Richard Wentk

Cocoa®

www.**wileydevreference**.com

Cocoa®

Richard Wentk

Wiley Publishing, Inc.

Cocoa®

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2010 by Wiley Publishing, Inc., Indianapolis, Indiana

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-49589-6

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, 201-748-6011, fax 201-748-6008, or online at http://www.wiley.com/go/permissions.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services or to obtain technical support, please contact our Customer Care Department within the U.S. at (877) 762-2974, outside the U.S. at (317) 572-3993 or fax (317) 572-4002.

Library of Congress Control Number: 2010935569

Trademarks: Wiley and the Wiley logo are registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Cocoa is a registered trademark of Apple, Inc. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

To Bea, for the inspiration.

Nam et ipsa scientia potestas est.

About the Author

With more than ten years of experience as a developer and more than fifteen years in publishing, Richard Wentk is one of Great Britain's most reliable technology writers. He covers Apple products and developments for *Macworld* and *MacFormat* magazines and also writes about technology, creativity, and business strategy for magazines such as *Computer Arts* and *Future Music*. As a trainer and a former professional Apple developer returning to development on the iPhone and OS X, he is uniquely able to clarify the key points of the development process, explain how to avoid pitfalls and bear traps, and emphasize key benefits and creative possibilities. He lives online but also has a home in Wiltshire, England. For details of apps and other book projects, visit `www.zettaboom.com`.

Credits

Acquisitions Editor
Aaron Black

Project Editor
Martin V. Minner

Technical Editor
Benjamin Schupak

Copy Editor
Lauren Kennedy

Editorial Director
Robyn Siesky

Editorial Manager
Rosemarie Graham

Business Manager
Amy Knies

Senior Marketing Manager
Sandy Smith

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Senior Project Coordinator
Kristie Rees

Graphics and Production Specialists
Joyce Haughey
Jennifer Henry

Quality Control Technician
John Greenough

Proofreading
Laura Bowman

Indexing
BIM Indexing & Proofreading Services

Media Development Project Manager
Laura Moss

Media Development Assistant Project Manager
Jenny Swisher

Media Development Associate Producer
Shawn Patrick

Contents

Preface xv

Acknowledgments xvi

Introduction xvii

Part I: Getting Started 1

Chapter 1: Introducing Cocoa 3

Introducing Cocoa 3
Understanding Cocoa's history 3
Moving from NeXTStep to Cocoa 7
Profiting from Cocoa 9
Profiting from the iPhone 12
Developing for fun 14
Introducing Xcode and the Apple Developer Programs 15
Working with Xcode and Interface Builder 16
Working with Safari 17
Summary 18

Chapter 2: Think Cocoa! 19

Designing for Cocoa 19
Understanding Aqua 20
Using Aqua with Cocoa 21
Creating Cocoa Applications 22
Understanding layers and frameworks 22
Developing features across layers 27
Moving to Cocoa and Objective-C from Other Platforms 31
Working with Objective-C objects and messages 31
Managing data in Cocoa and Objective-C 38
Exploring other Cocoa features 45
Comparing Cocoa to other platforms 48
Summary 55

Chapter 3: Introducing the Cocoa and OS X Documentation 57

Getting Started with the Documentation 59
Understanding resource types 61
Understanding Topics 69
Using the Documentation 71
Sorting the documentation 71
Working with source code 72
Summary 72

Chapter 4: Getting Started with Xcode 75

Getting Ready for Xcode 75
Registering as a developer 77
Joining the Mac Developer and iPhone Developer programs 80
Installing Xcode 82
Creating a New OS X Project 84
Exploring Xcode's Windows 90
Understanding Groups & Files 90
Selecting items for editing 92
Customizing the toolbar 93
Summary 94

