
{
    return firstNumber;
}

- (int)secondNumber
{
    return secondNumber;
}
@end
```

Here is the play-by-play for each method:

> **prepareRandomNumbers** uses the standard **random** function to generate a pseudorandom number. You use the mod operator (%) and add 1 to get the number in the range 1–100.
>
> **setEntryDate:** sets the pointer `entryDate` to a new value.
>
> **entryDate**, **firstNumber**, and **secondNumber** return the values of variables.

Changing main.m

Now let's look at `main.m`. Many of the lines have stayed the same, but several have changed. The most important change is that we are using **LotteryEntry** objects instead of **NSNumber** objects.

Here is the heavily commented code. (You don't have to type in the comments.)

```
#import <Foundation/Foundation.h>
#import "LotteryEntry.h"

int main (int argc, const char *argv[]) {

    @autoreleasepool {

        // Create the date object
        NSDate *now = [[NSDate alloc] init];
        NSCalendar *cal = [NSCalendar currentCalendar];
        NSDateComponents *weekComponents =
            [[NSDateComponents alloc] init];
        // Seed the random number generator
        srandom((unsigned)time(NULL));
        NSMutableArray *array;
        array = [[NSMutableArray alloc] init];

        int i;
        for (i = 0; i < 10; i++) {

            [weekComponents setWeek:i];

            // Create a date/time object that is 'i' weeks from now
            NSDate *iWeeksFromNow;
            iWeeksFromNow = [cal dateByAddingComponents:weekComponents
                                                 toDate:now
                                                options:0];

            // Create a new instance of LotteryEntry
            LotteryEntry *newEntry = [[LotteryEntry alloc] init];
            [newEntry prepareRandomNumbers];
            [newEntry setEntryDate:iWeeksFromNow];

            // Add the LotteryEntry object to the array
            [array addObject:newEntry];

        }

        for (LotteryEntry *entryToPrint in array) {
            // Display its contents
            NSLog(@"%@", entryToPrint);
        }
    }
    return 0;
}
```

Note the second loop. Here you are using Objective-C's mechanism for enumerating over the members of a collection.

This program will create an array of `LotteryEntry` objects, as shown in Figure 3.7.

Implementing a description Method

Build and run your application. You should see something like Figure 3.8.

Hmm. Not quite what we hoped for. After all, the program is supposed to reveal the dates and the numbers you should play on those dates, and you can't see either. (You are seeing the default **`description`** method as defined in **`NSObject`**.) Next, you will make the **`LotteryEntry`** objects display themselves in a more meaningful manner.

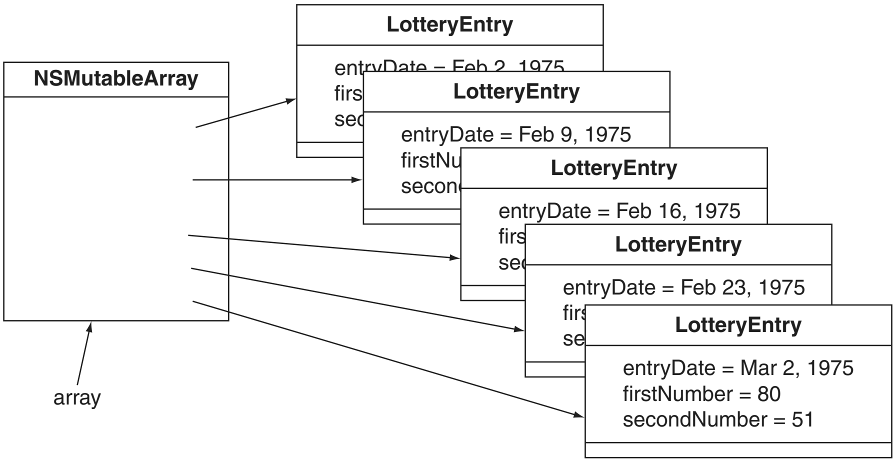
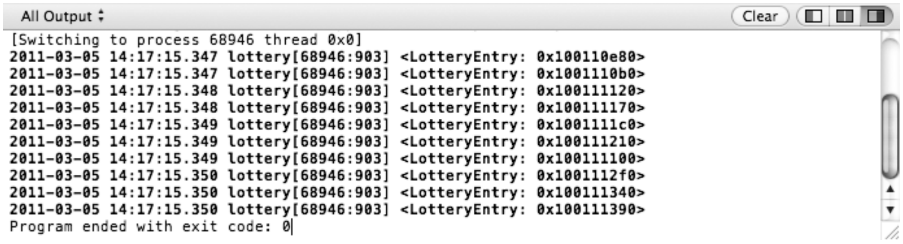


Figure 3.7 Object Diagram



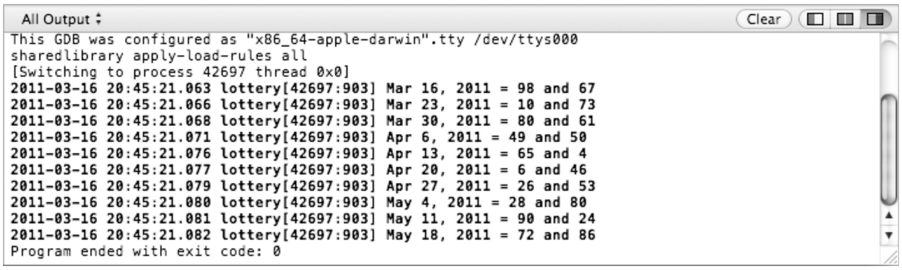
```
[Switching to process 68946 thread 0x0]
2011-03-05 14:17:15.347 lottery[68946:903] <LotteryEntry: 0x100110e80>
2011-03-05 14:17:15.347 lottery[68946:903] <LotteryEntry: 0x1001110b0>
2011-03-05 14:17:15.348 lottery[68946:903] <LotteryEntry: 0x100111120>
2011-03-05 14:17:15.348 lottery[68946:903] <LotteryEntry: 0x100111170>
2011-03-05 14:17:15.349 lottery[68946:903] <LotteryEntry: 0x1001111c0>
2011-03-05 14:17:15.349 lottery[68946:903] <LotteryEntry: 0x100111210>
2011-03-05 14:17:15.349 lottery[68946:903] <LotteryEntry: 0x100111100>
2011-03-05 14:17:15.350 lottery[68946:903] <LotteryEntry: 0x1001112f0>
2011-03-05 14:17:15.350 lottery[68946:903] <LotteryEntry: 0x100111340>
2011-03-05 14:17:15.350 lottery[68946:903] <LotteryEntry: 0x100111390>
Program ended with exit code: 0
```

Figure 3.8 Completed Execution

Add a **description** method to `LotteryEntry.m`:

```
- (NSString *)description
{
    NSDateFormatter *df = [[NSDateFormatter alloc] init];
    [df setTimeStyle:NSDateFormatterNoStyle];
    [df setDateStyle:NSDateFormatterMediumStyle];
    NSString *result;
    result = [[NSString alloc] initWithFormat:@"%@ = %d and %d",
              [df stringFromDate:entryDate],
              firstNumber, secondNumber];
    return result;
}
```

Build and run the application. Now you should see the dates and numbers:



```
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys000
sharedlibrary apply-load-rules all
[Switching to process 42697 thread 0x0]
2011-03-16 20:45:21.063 lottery[42697:903] Mar 16, 2011 = 98 and 67
2011-03-16 20:45:21.066 lottery[42697:903] Mar 23, 2011 = 10 and 73
2011-03-16 20:45:21.068 lottery[42697:903] Mar 30, 2011 = 80 and 61
2011-03-16 20:45:21.071 lottery[42697:903] Apr 6, 2011 = 49 and 50
2011-03-16 20:45:21.076 lottery[42697:903] Apr 13, 2011 = 65 and 4
2011-03-16 20:45:21.077 lottery[42697:903] Apr 20, 2011 = 6 and 46
2011-03-16 20:45:21.079 lottery[42697:903] Apr 27, 2011 = 26 and 53
2011-03-16 20:45:21.080 lottery[42697:903] May 4, 2011 = 28 and 80
2011-03-16 20:45:21.081 lottery[42697:903] May 11, 2011 = 90 and 24
2011-03-16 20:45:21.082 lottery[42697:903] May 18, 2011 = 72 and 86
Program ended with exit code: 0
```

Figure 3.9 Execution with Description

NSDate

Before moving on to any new ideas, let's examine **`NSDate`** in some depth. Instances of **`NSDate`** represent a single point in time and are basically immutable: You can't change the day or time once it is created. Because **`NSDate`** is immutable, many objects often share a single date object. There is seldom any need to create a copy of an **`NSDate`** object.

Here are some of the commonly used methods implemented by **`NSDate`**:

```
+ (id)date
```

Creates and returns a date initialized to the current date and time.

This is a *class method*. In the interface file, implementation file, and documentation, class methods are recognizable because they start with + instead of –. A class method is triggered by sending a message to the class instead of an instance. This one, for example, could be used as follows:

```
NSDate *now;
now = [NSDate date];
```

```
- (id)dateByAddingTimeInterval:(NSTimeInterval)interval
```

Creates and returns a date initialized to the date represented by the receiver *plus* the given `interval`.

```
- (NSTimeInterval)timeIntervalSinceDate:(NSDate *)anotherDate
```

Returns the interval in seconds between the receiver and `anotherDate`. If the receiver is earlier than `anotherDate`, the return value is negative. **`NSTimeInterval`** is the same as `double`.

`+ (NSTimeInterval)timeIntervalSinceReferenceDate`

Returns the interval in seconds between the first instant of January 1, 2001 GMT and the receiver's time.

`- (NSComparisonResult)compare:(NSDate *)otherDate`

Returns `NSOrderedAscending` if the receiver is earlier than `otherDate`, `NSOrderedDescending` if `otherDate` is earlier, or `NSOrderedSame` if the receiver and `otherDate` are equal.

Writing Initializers

Notice the following lines in your **`main`** function:

```
newEntry = [[LotteryEntry alloc] init];
[newEntry prepareRandomNumbers];
```

You are creating a new instance and then immediately calling **`prepareRandomNumbers`** to initialize `firstNumber` and `secondNumber`. This is something that should be handled by the initializer, so you are going to override the **`init`** method in your **`LotteryEntry`** class.

In the `LotteryEntry.m` file, change the method **`prepareRandomNumbers`** into an **`init`** method:

```
- (id)init
{
    self = [super init];
    if (self)
    {
        firstNumber = ((int)random() % 100) + 1;
        secondNumber = ((int)random() % 100) + 1;
    }
    return self;
}
```

The **`init`** method calls the superclass's initializer at the beginning, initializes its own variables, and then returns `self`, a pointer to the object itself (the object that is running this method). (If you are a Java or C++ programmer, `self` is the same as the `this` pointer.)

Now delete the following line in `main.m`:

```
[newEntry prepareRandomNumbers];
```

In `LotteryEntry.h`, delete the following declaration:

```
- (void)prepareRandomNumbers;
```

Build and run your program to reassure yourself that it still works.

Take another look at our **`init`** method. Why do we bother to assign the return value of the superclass's initializer to `self` and then test the value of `self`? The answer is that the initializers of some Cocoa classes will return `nil` if initialization was impossible. In order to handle these cases gracefully, we must both test the return value of `[super init]` and return the appropriate value for `self` from our initiailizer.

This pattern is debated among some Objective-C programmers. Some say that it is unnecessary, since most classes' initializers don't fail, and most classes' initializers don't return a different value for `self`. We believe it best to be in the habit of assigning to `self` and testing that value. The effort required is minimal compared to the debugging headaches that await you if you make an incorrect assumption about the superclass's behavior.

Initializers with Arguments

Look at the same place in `main.m`. It should now look like this:

```
LotteryEntry *newEntry = [[LotteryEntry alloc] init];
[newEntry setEntryDate:iWeeksFromNow];
```

It might be nicer if you could supply the date as an argument to the initializer. Change those lines to look like this:

```
LotteryEntry *newEntry = [[LotteryEntry alloc]
                                     initWithEntryDate:iWeeksFromNow];
```

You may see a compiler error; ignore it, as we are about to fix the problem.

Next, declare the method in `LotteryEntry.h`:

```
- (id)initWithEntryDate:(NSDate *)theDate;
```

Now, change (and rename) the **`init`** method in `LotteryEntry.m`:

```
- (id)initWithEntryDate:(NSDate *)theDate
{
    self = [super init];
    if (self)
    {
        entryDate = theDate;
        firstNumber = ((int)random() % 100) + 1;
        secondNumber = ((int)random() % 100) + 1;
    }
    return self;
}
```

Build and run your program. It should work correctly.

However, your class **LotteryEntry** has a problem. You are going to e-mail the class to your friend Rex. Rex plans to use the class **LotteryEntry** in his program but might not realize that you have written **initWithEntryDate:**. If he made this mistake, he might write the following lines of code:

```
NSDate *today = [NSDate date];
LotteryEntry *bigWin = [[LotteryEntry alloc] init];
[bigWin setEntryDate:today];
```

This code will not create an error. Instead, it will simply go up the inheritance tree until it finds **NSObject**'s **init** method. The problem is that `firstNumber` and `secondNumber` will not get initialized properly—both will be zero.

To protect Rex from his own ignorance, you will override **init** to call your initializer with a default date:

```
- (id)init
{
    return [self initWithEntryDate:[NSDate date]];
}
```

Add this method to your `LotteryEntry.m` file.

Note that **initWithEntryDate:** still does all the work. Because a class can have multiple initializers, we call the one that does the work the *designated initializer*. If a class has several initializers, the designated initializer typically takes the most arguments. You should clearly document which of your initializers is the designated initializer. Note that the designated initializer for **NSObject** is **init**.

Conventions for Creating Initializers (rules that Cocoa programmers try to follow regarding initializers):

- You do not have to create any initializer in your class if the superclass's initializers are sufficient.
- If you decide to create an initializer, you must override the superclass's designated initializer.

- If you create multiple initializers, only one does the work—the designated initializer. All other initializers call the designated initializer.
- The designated initializer of your class will call its superclass's designated initializer.

The day will come when you will create a class that must, must, must have some argument supplied. Override the superclass's designated initializer to throw an exception:

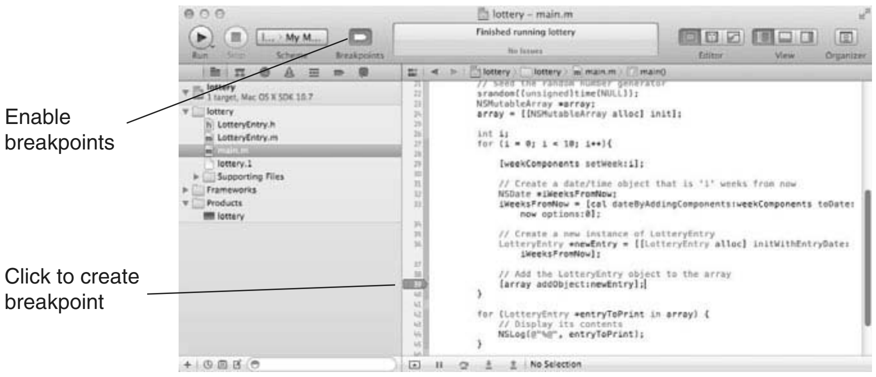
```
- (id)init
{
    @throw [NSException exceptionWithName:@"BNRBadInitCall"
                 reason:@"Initialize Lawsuit with initWithDefendant:"
               userInfo:nil];
    return nil;
}
```

The Debugger

The Free Software Foundation developed the compiler (gcc) and the debugger (gdb) that come with Apple's developer tools. Apple has made significant improvements to both over the years. This section discusses the processes of setting breakpoints, invoking the debugger, and browsing the values of variables.

While browsing code, you may have noticed a gray margin to the left of your code. If you click in that margin, a breakpoint will be added at the corresponding line. Add a breakpoint in `main.m` at the following line (Figure 3.10):

```
[array addObject:newEntry];
```



```
    // Seed the random number generator
    srandom((unsigned)time(NULL));
    NSMutableArray *array;
    array = [[NSMutableArray alloc] init];

    int i;
    for (i = 0; i < 10; i++){

        [weekComponents setWeek:i];

        // Create a date/time object that is 'i' weeks from now
        NSDate *iWeeksFromNow;
        iWeeksFromNow = [cal dateByAddingComponents:weekComponents toDate:
            now options:0];

        // Create a new instance of LotteryEntry
        LotteryEntry *newEntry = [[LotteryEntry alloc] initWithEntryDate:
            iWeeksFromNow];

        // Add the LotteryEntry object to the array
        [array addObject:newEntry];
    }

    for (LotteryEntry *entryToPrint in array) {
        // Display its contents
        NSLog(@"%@", entryToPrint);
    }
```

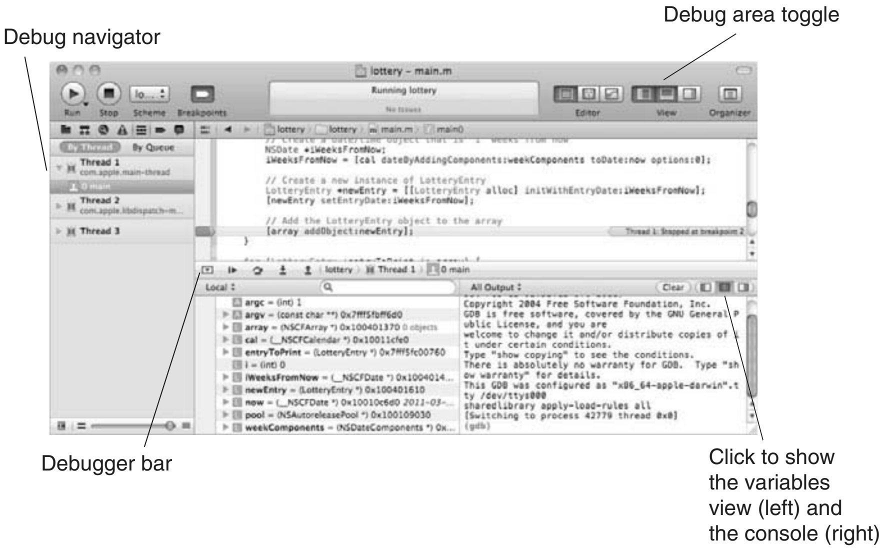
Figure 3.10 Creating a Breakpoint

When you run the program, Xcode will start the program in the debugger if you have any breakpoints. To test this, run it now. The debugger will take a few seconds to get started, and then it will run your program until it hits the breakpoint.

When your application is running, the debugger bar will be shown below the editor area. The debugger bar contains a button to toggle visibility of the full debugger area, including the variables view and console, as well as buttons to control the execution of your program and information about the current thread and function.

Xcode's default behavior is to show the full debugger area when a breakpoint is hit. If you do not see the debugger area at the bottom of the window, use the debugger area view toggle in the debugger bar (or toolbar), or the View->Show Debugger Area menu item.

You should also see the Debug navigator on the left, which shows the threads in our application and frames on the stack for each thread. Because the breakpoint is in **`main()`**, the stack is not very deep. In the variables view on the left in the debugger area, you can see the variables and their values (Figure 3.11).



```
NSDate *iWeeksFromNow;
iWeeksFromNow = [cal dateByAddingComponents:weekComponents toDate:now options:0];

// Create a new instance of LotteryEntry
LotteryEntry *newEntry = [[LotteryEntry alloc] initWithEntryDate:iWeeksFromNow];
[newEntry setEntryDate:iWeeksFromNow];

// Add the LotteryEntry object to the array
[array addObject:newEntry];
}
```

```
argc = (int) 1
argv = (const char **) 0x7fff5fbff6d0
array = (NSCFArray *) 0x100401370 0 objects
cal = (__NSCFCalendar *) 0x10011cfe0
entryToPrint = (LotteryEntry *) 0x7fff5fc00760
i = (int) 0
iWeeksFromNow = (__NSCFDate *) 0x1004014...
newEntry = (LotteryEntry *) 0x100401610
now = (__NSCFDate *) 0x10010c6d0 2011-03-...
pool = (NSAutoreleasePool *) 0x100109030
weekComponents = (NSDateComponents *) 0x...
```

```
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General P
ublic License, and you are
welcome to change it and/or distribute copies of i
t under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "sh
ow warranty" for details.
This GDB was configured as "x86_64-apple-darwin".t
ty /dev/ttys000
sharedlibrary apply-load-rules all
[Switching to process 42779 thread 0x0]
(gdb)
```

Figure 3.11 Stopped at a Breakpoint

Note that the variable `i` is currently 0.

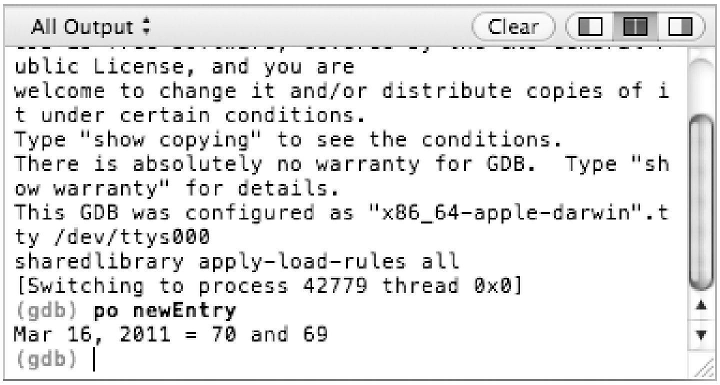
Return your attention to the debugger bar. Four of the buttons above the variables view are for pausing (or continuing) and stepping over, into, and out of functions. Click the `Continue` button to execute another iteration of the loop. Click the `Step-Over` button to walk through the code line by line.

The gdb debugger, being a Unix thing, was designed to be run from a terminal. When execution is paused, the gdb terminal will appear in the Console panel.

In the debug console, you have full access to all of gdb's capabilities. One very handy feature is "print-object" (po). If a variable is a pointer to an object, when you po it, the object is sent the message **`description`**, and the result is printed in the console. Try printing the `newEntry` variable.