Chapter 5: Introducing Classes and Objects in Objective-C 95

Understanding Objects 95
Understanding classes 97
Designing objects 100
Creating classes 107
Defining a class interface 107
Defining accessors: setters and getters 109
Using self 110
Defining a class implementation 110
Defining public properties 111
Defining public methods 111
Using Objects in Objective-C 113
Summary 114

Chapter 6: Getting Started With Classes and Messages in Application Design 115

Understanding the Cocoa Development Process 115
Understanding Applications 116
Exploring standard application elements 118
Introducing the application delegate 118
Discovering Object Methods and Properties 121
Finding and using class references 121
Introducing Code Sense 126
Working with multiple classes 128
Receiving messages from OS X with a delegate 134
Receiving messages from OS X with NSResponder 142
Subclassing NSWindow 144
Creating a category on NSWindow 149
Summary 150

Chapter 7: Introducing Interface Builder 151

Introducing Nib Files 151
Loading objects from nib files 153
Editing nib files 154
Getting Started with Interface Builder 154
Introducing IB's windows 155
Introducing First Responder and File's Owner 168
Setting Classes and Subclasses 170
Summary 172

Chapter 8: Building an Application with Interface Builder 173

Designing a Project in Interface Builder 173
Introducing the Interface Builder workflow 174
Adding objects to a nib 176
Understanding links, outlets, and actions 184
Creating links in Interface Builder 189
Using NSTimer to create a simple seconds counter 195
Using Advanced UI Techniques 199
Using loose typing and (id) sender 200
Placing outlets and actions 202
Summary 205

Part II: Going Deeper 207

Chapter 9: Using Cocoa Design Patterns and Advanced Messaging 209

Understanding Model-View-Controller 209
Using MVC with Cocoa controller objects 212
Creating custom controllers 213
Defining the data model 214
Understanding Target-Action 214
Defining selectors 215
Using selectors in code 216
Understanding the limitations of selectors 216
Defining selectors in Interface Builder 217
Creating an example application 219
Other applications of selectors 221
Using Key-Value Coding 222
"Objectifying" values 222
Using Key-Value Observing 224
Making assignments KVO compliant 226
Using KVO 227

Using Notifications 228
Posting notifications 230
Using notifications and delegates 230
Handling Errors and Exceptions 232
Using NSError 232
Handling errors with NSException 233
Summary 234

Chapter 10: Working with Files, URLs, and Web Data 235

Creating and Using File Paths 236
Creating paths with NSString 236
Getting the application bundle path 236
Finding other standard directories 237
Using autocompletion 237
Using paths 238
Using file handles 238
Using the File Manager 239
Creating and Using URLs 240
Understanding paths and references 240
Using URLs to read and write data 240
Using Open and Save Panes 241
Using Web APIs 247
Getting started with bit.ly 248
Using the bit.ly API 251
Creating XML requests 255
Creating asynchronous Web requests 258
Using Cocoa's XML classes 260
Using WebView 261
Summary 264

Chapter 11: Using Timers, Threads, and Blocks 267

Using NSTimer 267
Using performSelector: 268
Implementing a pause method 269
Running the selector in a separate thread 269
Messaging across threads 269
Working with NSThread 270
Pausing a thread 270
Managing thread memory 271
Handling UI and thread interactions 271
Using NSOperation 271
Creating an NSOperation object 272
Using NSOperationQueue 274

Getting Started with Blocks 277
Understanding block syntax 277
Using NSBlockOperation 279
Passing parameters to NSBlockOperation 280
Introducing Grand Central Dispatch 281
Using NSTask 281
Summary 283

Chapter 12: Managing Data and Memory in Cocoa 285

Introducing Data Collection Objects 286
Using objects, keys, and values 287
Implementing Key-Value Observing 288
Using NSValue and NSNumber 290
Using NSArray 291
Using NSDictionary 296
Using NSSet and NSMutableSet 297
Enumerating items 297
Archiving and de-archiving collection objects 299
Using NSCoder and NSData 300
Introducing archiving and coding 300
Creating a class with NSCoder 302
Archiving and de-archiving an object 304
Managing Memory 307
Using garbage collection 307
Implementing manual memory management 308
Summary 311

Chapter 13: Using Preferences and Bindings 313

Understanding Bindings 313
Getting started with bindings 314
Using bindings to manage interactivity 323
Using KVO to manage bindings 326
Using formatters 328
Using Bindings with Controllers 330
Adding a controller object 332
Setting up the controller's data source 334
Reading data from the controller into a view 335
Implementing Preferences with Bindings 340
Understanding preferences 341
Creating an application with preferences 343
Creating and Using Value Transformers 346
Summary 350

Chapter 14: Using Core Data 351

Creating a Core Data Application Visually 352
Adding an entity 354
Adding properties 355
Creating relationships 356
Generating a user interface 359
Building the application 361
Exploring and Extending a Core Data Application 363
Understanding Core Data's objects and programming model 365
Displaying search results 370
Summary 374

Chapter 15: Working with Text and Documents 375

Using NSString 375
Using NSRange 376
Working with encodings 377
Using NSAttributedString 378
Drawing and using attributed strings 380
Creating Nanopad: A Rich Text Editor 381
Using NSFontManager 381
Saving and loading rich text 383
Implementing the Open Recent menu 384
Creating, Saving, and Loading Documents 385
Creating a default nib file 387
Setting document types 388
Implementing save and open code 391
Printing documents 393
Using NSUndoManager 394
Localizing Applications 395
Summary 400

Part III: Expanding the Possibilities 401

Chapter 16: Managing Views and Creating 2D Graphics 403

Understanding Windows and Views 404
Understanding the view hierarchy 406
Subclassing the root view 406
Adding and removing views from the view hierarchy 409
Handling mouse events in views 414
Understanding the Cocoa Graphics System 415
Understanding and defining basic geometry 416
Creating shapes and colors in drawRect: 418
Creating a simple project: MultiBezier 428

Using CoreImage Filters 429
Adding CoreImage effects in Interface Builder 430
Setting up filters for processing 432
Applying filters to an image 436
Summary 438

Chapter 17: Creating Animations and 3D Graphics 439

Using Direct Property Animation 440
Creating a timer for animation 441
Creating property animation code 442
Using drawRect: 443
Using Animators 443
Creating a simple proxy animation 443
Setting the animation duration 446
Customizing the animation object 446
Creating and using animation paths 449
Creating Animations with CALayer 452
Using layers for animation 452
Creating an animatable filter 453
Animating the filter 456
Using OpenGL 458
Introducing OpenGL 459
Creating an OpenGL animation 459
Controlling an OpenGL animation 462
Summary 464

Chapter 18: Debugging, Optimizing, and Managing Code 465

Using the Console and NSLog 466
Getting started with NSLog 466
Selectively enabling NSLog 471
Debugging with Breakpoints and the Debugger Window 473
Enabling debugging 474
Using the Debugger window 478
Using Instruments 481
Using Shark 485
Managing Code with Snapshots and Source Control 486
Copying projects and creating snapshot versions manually 487
Using Snapshots 489
Using SVN source control 491
Summary 492

Chapter 19: Developing for the iPhone and iPad 493

Introducing the iPhone, iPod touch, and iPad 494
Comparing iOS and OS X applications 495
Understanding the mobile app business model 498
Moving to iOS from OS X 502
Getting started with the iPhone SDK 502
Understanding iOS app design goals 503
Understanding key iOS coding differences 505
Considering iOS and hardware compatibility 505
Understanding iOS Views and UI Design 505
Working with Windows and views on the iPhone 505
Managing orientation 509
Adding navigation and control features 511
Handling touch events 514
Working with windows and views on the iPad 515
Developing for iOS in Xcode 516
Using the Xcode Simulator 516
Introducing the Xcode templates 517
Building a Simple Application 524
Adding view controller subclasses 526
Implementing the view controllers 526
Creating views 528
Handling events with protocol messaging 532
Creating an animated view swap 534
Selling in the App Store 536
Understanding certificates, provisioning profiles, and permissions 538
Packaging an app for the App Store 541
Uploading an app to the App Store 542
Summary 543