```
po newEntry
```

You should see the result of your **`description`** method (Figure 3.12).



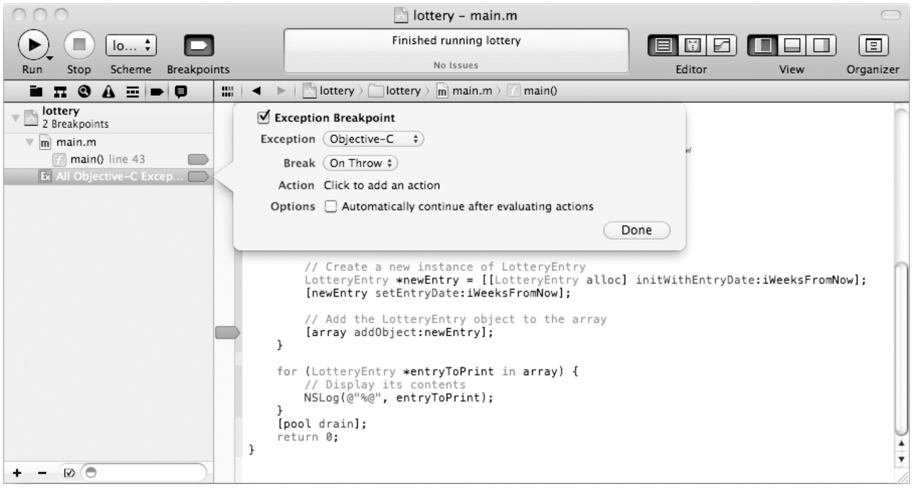
```
ublic License, and you are
welcome to change it and/or distribute copies of i
t under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "sh
ow warranty" for details.
This GDB was configured as "x86_64-apple-darwin".t
ty /dev/ttys000
sharedlibrary apply-load-rules all
[Switching to process 42779 thread 0x0]
(gdb) po newEntry
Mar 16, 2011 = 70 and 69
(gdb) |
```

Figure 3.12 Using the gdb Console

Exceptions are raised when something goes very wrong. To make the debugger stop whenever an exception is thrown, you will want to add an exception breakpoint. Click the Add button at the bottom of the breakpoint navigator and select Add Exception Breakpoint.... Set the exception type to Objective-C and click Done (Figure 3.13). Disable the existing breakpoint in `main()` by clicking on the blue breakpoint icon in the breakpoint navigator. The breakpoint will be dimmed when it is disabled.

You can test this exception breakpoint by asking for an index that is not in an array. Immediately after the array is created, ask it what its first object is:

```
array = [[NSMutableArray alloc] init];
NSLog(@"first item = %@", [array objectAtIndex:0]);
```



Exception Breakpoint
Exception: Objective-C
Break: On Throw
Action: Click to add an action
Options: ☐ Automatically continue after evaluating actions

```
        // Create a new instance of LotteryEntry
        LotteryEntry *newEntry = [[LotteryEntry alloc] initWithEntryDate:iWeeksFromNow];
        [newEntry setEntryDate:iWeeksFromNow];

        // Add the LotteryEntry object to the array
        [array addObject:newEntry];
    }

    for (LotteryEntry *entryToPrint in array) {
        // Display its contents
        NSLog(@"%@", entryToPrint);
    }
    [pool drain];
    return 0;
}
```

Figure 3.13 Adding an Exception Breakpoint

Rebuild and restart the program. It should stop when the exception is raised.

One of the challenging things about debugging Cocoa programs is that they will often limp along in a broken state for quite a while. Using the macro **`NSAssert()`**, you can get the program to throw an exception as soon as the train leaves the track. For example, in **`setEntryDate:`**, you might want an exception thrown if the argument is `nil`. Add a call to **`NSAssert()`**:

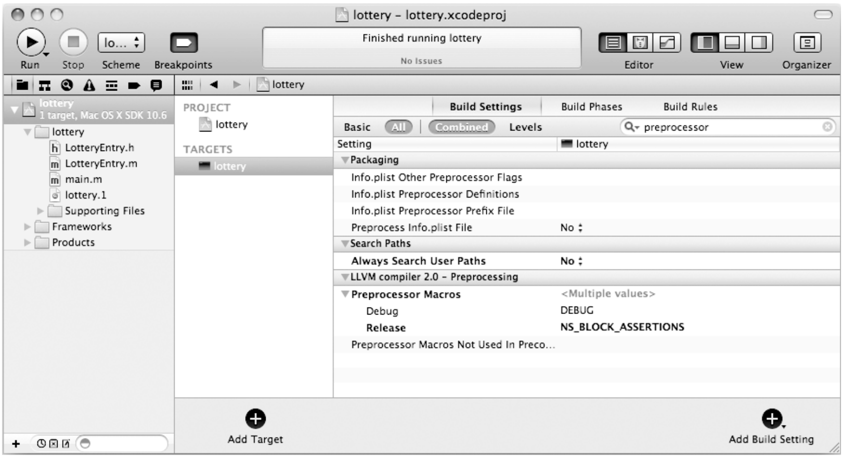
```
- (id)initWithEntryDate:(NSDate *)theDate
{
    self = [super init];
    if (self) {
        NSAssert(theDate != nil, @"Argument must be non-nil");
        entryDate = theDate;
        firstNumber = ((int)random() % 100) + 1;
        secondNumber = ((int)random() % 100) + 1;
    }
    return self;
}
```

Build it and run it. Your code, being correct, will not throw an exception. So change the assertion to something incorrect:

```
NSAssert(theDate == nil, @"Argument must be non-nil");
```

Now build and run your application. Note that a message, including the name of the class and method, is logged and an exception is thrown. Wise use of **`NSAssert()`** can help you hunt down bugs much more quickly.

You probably do not need your assert calls checked in your completed product. On most projects, there are two build configurations: Debug and Release. In the Debug version, you will want all your asserts checked. In the Release configuration, you will not. You will typically block assertion checking in the Release configuration (Figure 3.14).



```
Setting                                         lottery
▼Packaging
  Info.plist Other Preprocessor Flags
  Info.plist Preprocessor Definitions
  Info.plist Preprocessor Prefix File
  Preprocess Info.plist File                    No
▼Search Paths
  Always Search User Paths                      No
▼LLVM compiler 2.0 - Preprocessing
▼Preprocessor Macros                            <Multiple values>
    Debug                                       DEBUG
    Release                                     NS_BLOCK_ASSERTIONS
  Preprocessor Macros Not Used In Preco...
```

Figure 3.14 Disabling Assertion Checking

To do this, bring up the build settings by selecting the lottery project in the project navigator (topmost item). Then select the `lottery` target, change to the Build Settings tab, and find the Preprocessor Macros item. A quick way to find it is to use the search field at the top of the Build Settings panel. The Preprocessor Macros item will have one item beneath it for each build configuration: `Debug` and `Release`. Set the Release item value to `NS_BLOCK_ASSERTIONS`.

Now, if you build and run the Release configuration, you'll see that your assertion is not getting checked. (Before going on, fix your assertion: It should ensure that dates are *not* nil.)

You can change your current build configuration to Release by opening the scheme editor (in the Product menu, click Edit Scheme...). Select the Run action; on the `Info` panel, change Build Configuration to Release. Now when you build and run your application, it will be built using the Release configuration. Note that the default build configuration for the Archive action is Release. We will discuss build configurations in more detail in Chapter 37.

`NSAssert()` works only inside Objective-C methods. If you need to check an assertion in a C function, use **`NSCAssert()`**.

That's enough to get you started with the debugger. For more in-depth information, refer to the documentation from the Free Software Foundation (www.gnu.org/).

What Have You Done?

You have written a simple program in Objective-C, including a **`main()`** function that created several objects. Some of these objects were instances of **`LotteryEntry`**, a class that you created. The program logged some information to the console.

At this point, you have a fairly complete understanding of Objective-C. Objective-C is not a complex language. The rest of the book is concerned with the frameworks that make up Cocoa. From now on, you will be creating event-driven applications, not command-line tools.

Meet the Static Analyzer

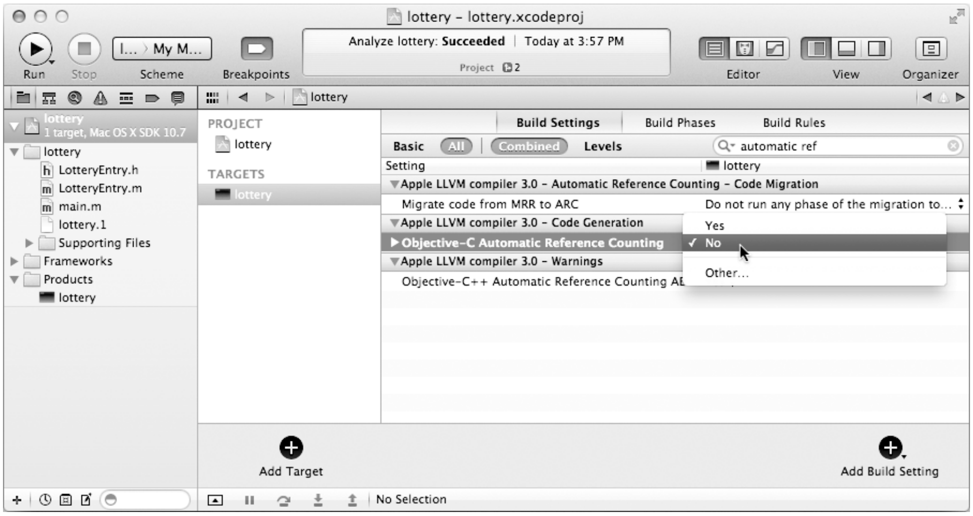
One of the handiest tools in Xcode is the static analyzer. The static analyzer uses Apple's LLVM compiler technology to analyze your code and find bugs. Traditionally, developers have relied on compiler warnings for hints on potential trouble areas in their code. The static analyzer goes much deeper, looking past syntax and tracing how values are used within your code.

Because of the default compiler settings and our careful typing, you should find, if you run the analyzer now, that our application has no issues as it stands. Let's modify our project settings so that we can better see the static analyzer at work.

As we did before, open the project's build settings by selecting the project in the project navigator on the left. Then select the lottery target. In the Build Settings tab, find the setting for Objective-C Automatic Reference Counting. Change its value to No (Figure 3.15).

Now analyze the lottery application. In the Product menu, click Analyze. In the issues navigator, you will see several issues found by the static analyzer; select one and drill down in the tree to examine the analyzer's thought process (Figure 3.16).

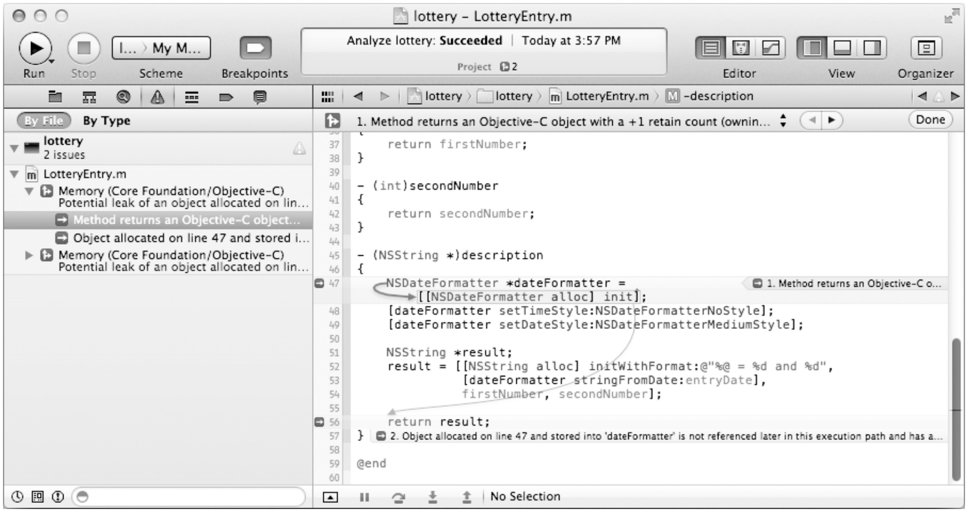
In this case, the static analyzer has found a number of memory-related problems in our program because we disabled a feature called automatic reference counting, which we will discuss in the next chapter. This is one of the more



Analyze lottery: **Succeeded** | Today at 3:57 PM

Setting	lottery
▼Apple LLVM compiler 3.0 – Automatic Reference Counting – Code Migration	
Migrate code from MRR to ARC	Do not run any phase of the migration to…
▼Apple LLVM compiler 3.0 – Code Generation	
▶Objective-C Automatic Reference Counting	Yes [✓ No] Other…
▼Apple LLVM compiler 3.0 – Warnings	
Objective-C++ Automatic Reference Counting AE	

Figure 3.15 Disable Automatic Reference Counting



Analyze lottery: **Succeeded** | Today at 3:57 PM

lottery
2 issues
LotteryEntry.m
Memory (Core Foundation/Objective-C)
Potential leak of an object allocated on lin…
Method returns an Objective-C object…
Object allocated on line 47 and stored i…
Memory (Core Foundation/Objective-C)
Potential leak of an object allocated on lin…

1. Method returns an Objective-C object with a +1 retain count (ownin…