Part IV: Appendixes 545

Appendix A: Building Dashboard Widgets 547

Appendix B: Maximizing Productivity and Avoiding Errors 565

Index 575

Preface

When I started developing for the iPhone after a fifteen-year break from software, my first thought was: *What is going on here?* I'd written machine code for Macs and had some experience with earlier versions of Mac OS. It soon became obvious that Cocoa Touch was doing clever things behind the scenes, and that my apps were supposed to be exchanging information with those clever things.

Unfortunately, neither the official documentation nor unofficial sources of help were making it clear what those things were.

With enough persistence, it's possible for almost any developer to reverse-engineer the documentation and answer the "What is going on here?" question for himself or herself. But it's more productive to have that information before starting out. So my first goal for this book is to equip you, as a developer, with the key concepts you need to build Cocoa projects efficiently and productively.

Understanding Cocoa means more than being able to name-check concepts like delegation and Model-View-Controller; it means learning how Cocoa applies these concepts, how they influence the design of Cocoa's classes, and how your code can leverage the features built into Cocoa to simplify projects and minimize development time. In short, it means discovering how to *think* Cocoa. New features will begin to feel intuitive once you understand the reasoning behind them.

My second goal for the book is to give readers the skills they need to answer Cocoa questions for themselves, without handholding. OS X is vast and complex, and a full printed guide of every feature would have to be delivered on a truck. Books always sell better when readers can pick them up and take them home without stalling traffic, so this book doesn't try to detail every Cocoa feature. It also doesn't try to build complex sample projects that are unlikely to match your specific needs. Instead, it gives you the skills you need to find answers to questions for yourself, using the official documentation and other sources of insight.

One feature you won't find in this book is cheerleading. Like any other development environment, Cocoa is a mix of excellence and unpredictability. Cocoa's best features are almost supernaturally productive and take you where you want to go with almost no code at all. Other elements offer a more scenic journey through less intuitive class relationships. Instead of a sales pitch, this book gives you a guided tour of the highlights but also warns you about some of the more dangerous parts of town.

Finally, software is as much an art as a science. Art is about creating captivating, enjoyable, and colorful experiences for an audience. In common with the Apple ethic, this book is deliberately less formal and more creative than a pure software reference. You'll find the rules here. And sometimes you'll also find suggestions for breaking the rules.

Every author tries to make his or her books as helpful as possible. Comments and feedback are welcome at `cocoadr@zettaboom.com`.

Acknowledgments

Books don't write themselves — not yet, anyway. Until operating systems become self-documenting, writing a book continues to be a team effort.

I'd like to thank acquisitions editor Aaron Black for enthusiastically suggesting the project and project editor Marty Minner for his support and for taking the manuscript and producing a book from it. Sincere thanks are also due to the rest of the team at Wiley for their hard work behind the scenes.

Software development has become a communal activity, and particular appreciation is due to the countless bloggers, experimenters, developers, and problem-solvers on the Web whose generosity and creativity have made so much possible in so many ways.

Finally — love as always for Team HGA. I couldn't have written it without you.

Introduction

This book is about developing Cocoa projects for OS X using the Xcode SDK. The theoretical elements of Cocoa are similar to those in Cocoa Touch and apply equally to both OS X and iOS. The more practical elements were written to describe OS X but with significant overlap with the equivalent features in iOS.

You'll find this book useful if you're a newcomer to Cocoa at the beginner or intermediate level and have experience with C/C++/C#, Java, Flash, Python, or a Web language such as PHP. If you're ambitious and feel up to a challenge, you can start with no experience at all. If you do, you'll find it helpful to use *Objective-C* (Wiley, 2010) as a companion title.

Cocoa isn't a synonym for OS X, and for practical reasons this book says little about the low-level Mach/POSIX features that underpin OS X. It mentions some of the C-level frameworks that Cocoa is built on but doesn't detail them, although it does give you enough information to explore them for yourself if you choose to.

Chapter 1 is an introduction to the history of Cocoa and OS X and explains how Cocoa evolved from Smalltalk and from the Objective-C development environment introduced by NeXT in the late 1980s. It also includes some strategic hints about the OS X and iOS application markets and how to research the current state of both so that you can target your applications for maximum return.

Chapter 2 is an informal introduction to the features that make Cocoa unique. Whether you're starting programming from scratch, or have a background in some other environment, this is one of the most critical chapters in the book. Reading it will save you time later.

Chapter 3 is a guide to the Apple documentation. It may not be obvious why this needs a guide, but Apple has structured the documentation in specific ways, and you'll progress more quickly and with less effort if you understand what this means in practice. Understanding and using the documentation is a key skill. Don't skip this chapter, even if you already have experience in other environments.

Chapter 4 explains how to join Apple's Developer Programs, and how to download and install Xcode. It also introduces the key features of Xcode 3.2.3, including the windows, menu items, and customizable toolbar. This chapter explains how to create a new sample project — an essential skill that's used repeatedly later in the book.

Chapter 5 introduces objects and classes and describes how they're implemented in Objective-C. If you have experience in other object-oriented environments, you'll need this chapter to reorient yourself to Objective-C. If you haven't, you'll find an explanation of object-oriented development that's a fundamental requirement for understanding Cocoa.

Chapter 6 explores objects in Cocoa in a more hands-on way, with very simple projects that illustrate how to use objects and their features in real Cocoa applications.

Chapter 7 introduces the key features of Interface Builder and explains how you can use IB to build complete applications, because IB isn't just for interfaces.

Chapter 8 demonstrates how to use IB to build a working application with a custom interface assembled using Cocoa library objects and how to connect a UI created in IB to code written in Xcode. This is another essential chapter. You'll need this information to build Cocoa successful applications.

Chapter 9 introduces some of the standard Cocoa design patterns and their supporting features, including target-action, Model-View-Controller, and selectors. It also looks more closely at Cocoa key-value technologies such as Key-Value Coding and Key-Value Observing and explains how to work with them effectively.

Chapter 10 introduces the Cocoa file interface and explains how it's built into many Cocoa objects, making a file manager unnecessary. For completeness, this chapter also introduces the file manager and explains how to add open and save panes to an application.

Chapter 11 explains how to manage timing, threads, and tasks in Cocoa. It also introduces the new block syntax, which is slated to replace delegation and other design patterns in future versions of OS X.

Chapter 12 introduces Cocoa's data collection classes, including `NSArray`, `NSDictionary`, and `NSSet`, and explores some of their features. It explains how to use `NSCoder` to serialize data when saving it or reloading it and introduces the essentials of both manual memory management and automated garbage collection.

Chapter 13 explores bindings, which are often seen as one of Cocoa's more challenging features but which are explained here in an unusually straightforward and practical way.

Chapter 14 follows from the previous chapter with an introduction to Core Data. It explains how to build a working card index application with no code at all and also how to customize it to make it more useful and flexible.

Chapter 15 introduces Cocoa's attributed — styled — text features and explains how to create applications with multiple document windows. You'll also find information about printing, undoing, and localizing text for foreign markets.

Chapter 16 explains how to create 2D graphics, using Cocoa's path, fill, and stroke features and also gives a low-level example of creating effects with Cocoa's Core Image filters.

Chapter 17 expands on the techniques of the previous chapter and demonstrates various animation techniques, including a simplified but animated Core Image filter. You can also find an introduction to OpenGL in Cocoa, with a sample animated teapot application.

Chapter 18 introduces various tools and strategies for debugging and profiling code and optimizing performance.