```
    return firstNumber;
}

- (int)secondNumber
{
    return secondNumber;
}

- (NSString *)description
{
    NSDateFormatter *dateFormatter =
        [[NSDateFormatter alloc] init];
    [dateFormatter setTimeStyle:NSDateFormatterNoStyle];
    [dateFormatter setDateStyle:NSDateFormatterMediumStyle];

    NSString *result;
    result = [[NSString alloc] initWithFormat:@"%@ = %d and %d",
              [dateFormatter stringFromDate:entryDate],
              firstNumber, secondNumber];

    return result;
}

@end
```

1. Method returns an Objective-C o…

2. Object allocated on line 47 and stored into 'dateFormatter' is not referenced later in this execution path and has a…

Figure 3.16 The Static Analyzer at Work

useful aspects of the static analyzer: It knows the rules for retain-count memory management in Objective-C, and it can also identify other dangerous patterns in your code.

Leave automatic reference counting disabled for now.

For the More Curious: How Does Messaging Work?

As mentioned earlier, an object is like a C struct. **`NSObject`** declares an instance variable called `isa`. Because **`NSObject`** is the root of the entire class inheritance tree, every object has an `isa` pointer to the class structure that created the object (Figure 3.17). The class structure includes the names and types of the instance variables for the class. It also has the implementation of the class's methods. The class structure has a pointer to the class structure for its superclass.

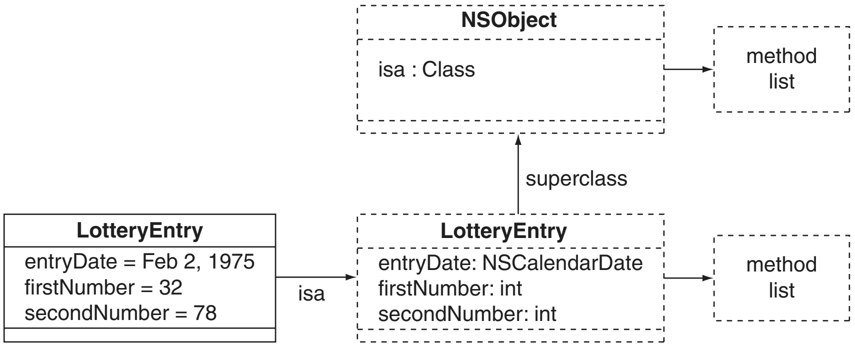
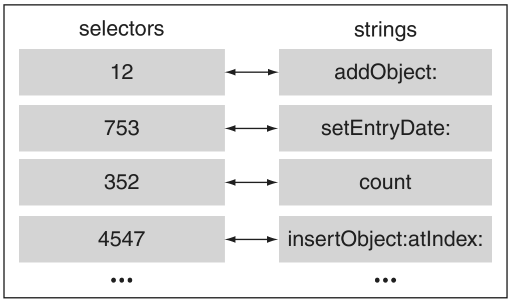


Figure 3.17 Each Object Has a Pointer to Its Class

The methods are indexed by the selector. The selector is of type `SEL`. Although `SEL` is defined to be `char *`, it is most useful to think of it as an `int`. Each method name is mapped to a unique `int`. For example, the method name **`addObject:`** might map to the number 12. When you look up methods, you will use the selector, not the string `@"addObject:"`.

As part of the Objective-C data structures, a table maps the names of methods to their selectors. Figure 3.18 shows an example.



selectors	strings
12	addObject:
753	setEntryDate:
352	count
4547	insertObject:atIndex:
...	...

Figure 3.18 The Selector Table

At compile time, the compiler looks up the selectors wherever it sees a message send. Thus,

```
[myObject addObject:yourObject];
```

becomes (assuming that the selector for **`addObject:`** is 12)

```
objc_msgSend(myObject, 12, yourObject);
```

Here, **`objc_msgSend()`** looks at `myObject`'s `isa` pointer to get to its class structure and looks for the method associated with 12. If it does not find the method, it follows the pointer to the superclass. If the superclass does not have a method for 12, it continues searching up the tree. If it reaches the top of the tree without finding a method, the function throws an exception.

Clearly, this is a very dynamic way of handling messages. These class structures can be changed at runtime. In particular, using the **`NSBundle`** class makes it relatively easy to add classes and methods to your program while it is running. This very powerful technique has been used to create applications that can be extended by other developers.

Challenge

Use **`NSDateFormatter`**'s **`setDateFormat:`** to customize the format string on the date objects in your **`LotteryEntry`** class.

Index

Symbols

: (colon), method name with arguments, 36
@"..." construct, 47
@ symbol
  C strings vs. NSStrings, 40–41
  Objective-C keywords, 27
^operator, blocks, 372, 374

A

abstract class
  defined, 160
  NSCoder as, 160
  NSController as, 129–130
acceptsFirstResponder method, keyboard events, 272–275, 280, 282
accessor methods
  declaring for new class, 50–51
  defined, 50
  implementing, 123–125
actions
  implicit animation and, 423–424
  targets and. *See* target/action
actions dictionary, 423
addObject method
  add objects to end of array, 36
  NSMutableArray, 46
addObjectsFromArray: method, NSMutableArray, 46
addOperationWithBlock: method, NSOperationQueue, 436–437
Alert panel
  as modal window, 336–337
  overview of, 229–230
  using string table, 241
alloc method
  coupling with init message, 43–44
  retain-count rules for ownership, 76
  retain count using, 69
AppKit framework.
  classes with delegates in, 112
  defined, 6
  UIKit vs. *See* iOS development
applications
  debugging hints, 98
  as directories, 172
  distributing your. *See* distributing your application
ARC (automatic reference counting)
  benefits and limitations, 68–69
  defined, 68
  disabling, 63–64
  overview of, 80–81
  strong references, 81
  weak references, 81–82
archiving
  automatic document saving, 174
  document architecture, 163–167
  loading and NSKeyedArchiver, 168–169
  NSCoder and NSCoding, 160–163
  overview of, 159–160
  preventing infinite loops, 172–173
  saving and NSKeyedArchiver, 167–168
arguments
  initializers with, 56–58
  methods taking, 36–37
arrangedObjects controller key, array controller, 136
array controllers
  NSArrayController. *See* NSArrayController
arrays
  methods implemented by NSArray, 45
  methods implemented by NSMutableArray, 46
asserts, debugging with, 61–62
assign attribute, properties, 125
Assistant Editor
  editing implementation file, 27

Assistant Editor, *continued*
  layout XIB file, 91
  making connections, 92–94
asynchronous connections
  NSURLConnection, 362–363
  receiving response, 377–381
  solving I/O- bound problems with, 434
asynchronous reads, NSTask, 456
attributed strings. *See* NSAtrributedString
attributes
  NSAtrributedString, 287–288
  in NSManagedObjectModel, 178–179
Attributes Inspector. *See* attributes
automatic document saving, 174
automatic reference counting. *See* ARC (automatic reference counting)
autorelease message
  accessor methods for instance variables, 79
  overview of, 74–76
  retain-count rules, 76–77
autorelease pools
  background threads, 430–431
  simplifying release of objects, 74–76
autosavesInPlace method, NSDocument, 174
autoscroll: method, 268–269
autoscrolling
  adding timer to, 328
  adding to application, 268–269
Autosizing, Size Inspector, 244, 445
awakeFromNib message
  chronology of applications, 32
  overview of, 30–31

B

background color
  table views, 216–217
background threads
  overview of, 427–428
  race conditions in, 428–429
  using NSOperationQueue, 435–438
becomeFirstResponder method, 272–275, 281
Behaviors in Preferences panel, configuring to show log in, 29
bindings
  attaching value transformers to, 351
  on cell-based vs. view-based table views, 191
  Core Data, 177, 183–188
  creating programmatically, 127
  removing, 127
  using key-value coding, 119–120
Bindings Inspector. *See* bindings
_block type specifier, 374–375
blocks
  availability of, 376
  memory and objects within, 375–376
  overview of, 371–373
  receiving asynchronous response, 377–381
  syntax for, 373–375
bold, drawing text with attributes, 294
BOOL, defined, 26
box
  binding, 187–188
  view swapping. *See* view swapping
breakpoint navigator, 60
breakpoints, 58–61
build configurations
  assertion checking, 62
  changing behavior, 462–464
  changing to Release configuration, 62
  distributing app, 461–462
bundles. *See* NSBundle
buttons
  NSButton, 85–86
  target and action of, 83–84

C

C++ language, Objective-C vs., 2
C programming language
  assertion checking, 62
  interaction with ARC, 82
  strings in Objective-C vs., 40–41
CAAnimation class, 417
CABasicAnimation class, 424
CALayer class, Core Animation
  configuring project, 417–423
  custom drawing, 425
  implicit animation and actions, 423–424
  overview of, 417
CAOpenGLLayer, 425
case sensitivity, 7–8, 21
caseInsensitiveCompare: method
  NSString, 47
  sorting with array controllers, 140–141
CAShapeLayer, 425
categories
  adding method to NSString, 307–309

creating, 307
declaring private methods, 309
CATextLayer, Core Animation, 420–421, 425
CATransaction, Core Animation, 417, 424
cell-based table views
bindings in, 183
limitations of, 180
view-based table views vs., 191
center-justify button, text field Attributes Inspector, 19
changeBackgroundColor: message
adding panel to application, 204
editing defaults, 214–215
notifications, 224–227
characters method, NSEvent, 273
characters with accents
Option key, 235, 237
clang/LLVM open source compiler projects, 35
class method, 54
classes
creating in Interface Builder, 20–22
creating own. *See* Objective-C, creating own classes
creating/using existing. *See* Objective-C, using existing classes
declaring in Objective-C vs. Java, 25–26
denoting with NS prefix, 43
framework as collection of, 6
implementing protocols, 160–163
overview of, 5
typographical conventions for, 7
clearContents method, NSPasteboard, 296
close button, panels, 194
Cocoa Touch
defined, 3
developing for iOS. *See* iOS development
Cocoa, understanding
common mistakes, 8
frameworks, 6
history of, 1–3
how to learn, 8–9
language, 4–5
objects, classes, methods, and messages, 5–6
tools, 3–4
Code Signing and Application Sandboxing Guide, 468
colon (:), method name with arguments, 36
command line tool, creating, 37–41
compare: method, sorting array controllers, 140–141
compositing image onto view, 264–265
concludeDragOperation: message, drag destination, 316, 318
concurrency
multithreading, 427–429
NSOperationQueue and, 435–438
overview of, 427
simple background threads, 429–431
time profiling in Instruments, 431–435
Connection panel
make connections in Interface Builder, 93–95
Connection to Action, 92
Connection to Outlet, 92–93
connections
Interface Builder, 15, 22–25
console, debugging with, 29, 98
constants, Objective-C, 26
containsObject method, NSArray, 45
content view, view hierarchy, 241–243
contentsGravity property, Core Animation, 423
context pointer, using defensively, 153
contextInfo, sheets, 329, 335–336
Continue button, debugger bar, 60
controller classes
NSArrayController. *See* NSArrayController
NSController, 129
NSObjectController, 129–130
controls
layout XIB file, 90–94
NSButton, 85–86
NSSlider, 86–87
NSTextField, 87–89
setting target programmatically, 96
target/action of, 83
Convert to Objective-C Automatic Reference Counting, 82
coordinate system, views, 266–268
copy. *See also* drag-and-drop
blocks, when assigning to instance variables, 375
copy attribute, properties, 125
copy method, retain-count rules for ownership, 76
Core Animation
CALayer, 425

Core Animation, *continued*
implicit animation and actions, 423–424
overview of, 417
creating custom NSManagedObject classes, 409–411
editing model, 407–408
Core Data framework
creating applications, 130
defined, 6
NSManagedObjectModel, 177–179
overview of, 177
Core Graphics framework. *See* Quartz
count method, NSArray, 45
currentContext, drawing to screen, 358
currentEvent method, NSApplication, 328
cut, adding, 298–300

D

dataOfType:error: method, 168, 455
dataSource outlet, NSTableView
make connections, 109
date method, NSDate, 54–55
dateByAddingTimeInterval method, NSDate, 54
dealloc method
in ARC, 81
overview of, 72–73
deallocating objects, debugging hints, 98
Debug build configuration
using in development, 98
assertion checking, 62
overview of, 461
Debug navigator, 59
debugger (gdb)
adding exception breakpoints, 60–61
console, 29
debugger bar, 59–60
defined, 5
hints for, 98
print-object feature, 60
using NSAssert(), 61
debugging, with static analyzer, 63–64
DebugLog function, preprocessor macros, 463–464
decoding data, NSCoder, 162–163
defaults. *See* user defaults
defaults tool, 217–218
delegates
AppKit framework classes, 112
creating, 114
creating helper objects, 100–104
designing, 381
errors in implementing, 112
notifications and, 227
of NSControl, 347–348
NSTableView and, 104–107
pasteboards, nil-targeted actions and, 301
understanding, 113–114
dequeueReusableCellWithIdentifier: method, UITableView in iOS, 393
description method
autoreleasing objects, 73–74
implementing, 52–53
NSObject, 44
printing in debug console, 60
designated initializer, 57
destination, drag, 315–319
dictionaries, 208–209
disk image (DMG), packaging application for download, 466
distributing your application
application sandboxing, 466–468
build configurations, 461–462
Mac App Store, 468–470
preprocessor macros and changing behavior with build configurations, 462–464
release build, 464–466
DMG (disk image), packaging application for download, 466
dock, Interface Builder, 20
document architecture
defined, 131
Info.plist and NSDocumentController, 163–164
NSDocument, 164–167
NSWindowController, 167
overview of, 163
document controller, 163–164
document type information, setting, 170
Documentation and API Reference, 31
documents. *See also* NSDocument
applications based on, 131
suppressing creation of untitled, 215–216
domains, setting precedence of defaults, 212
dot notation, accessors, 124

drag-and-drop
  drag destination, 315–319
  drag source, 312–315
  operation mask, 319
  overview of, 311–312
draggingEntered: message, 316–317
draggingExited: message, 315, 318
draggingSourceOperationMaskForLocal: message, 312–317, 319
draggingUpdated: message, 315, 316, 319
drawInRect: method, NSAttributedString, 309
drawLayer:inContext:, CALayer, 425
drawRect: method
  adding highlighting, 316–317
  adding printing, 355
  compositing of image, 265
  coordinate system of views, 268
  drawing text with attributes, 291
  getting view to draw itself, 246–248
  getting view to generate PDF data, 291–293
  rollovers, 283
  using keyboard events, 280

E

encodeWithCoder: method, NSCoder, 160–161
encoding data
  NSCoder, 160–161
  preventing infinite loops, 172–173
endSheet: method, 329
en.lproj directory, localizing app in English, 233, 237
entities
  Core Data model, 180, 407
  editing model, 407–408