Chapter 19 is about developing for iOS. It introduces the key differences between Cocoa and Cocoa Touch, explains how to use the iOS simulator and how to get started with development on real hardware, and also explores some of the commercial opportunities offered by the iPhone and iPad.

Appendix A is about building dashboard widgets, which use JavaScript instead of Objective-C and are a quick and easy way to get started with Mac development.

Appendix B lists some of the common errors that appear in Cocoa code and introduces some possible bug-busting strategies.

Code appears `in a monospaced font`. Items you type appear **in bold.**

Projects were developed with Xcode 3.2.3 on OS X 10.6.3. Supporting code is available on the book's Web site at `www.wiley.com/go/cocoadevref`. See the readme there for the most recent system and software requirements. Code is supplied as is with no warranty and can be used in both commercial and private Cocoa projects but may not be sold or repackaged as tutorial material.

Getting Started

In This Part

Chapter 1
Introducing Cocoa

Chapter 2
Think Cocoa!

Chapter 3
Introducing the Cocoa and OS Documentation

Chapter 4
Getting Started with Xcode

Chapter 5
Introducing Classes and Objects in Objective-C

Chapter 6
Getting Started With Classes and Messages in Application Design

Chapter 7
Introducing Interface Builder

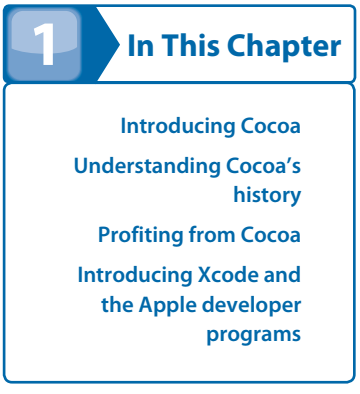
Chapter 8
Building an Application with Interface Builder

Introducing Cocoa

Apple's Cocoa technology is one of computing's success stories. When OS X 10.0 was released in 2001, it immediately revolutionized the look and feel of desktop applications. Since then, other operating systems have borrowed freely from Cocoa's innovations. Apple has continued to innovate with the iPhone and iPad, introducing Cocoa Touch for mobile devices. Cocoa Touch offers a simplified and more tactile user experience, and is the first popular and successful attempt to move beyond a traditional window, mouse, and menu interface. Future versions of Cocoa on the Mac are likely to blend the iPhone's tactile technology with the sophisticated data handling, 64-bit memory management, and rich user interface options that are already available to Cocoa developers. Cocoa is widely used in Apple's own projects, and it determines the look and feel of an application such as Aperture, shown in Figure 1.1.

Introducing Cocoa

Cocoa is the collection of libraries and design principles used to build skeleton Mac applications, create and display a user interface, and manage data. Cocoa is also a design philosophy based on unique ideas about application design and development that you can find throughout the rest of this book. You don't need to understand Cocoa's history to use the Cocoa libraries, but their features may be easier to work with if you do.