  in NSManagedObjectModel, 177–179
equality, in NSObject, 44
events
  images and mouse. *See* images and mouse events
  keyboard. *See* keyboard events
  nextResponder and, 414–415
  role of window server, 33
  timer. *See* NSTimer
exceptions
  adding breakpoints, 60–61
  debugging hints, 98
  key-value coding and nil, 139–140
exported UTIs
  configuring, 170–171
  customized, 303
  storing in UTExportedTypeDeclarations, 175
Extended Detail pane, Instruments, 433
extensions
  Objective-C, 40–41
  setting for file types, 170–172

F

file formats
  copying data onto pasteboard, 295
File Inspector, 234, 237–238
file system, NSUserDefaults and, 210–211
file wrappers, 165–167
File's Owner
  dock icon, 20
  NIB files and NSWindowController, 198
First Responder, dock icon, 20
flagsChanged event, NSResponder, 273
floats, nil value and, 139
fonts
  NSFont. *See* NSFont
  NSFontManager, 293
  typographical conventions in this book, 7–8
format strings, explicit ordering of tokens in, 240
formatters. *See also* number formatters
  creating. *See* NSFormatters
  NSTextField, 88
  overview of, 339–340
forwardInvocation: method, 145
Foundation framework
  creating command-line tool, 38–41
  defined, 6
.framework extension, 6
fr.lproj directory, localizing app in French, 233–238, 237–238
fromValue property, CABasicAnimation, 424
focus ring, fuzzy blue box, 284

G

garbage collector, 67–68
gcc (GNU C compiler), 35, 58–59
GCD (*See* Grand Central Dispatch)
  and blocks, 376
  faster performance and, 439

gdb. *See* debugger (gdb)
general pasteboard, 296
generalPasteboard method, NSPasteboard, 296
genstrings tool, 239
global variables
  creating keys for names of defaults, 212
  NSAtrributedString, 287–288
  NSPasteboard, 297
  notifications, 225
GNU C compiler (gcc), 35, 58–59
gradients, CAGradientLayer, 425
Grand Central Dispatch (GCD)
  and blocks, 376
  faster performance and, 439
graphics. *See* Core Animation

H

header file
  creating classes, 20
  creating new class, 50
  in Objective-C, 26–27
helper objects
  classes with delegate outlets, 112
  delegates, 100–104
  errors implementing delegates, 112
  how delegates work, 113–114
  making connections, 109
  NSTableView and its dataSource, 104–107
  overview of, 99
hidesOnDeactivate variable, windows, 194
hierarchy, views, 241–243
highlights
  for color well, 204
  for erroneous line of code, 28
  drag destination, 316–317
  for rollovers, 283
HTTP, Web services, 359

I

IBAction
  adding cut, copy and paste, 300
  defined, 26
  making connections, 92
IBOutlet, 26
ibtool command, 239–240
.icns file, 170–172
icons
  copying into project directory, 170
  dock, 20
  setting for file types, 170–172
id, 26
identifier
  interface layout for Web service, 367
  setting document type in archiving, 170
  setting for defaults, 212
Identity Inspector
  instance of view subclass, 244
  labeling objects, 180
image view
  binding, 186–187
  view-based table views, 181–184
images and mouse events
  autoscrolling, 268–269
  composite images, 264–265
  getting mouse events, 259
  NSEvent, 257–258
  NSImage, 269
  NSOpenPanel, 259–264
  NSResponder, 257
  view's coordinate system, 266–268
immutable, NSArray as, 45
implementation file
  creating classes, 20
  editing, 27–28
#import
  creating keys for names of defaults, 212
  header file and, 26–27
Info.plist, NSDocumentController, 163–164
inheritance
  "inherits from", vs. "uses" or "knows about", 48
  NSControl and, 84–85
  in Objective-C, 26
  in Objective-C vs. C++, 48
init method
  creating and using instances, 36
  as designated initializer for NSObject, 57
  initializers with arguments and, 56–58
  NSObject, 43
  writing initializers, 55–56
initialFirstResponder method, keyboard events, 276–278
initialize method, registering defaults, 213
initializers. *See also* init method
  with arguments, 56–58
  conventions for creating, 57–58

initWithCoder: method, NSCoder, 160, 163
initWithFormat method, NSString, 47
insertObject method, NSMutableArray, 46
Inspection Range, Instruments, 433
Inspector panel, 16
installers, 464–466
instance variables
  adding to classes, 21
  archiving, 159
  defined, 5
  designing classes with, 148–149
  enabling accessor methods for, 77–80
  header files in Objective-C and, 26–27
  isa pointer, 65–66
  naming conventions for, 22
  protected in Objective-C, 27
instances, creating, 5, 22, 35–37
Instruments
  time profiling in, 431–434
  defined, 4
Interface Builder
  common mistakes, 8
  creating an instance, 22
  creating class, 20–22
  dock, 20
  functionality of, 15–16
  interface layout, 17–19
  making connections in, 22–25, 91–94
  XIBs and NIBs, 17
interface layout
  in Interface Builder, 17–19
  overview of, 17–19
  panels, 201–202
  sheets, 331–334
International pane, System Preferences, 235
interpretKeyEvents method, NSResponder, 281
Invert Call Tree, Time Profiler, 433
invocations, 145–148
iOS development
  adding navigation controller, 388–390
  overview of, 383
  pushing view controllers, 393–395
  UITableViewController, 391–393
iOS SDK, 3
iPad apps, 383
iPhone apps, 383
isa pointer, messaging, 65–66
isARepeat method, NSEvent, 273
isEqual: message, NSArray, 45
isEqual: message, NSObject, 44
isFlipped, flipping views, 255
isOpaque method, project with keyboard events, 280
Issue navigator, 28
italics, drawing text with attributes, 294

J

Jump to Next Counterpart command, editing implementation file, 27

K

key paths
  key-value observing, 127
  overview of, 126–127
  sorting people, 142
key-value coding
  for bindings, 119–120
  making keys observable, 123
  and nil, 139–140
  overview of, 117–119
  to-many relationships and, 148–152
key-value observing
  enabling undo for edits, 152–153
  key paths, 126–127
  making keys observable, 121–123
  overview of, 120–121
  properties, 124–126
  understanding, 127
key-value pairs
  dictionaries, 208
  string table as collection of, 236
keyboard events
  nextResponder, 414–415
  NSEvent, 273–274
  NSResponder, 273
  overview of, 271–272
keyCode method, NSEvent, 274
keyDown event, 273, 281
keyUp event, NSResponder, 273
keywords, Objective-C, 27
KITT the super car, 99
Knight Industries, 99
“knows about”, “inherits from” vs., 48
KVC. *See* key-value coding

L

labels
  objects in Interface Builder editor, 180
language
  Objective-C, 4–5
  Objective-C vs. C++, 2
Language and Text pane, System Preferences, 236
lastObject method, NSArray, 45
layers
  Core Animation, 417, 419–421
  implicit animation and actions, 423–424
lazy copying, drag and drop, 295–298, 304–305
length member, NSRange, 286
level indicator, 181–184
libraries. *See* framework
library of code, bundles, 205
Library panel, 17–19
LLVM (Low Level Virtual Machine) compiler
  compiling Objective C-2, 5
  defined, 35
  static analyzer using, 63–64
loading
  NIB files. *See* NIB files and NSWindowController
  NSKeyedUnarchiver and, 168–169
  using NSDocument, 166–167
Localizable.strings, 236–238
localization
  explicit ordering of tokens in format strings and, 240
  ibtool and, 239–240
  Nib file and, 234–236
  overview of, 233–234
  string tables and, 236–239
location member, NSRange, 286
lock focus on view, drawRect, 246–247
Low Level Virtual Machine. *See* LLVM (Low Level Virtual Machine) compiler

M

Mac App Store
  distributing app, 468–469
  for Mac developers, 3
  receipt validation, 469–470
Mac OS X
  Developer Tools, 3–4
macros
  hardcoding behavioral settings with preprocessor, 462–464
  using string table, 238–239
main function
  changing, 51–52
  creating command-line tool, 38–41
  initializers with arguments, 56–58
  overview of, 15
MainMenu.xib
  menu item setup for adding panels, 197
  Interface Builder, 15
man command, for ibtool, 240
managed object model, NSManagedObjectModel, 177–181, 189
manual reference counting
  ARC working with, 82
  autoreleasing objects, 73–76
  dealloc method, 72–73
  in iOS 5, 383
  limitations of, 68
  memory management, 67
  overview of, 69–70
  retain-count rules, 76–77
matrices, creating with cells, 253–254
mediated file access, and Powerbox, 468
memory management
  accessor methods, 77–80
  ARC, 80–82
  within blocks, 375–376
  iOS 5, 383
  manual reference counting. *See* manual reference counting
  overview of, 67–69
  static analyzer troubleshooting, 63–64
  thinking locally for, 77
menu items, target/action of, 95
messages
  calling methods by sending objects, 6
  handling, 65–66
  sending to nil, 41–42
  syntax for, 35–37
method name (selector)
  adding colon when taking arguments to, 36–37
  methods indexed by, 65–66
methods
  adding to class, 21
  adding to NSString, 307–309

calling by sending objects messages, 6
conventions for naming, 22
declaring in new class, 50
declaring private, 309
misspelling when implementing delegates, 112
NSAtrributedString, 289
NSDictionary, 209
NSFont, 285–286
NSMutableDictionary, 209
NSNotificationCenter, 222–224
NSOpenGLView, 441
NSPasteboard, 296–298
NSResponder, 274
NSString, 289
NSUserDefaults, 210–211
NSView, 241–243
as public in Objective-C, 27
retain-count rules for, 76–77
taking arguments, 36–37
modal windows, 336–337
model classes, 129
Model-View-Controller design pattern
Cocoa and, 129–130
document architecture relating to, 163
in object-oriented programming, 129
modifierFlags method, NSEvent, 274
mouse events
autoscrolling, 268–269
composite images, 264–265
nextResponder, 414–415
NSEvent, 257–258
NSResponder, 257
rollovers, 282–283
view's coordinate system, 266–268
mouseDown: method
adding timer to autoscrolling, 328
coordinate system of views, 267
getting mouse events, 259
NSResponder, 257
mouseDragged: method
adding autoscrolling, 268–269
adding timer to autoscrolling, 328
coordinate system of views, 267
getting mouse events, 259
NSResponder, 257
mouseEntered event, rollovers, 283
mouseExited event, rollovers, 283
mouseMoved event, rollovers, 282–283
mouseUp: method
adding timer to autoscrolling, 328
coordinate system of views, 267
getting mouse events, 259
NSResponder, 257
multicore processors, and multithreading, 428
multithreading
time profiling, 431–435
faster performance and, 438–439
overview of, 427–428
race condition problem in, 428–429
simple Cocoa background threads, 429–431
using NSOperationQueue, 435–437
mutableCopy method, NSArray, 45
mutex locks, thread synchronization and, 438

N
naming conventions
delegates and notifications, 227
keys for names of defaults, 212
methods and instance variables, 22
navigation bar, 386–387
navigation controller, iOS
adding, 388–390
pushing view controllers, 393–395
New File menu item, creating classes, 20
new method, retain-count rules for ownership, 76
New Project, create new project, 12–14
nextKeyView, project with keyboard events, 276–278
nextResponder
and events, 414–415
pasteboards and nil-targeted actions, 300
NeXTSTEP, 1–3
NIB files
awakeFromNib called automatically, 30–31
chronology of applications, 32
localizing, 234–235
overview of, 17
NIB files and NSWindowController
adding panel to application, 194–196
menu item setup, 197
NSBundle, 204–205
overview of, 193
nil
ARC setting weak references to, 81–82
defined, 26
key-value coding and, 139–140

nil, *continued*
not adding to arrays, 46
representing nothingness in array with NSNull, 46
sending messages to, 41–42
nil-targeted actions, pasteboards and, 300–302
NO, 26
nonatomic properties, 125
notifications
delegates and, 227
handling upon arrival, 226
NSNotification, 222
NSNotificationCenter, 222–224
overview of, 221
passing between applications, 222
posting, 224–225
registering as observer, 225–226
userInfo dictionary and, 226–227
NS prefix, defined, 43
NSApplication object
chronology of applications, 32–33
defined, 20
suppressing creation of untitled documents, 215–216
NSApplicationMain()function, 15, 32
NSArray, 45, 212, 371
NSArrayController
add sorting, 140–141
binding view-based table views, 183–188
key-value coding and nil, 139–140
overview of, 129–130
rewriting RaiseMan without using, 142–144
sorting without, 141–142
NSAssert(), 61
NSAtrributedString
drawing text with attributes, 286–288
drawInRect: as category for, 309
formatters returning attributed strings, 350
methods for drawing onto view, 289
NSBezierPath, 247–250
NS_BLOCK_ASSERTIONS macro, 62, 462–464
NSBox
subviews of, 241–242
view hierarchy, 242
in view swapping, 398–399
NSBundle
creating string tables for localization, 236
using string table, 238–239
working with, 204–205
NSButton
creating views programmatically, 252–253
first responders, 302
keyboard events, 271
NSButtonCell, 253
overview of, 85–86
as subclass of NSControl, 83–84
as view, 241
view hierarchy, 242
NSButtonCell, 253
NSCAssert(), assertion checking in C, 62
NSCoder
decoding data, 162–163
encoding data, 160–161
overview of, 160
NSCoding protocol, 160–163
NSColor
creating with basic formatter, 341
registering defaults, 213
using NSColorList, 344–345
NSColorList, 341, 344–345
NSColorWell
as subclass of NSControl, 83
as view, 241
view hierarchy, 242
NSComparisonResult, NSDate, 55
NSCondition, thread synchronization, 438
NSControl
commonly used subclasses of, 85–89
delegate of, 347–348
inheriting from NSView, 253
overview of, 83–85
setting target programmatically, 96
NSControlKeyMask, 258
NSController, 129–130
NSData object
loading using NSDocument, 166–167
as property list class, 212
saving and NSKeyedArchiver, 167–168
saving using NSDocument, 165–166
user defaults, 213–214
Web services, 359–362, 363–364
NSDate
create, 54–55
current time, 30–31
formatter, 88
initializers with arguments, 56–58
methods, 54–55
as property list class, 212
NSDateComponents, 51–52, 70

NSDateFormatter
  attaching to text field, 88
  overview of, 339–340
NSDatePicker, 112, 181–183
NSDecimalNumber, 139–140
NSDictionary
  drawing text with attributes, 288
  NSPasteboard using, 297
  NSUserDefaults and, 211
  overview of, 208–209
  as property list class, 212
  userInfo dictionary and, 226–227
NSDistributedNotificationCenter, 222
NSDocument
  automatic document saving, 174
  defined, 163
  document architecture, 164
  document-based applications without undo, 175
  loading document, 166–167
  nil-targeted actions and, 301
  printing documents, 353
  saving document, 164–166
  undo manager for, 148
NSDocumentController, 163–164
NSDraggingInfo protocol, 315, 317–319
NSDragOperationCopy, 312–317, 319
NSDragOperationDelete, 312, 315
NSDragPboard, 296
NSEntityDescription, 189
NSError object, 165–167
NSEvent
  defined, 257
  keyboard events, 273–274
  mouse events, 257–258
  NSResponder methods, 273
NSFileHandleNotificationDataItem, 459
NSFileWrapper object, 165–167
NSFindPboard, 296
NSFont
  drawing text with attributes, 285–286
  NSAtrributedString, 287–288
NSFontManager, 293
NSFontPboard, 296
NSFormatters
  checking partial strings, 348–350
  defined, 88
  implementing, 345–347
  NSValueTransformer vs., 351
  overview of, 339–341
  returning attributed strings, 350
NSGeneralPasteboard, 296
NSGraphicsContext
  drawing with Quartz using, 441
  printing, 358
NSImage
  compositing image onto your view, 264–265
  coordinate system of views, 266–268
  representations of, 269
  using, 269
NSImageRep, 269
NSImageView
  binding, 186–187
  view-based table views, 181–184
NSInputManager, 293
NSInvocation, NSUndoManager, 145
NSKeyedArchiver, 167–168
NSKeyedUnarchiver, 168–169
NSLevelIndicator, view-based table views, 181–184
NSLock, thread synchronization, 438
NSLog() function
  defined, 40
NSManagedObject, 409–411
NSManagedObjectContext
  accessing for view swapping, 397
  creating, 178
  defined, 177
  how Core Data works, 189
  interface, 179–180
NSManagedObjectModel, 177–181, 189
NSMatrix
  working with cells, 253–254
NSMenuItem, 95
NSMutableArray
  in compositions, 48
  defined, 45
  instances, 35–37
  methods, 46
  sorting with, 141–142
  thread synchronization and, 438
NSMutableAtrributedString, 286–288
NSMutableDictionary
  overview of, 208–209
NSMutableString, 45
NSMutableURLRequest class, 361–363

NSNotification, 222
NSNotificationCenter, 221–224
NSNull, 46
NSNumber
  as immutable, 45
  key-value coding, 118
  key-value coding and nil, 139–140
  as property list class, 212
NSNumberFormatter, 340
NSObject
  creating class, 20–22
  header files and, 26–27
  init as designated initializer for, 57
  isa pointer and, 65–66
  as root of Objective-C class hierarchy, 43
NSObjectController, 129
NSOffState, buttons, 86
NSOnState, buttons, 86
NSOpenGLView
  working with OpenGL, 441
NSOpenPanel
  using, 263–264
  mediated file access and Powerbox, 468
NSOperationQueue
  overview of, 435
  thread synchronization, 437–438
NSOrderedAscending, 47
NSOrderedDescending, 47
NSPanel
  creating, 193–194
NSPasteboard
  adding cut, copy and paste, 298–300
  methods, 296–298
  overview of, 295
NSPasteboardItem
  lazy copying, 304–305
  creating custom UTIs, 303
  defined, 297
NSPasteboardReading protocol, 297, 299, 303
NSPasteboardWriting protocol, 297, 299, 303
NSPersistentDocument, 178, 189
NSPersistentStoreCoordinator, 189
NSPipe, NSTask, 453–454, 456–460
NSPoint
  coordinate system of views, 266–268
  drawing with NSBezierPath using, 248–250
  drawRect: method, 247
NSPrintOperation, 353–357
NSProgressIndicator, 321, 324–325
NSRange
  dealing with range of numbers, 286
  searching strings for substrings, 344–345
NSRect, 247–248
NSResponder
  keyboard events and, 273
NSRulerPboard, 296
NSRunAlertPanel() function, 229–230
NSRunApplication(), 16
NSSavePanel, 468
NSScrollView, 242, 250–252
NSSecureTextField, 88
NSSize, 247
NSSlider
  NSSliderCell, 253
  overview of, 86–87
  as subclass of NSControl, 83, 85
NSSortDescriptor objects, 141–142
NSSpeechSynthesizer
  delegate methods of, 100–104
NSSpeechSynthesizerDelegate protocol, 100–102, 106
NSSplitView, 242
NSSRunLoop, 328
NSString
  converting strings to other objects. *See* NSFormatters
  defined, 47
  as immutable, 45
  methods, 47
  methods for drawing onto view, 289
  NSMutableString, 45
  overview of, 40–41
  as property list class, 212
  using compare: to sort strings, 371
NSTableCellView
  binding view-based table views, 185
  creating view-based table views, 181
  defined, 254
NSTableDataSource, 104–107
NSTableView
  connections, 109
  dataSource of, 104–107
  view hierarchy, 241–243
NSTableViewDataSource methods, 368

NSTask
asynchronous reads, 456
overview of, 451
NSTextField
binding view-based table views, 187–188
creating view-based table views, 181
first responders, 271–274, 302
keyboard events, 271
NSTextFieldCell view, 253
overview of, 87–89
as subclass of NSControl, 83, 85
view hierarchy, 241–242
NSTextView
first responders, 302
inheritance diagram for NSControl, 85
keyboard events, 271
NSUndoManager and, 158
NSThread, 429–431
NSTimeInterval, NSDate, 54–55
NSTimer
NSSRunLoop used with, 328
overview of, 321–323
NSUndoManager
document-based applications without, 174
for edits, 153–155
how it works, 146–148, 189
key-value observing, 152–153
NSInvocation and, 145
overview of, 145
windows and, 158
NSURL class, 359
NSURLConnection
fetching data from Web service, 361–363
asynchronous, 377–381
NSURLRequest, 359, 361–363
NSUserDefaults, 207, 210–211
NSUserDefaultsController, 217
NSValueTransformer, 351
NSView
with cells, 253–254
custom views with, 241
drawing with NSBezierPath, 248–250
drawRect, 246–248
generating PDF data, 291–293
inheritance diagram for NSControl, 84–85
inheriting from NSResponder, 257
keyboard events. *See* keyboard events
Size Inspector, 244–246
starting drag, 312–313
NSViewController, 397, 399, 403
NSWindow
sheets. *See* sheets
becoming firstResponder, 271–272
initialfirstresponder outlet, 94
view hierarchy, 241–242
NSWindowController
defined, 163
document architecture, 167
loading NIB file without, 205
loading NIB files with. *See* NIB files and NSWindowController
pasteboards and nil-targeted actions, 301
NSWorkspace, 369
NSXMLDocument, 359–360
NSXMLNode, 359–360
NSXMLParser, 360, 363–366
NSXMLParserDelegate, 362, 364–366

O

Object Library, 17–18
object-oriented programming, Cocoa as, 5–6
object relationships, as focus of ARC, 81–82
objectAtIndex method, NSArray, 45
Objective-C
awakeFromNib method, 30–31
declaring classes in, 25
editing implementation file, 27–28
header file, 26–27
how messaging works, 65–66
instances, 35–37
Objective-C 2 features, 4
overview of, 35
single inheritance in, 26
static analyzer, 63–64
types and constants in, 26
Objective-C, creating own classes
implement description method, 52–53
initializers with arguments, 56–58
overview of, 48–49
initializers, 55–56
objects
archiving. *See* archiving
within blocks, 375–376
creating, sending messages to and destroying, 35–37
overview of, 5–6
retain count system, 69–70

observers
  defined, 221
  NSNotificationCenter methods, 222–224
one-to-one relationship in Core Data 407–408
Open menu item, NSDocument, 166–167
Open Recent menu item, NSDocument, 166–167
OpenGL
  Core Animation. *See* Core Animation
  overview of, 441
OpenOffice, NSAtrributedString file format, 288
OpenStep, 2–3
operators, in key paths, 126–127
opt-in garbage collector, Objective-C 2, 4
@optional, protocol, 173
ordered to-many relationships, key-value coding, 148–149
outlets
  defined, 21
  making connections, 23–25, 91–94
ownership, object, 76

P

pagination when printing, 353–357
panels
  adding to application, 194–196
  general windows vs., 193–194
parse method, NSXMLParser, 365–366
parse XML document, Web services, 359–360
partial strings, formatter that checks, 348–350
paste, 298–302. *See also* drag-and-drop
pasteboards
  lazy copying, 304–305
  nil-targeted actions and, 300–302
  NSPasteboard, 296–298
  overview of, 295–296
  UTIs and, 303
PDFs
  copying image on pasteboard as, 295
  getting your view to generate, 291–293
  printing, 353
  Quartz generating, 2
  using Cartesian coordinate system, 255
performance overhead
  autorelease message, 79
  garbage collector, 68
performDragOperation: message, 316, 318
performSelectorInBackground:withObject: method, 430–431, 435
performSelectorOnMainThead:withObject: waitUntilDone: method, 430–431
placeholder string, text fields, 87–89
plist
  dumping localized strings into, 239–240
  specifying application entitlements, 467
PNG image, copying data onto pasteboard as, 295
po (print-object)
  debug console, 60
  NSObject, 44
pointer
  nil value and, 139
  NSArray as list of, 45
  using contextInfo when starting sheet as, 335–336
Pop-up button, 397–399, 402–403
posting notifications, 224–227
Powerbox, and mediated file access, 468
#pragma, 259
Preferences panel, creating
  adding to application, 194–196
  user defaults. *See* user defaults
prepareForDragOperation: message, 315–318
prepareWithInvocationTarget: method, NSUndoManager, 146–147
preprocessor macros, 62, 462–464
print-object (po)
  debug console, 60
  NSObject, 44
printing
  drawing differently on screen, 358
  overview of, 353
  pagination when, 353–357
printOperationWithSettings:error: method, 353, 357
private methods, declaring, 309
programmatically creating custom views, 252
project directory, 13
Project Navigator, 27, 62
projects, creating new, 12–14
properties
  attributes of, 125–126
  eliminating code with, 124
  in NSManagedObjectModel, 177–179
@property, declaring properties, 125
property list classes, registering defaults, 213

protocols
  creating own, 173–174
  as lists of method declarations, 160
  NSCoding, 160–163
Python, 4

Q

Quartz, 2
QuartzCore framework, 418
Quit command, 33

R

race conditions
  problems in multithreading, 428–429
  thread synchronization and, 437–438
random function, 51
readFromData:ofType:error: method
  loading and NSKeyedArchiver, 168–169
readFromPasteboard: method, 299, 308
readFromURL:ofType:error: method, NSTask o, 453
reading and writing defaults, 210–214, 217–218
readonly attribute, properties, 125
readwrite attribute, properties, 125
receipt validation, Mac App Store, 469–470
Received actions panel, connections, 25
Redo stack, NSUndoManager, 146–147
registerForDraggedTypes: method, 315–316
registering
  defaults, 210–213
  as observer, 222–226
relationships
  Core Data. *See* Core Data relationships
  in NSManagedObjectModel, 178
Release build configurations
  blocking assertion checking, 62
  changing current build configuration to, 62
  creating, 464–466
  overview of, 461–462
release message
  autoreleasing objects, 73–76
  calling dealloc method, 72–73
  deallocating objects, 70–71
  decrementing retain count, 69
  enabling accessor methods for instance variables, 78–79
  retain-count rules, 76–77
removeObject method, NSMutableArray, 46
removeObjectAtIndex method, NSMutableArray, 46
resignFirstResponder method, 272, 273, 280
resources, Xcode tracking application, 12
responder chain, 300–301
respondsToSelector: method, delegates, 113–114
retain counts, 64, 81–82. *See also* manual reference counting
retain cycles, 68–69
retain message
  accessor methods for instance variables, 78–79
  incrementing retain count, 69
  memory management, 71–72
  retain-count rules, 76–77
Revert To Saved menu item, NSDocument, 166–167
Rich Text Format (RTF), 288
Rich Text Format with attachments (RTFD), 288
RoboCop, 99
rollovers, 282–283
RTF (Rich Text Format), 288
RTFD (Rich Text Format with attachments), 288
Ruby, 4
Run action
  changing current build configuration to, 62
  run application, 28–29
Run toolbar, 13

S

sandboxing, application
  mediated file access and Powerbox, 468
  overview of, 466
  specifying entitlements, 467–468
Save All menu item, NSDocument, 165–166
Save As menu item, NSDocument, 165–166
Save menu item, NSDocument, 165–166
saving
  automatic document, 174
  and NSKeyedArchiver, 167–168
  PDFs, 292
  using NSDocument, 164–166
Scheme Editor
  build configurations, 461–462
  changing build configuration, 62
  creating release build, 464–466

scroll view
  hierarachy, 242
  resizing with window, 250–252
security, and application sandboxing, 466–468
selector table, 65
self, writing initializers, 55–56
setAcceptsMouseMovedEvents: message, rollovers, 282
setEnabled message, buttons, 86
setFloatValue methods, sliders, 86
setImage: method, NSImage, 268
setNeedsDisplay: message, redrawing view, 246–248
setNeedsDisplayInRect: message, 248
set NilValueForKey: key-value coding, 140
setState message, buttons, 86
setString: method, drawing text with attributes, 290
setValue:forKey method
  bindings, 120
  key-value coding, 117
setWantslayer, Core Animation, 419–420
shadows, 293
shapes, CAShapeLayer, 425
Share button, Release build configuration, 464–465
sheets
  adding, 330
  code for, 334–335
  creating modal windows, 336–337
  defined, 329
  interface layout, 331–334
  NSApplication methods, 329
  outlets and actions for, 331
  using contextInfo, 335–336
showWindow, panels, 203
simple attributes, key-value coding for, 148–149
Size Inspector
  customizing views, 244–246
  resizing NSScrollView, 251–252
sliders. *See also* NSSlider
  binding value of, 119–120
  making keys observable, 121–123
  for sheets, 332–334
Smalltalk, 35
sort descriptors, 141–142
sortedArrayUsingSelector: method, 374
sortedArrayUsingFunction: context: method, 371
sortedArrayUsingSelector: method, 371
sorting
  add to NSArrayController, 140–141
  people, based on names, 142
  with sortedArrayUsingSelector:, 371
  using blocks, 372–373
  without NSArrayController, 141–142
source code display, Instruments, 433–434
source, drag, 312–315
split view, hierarachy, 242
state message, buttons, 86
static analyzer, 63–64
Step-Over button, debugger bar, 60
string tables
  creating, 237–238
  localization using, 236
  using, 238–239
stringByAppendingString method, NSString, 47
strings
  converting with formatters. *See* NSFormatters
  drawing strings and attributed, 289
  explicit ordering of tokens in format, 240
  NSAtrributedString, 286–288
  Objective-C vs. C, 40–41
  searching for substrings, 344–345
  translated, getting into XIBs with ibtool, 239–240
.strings extension, string table, 236
stringWithFormat, NSString, 75–76
strong attribute, properties, 125
strong references
  in ARC, 81–82
  in manual reference counting, 72
subclasses
  creating instance of view, 243–244
  “inherits from” vs. “uses” or “knows about” and, 48
  NSControl, 85–89
  NSImageRep, 269
substrings, searching strings for, 344–345
subviews, view hierarchy, 241–243
superview, Size Inspector, 244–246
symbols, Time Profiler, 432–433

synchronous connections
  multithreading in, 427–428
  NSURLConnection, 361–363
@synthesize
  accessor methods, 124–125
  connections in Interface Builder, 93
  as Objective-C keyword, 27
  property attributes, 125–126

T

tab view, hierararchy, 242–243
table selector, 65
table view. *See also* view-based table views
  NSTableView and its dataSource, 104–107
  setting background color, 216–217
  UITableViewController, 392–393
.tar files, 460
target/action
  debugging hints, 98
  defined, 21, 83
  making connections, 24–25
  NSMatrix, 254
  NSTimer, 323
  overview of, 83–85
  pasteboards and nil-targeted actions, 300–302
  set target programmatically, 96
target, defined, 83
targeted actions, pasteboards and nil-, 300–302
terminate: message, 33
text, drawing on layer with CATextLayer, 425
text, drawing with attributes
  bold and italics, 294
  getting view to generate PDF, 291–293
  giving shadows to letters, 293
  making letters appear, 289–291
  NSAtrributedString, 286–288
  NSFont, 285–286
  NSFontManager, 293
  overview of, 285
  strings and attributed strings, 289
text fields. *See also* NSTextField
  awakeFromNib and, 30–31
  create outlet connection for, 92–93
  creating basic formatter, 342–344
Text Table Cell View, 181–182
.tgz files, 460
Objective-C language, 5, 35
threads, 427–429
Time Profiler, 431–435
timeIntervalSinceDate method, NSDate, 54
timeIntervalSinceReferenceDate method,
    NSDate, 55
timers. *See* NSTimer
title
  changing button, 18
to-many relationships
  Core Data, 407
  key-value coding for, 148–152
tokens, format strings, 40, 240
Tool project, 12
tools, 3–4
toValue property, CABasicAnimation, 424
troubleshooting, common problems, 98
types
  in Objective-C, 26

U

UIKit, 383
UINavigationController, iOS, 390
UITableView, iOS, 391–393
UITableViewController, iOS , 391–393
UITableViewDataSource methods,
    UITableViewController, 392
UIViewController
  creating RootViewController subclass,
      386–387
  creating Web view, 395
  porting RanchForecast to iOS, 383–385
UIWebView, iOS, 395
unarchiveObjectWithData: method, loading
    and NSKeyedArchiver, 168–169
unarchiving, 159, 168–169
unbind method, 127
undo manager. *See* NSUndoManager
Undo stack, NSUndoManager, 146–147
Unicode (UTF-8) file encoding, 238
universal type identifiers. *See* UTIs (universal type
    identifiers)
Unix processes, 1–3
  NSTask and, 451
unordered to-many relationships, key-value
    coding, 149–150
untitled documents, suppressing creation of,
    215–216
updateChangeCount method, NSDocument, 174
uppercaseString, NSString, 75–76

URLs
  adding WebView to application, 369–370
  opening for Web service, 368–369
user defaults
  enabling user to edit, 213–215
  NSUserDefaults, 210–211
  NSUserDefaultsController, 217
  overview of, 207–208
  precedence of types, 211
  reading and writing from command line, 217–218
  setting, 212–213
user interface. *See* interface layout
userInfo dictionary, 226–227
"uses", "inherits from" vs., 48
UTExportedTypeDeclarations key, exported UTIs, 175
UTF-8 (Unicode) file encoding, 238
utility area, Interface Builder, 16
UTIs (universal type identifiers)
  configuring exported, 170–171
  customizing, 303
  pasteboards and, 297–300, 303
  understanding, 175

V

value transformers, 351
valueForKey method
  bindings, 120
  key-value coding, 117
  key-value observing, 122
variables
  inside objects, 5
  reading and setting with accessor methods, 50
  viewing in debugger, 59–60
view-based table views
  bindings, 183–188
  vs. cell-based, 191
view classes, object-oriented programming, 129
view controllers. *See* view swapping
view controllers, in iOS
  overview of, 383–385
  pushing, 393–395
  UITableViewController, 391–393
view swapping
  creating view controllers and XIB files, 400–401
  overview of, 397–398
  resizing window, 403–405
viewDidMoveToWindow, rollovers, 283
views
  drag-and-drop to. *See* drag-and-drop
  images and mouse events. *See* images and mouse events
  layers vs., 417
views, custom
  cells, 253–254
  creating programmatically, 252
  drawing with NZBezierPath, 248–250
  flipping with isFlipped, 255
  keyboard events, 274
  overview of, 241
  that draw themselves, 243–248
  view hierarchy, 241–243

W

weak attribute, properties, 125
_weak variable, blocks, 375–376
__weak qualifier, ARC, 82
weak references
  ARC, 81–82
  manual reference counting, 72
  platforms not supporting ARC, 82
Web services
  NSURLConnection class, 361–363
  overview of, 359–360
Web view, 369–372, 395
Window Inspector, setting hidesOnDeactivate, 194
window server, 2, 33
windowControllerDidLoadNib: method, 166–169
windowDidLoad method, 203, 214–215
windows
  adding sheets to. *See* sheets
  collection of views for, 241
  firstResponder of key, 271
  iOS app, 383–384
  NSUndoManager and, 158
  NSWindowController, 167
  panels vs.main, 193–194
Word, NSAtrributedString file format, 288
write code, keyboard events, 278–282
writeObjects method, NSPasteboard, 297
writeToPasteboard: method, 298

X

X window server, 2
Xcode
  creating new project, 12–14
  getting started, 11–12
  Interface Builder editor, 4
  iPhone and iPad apps using, 383
  overview of, 4
XIB files
  creating `NSFormatters`, 342–344
  defined, 15
  Interface Builder, 17
  overview of, 17
  translating strings, 239–240
  view swapping, 400–401, 403–405
XML parsing
  Web services, 359–360

Y

YES
  defined, 26
  `NSArray`, 45
  `NSObject`, 44

Z

ZIP archives
  listing tar files with .tar and .tgz vs., 460
  packaging application for download, 466
zombies, debugging hints, 98

This page intentionally left blank