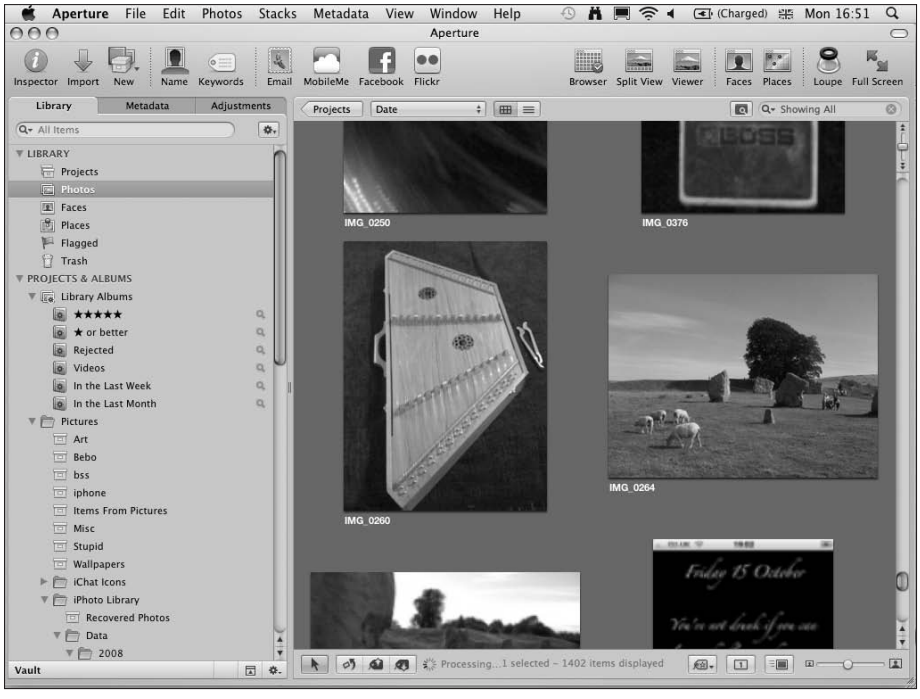
Understanding Cocoa's history

Cocoa's origins can be traced to the mid-1970s and are closely tied to the history of the Objective-C programming language. Cocoa and Objective-C are used at different levels. Cocoa is a code library and a set of interface and development guidelines. Objective-C is the language that implements them.

Cocoa is now available for other languages, including JavaScript, Python, and Ruby on Rails, but most Cocoa developers continue to work in Objective-C because its syntax and features are a natural fit for Cocoa projects.

Figure 1.1

Apple's Aperture application uses Cocoa technology and follows Apple's user interface design guidelines. Although Cocoa objects implement the interface, they don't enforce a standard look and feel.



Objective-C, developed by Brad Cox and Tom Love when they worked at ITT Corporation in the early 1980s, began as a mix of C and features copied from the Smalltalk experimental language. Smalltalk had been created — originally as a bet — by Alan Kay at the Xerox Palo Alto Research Center (PARC). PARC's famous graphical user interface (GUI) experiments inspired much of the visual design of both Mac OS and Windows. Smalltalk influenced those experiments by implementing a development environment in which independent objects communicated by sending and receiving messages.

At a time when most software was still *procedural* — it started at the beginning of a computer run and continued to the end, with occasional branches and subroutine calls — Smalltalk's model suggested a new and less rigid approach to software development. It enabled programmers to build applications from a library of "copy-able" but distinct interactive parts, connected by a messaging system that made the parts responsive and controllable.

A windowed GUI is difficult to manage in a procedural environment. In a Smalltalk environment, windows and icons become objects with properties — size, position, graphic contents,

and so on — that can be remotely controlled by messages. When a window receives a message, it not only stores the value, but also it can automatically redraw itself. Messaging makes it possible for objects to control each other remotely. Objects can update themselves or trigger behaviors in other objects because changing an object's properties can trigger a much more complex response.

Objects are *opaque*; that is, an object's internal code can be hidden and the object can be shared with other developers who see an *interface* — a set of properties and behaviors they can access remotely — but they don't need to think about the details of the *implementation* code behind these features. This enhances security and simplifies application design. Developers can use objects as functional building blocks without being distracted by the code that implements them.

Because objects can send and receive messages and respond to them in programmable ways, they're more powerful than conventional data structures and functions. Developers not only find objects simpler to use, but they also find it easier to invent new and more complex kinds of interactions. Updating a single property in a single object can automatically trigger a cascade of responses across an application.

From Smalltalk to Objective-C

Although Smalltalk's ideas were powerful, its syntax was cumbersome and it never evolved into a mainstream language. It is only rarely used now by professional developers, though it continues to be available in hobby projects. For example, experimenters can explore the Smoaktalk interpreter shown in Figure 1.2, which runs on the Web as a Flash application at `www.smoaktalk.com/st/071808`.

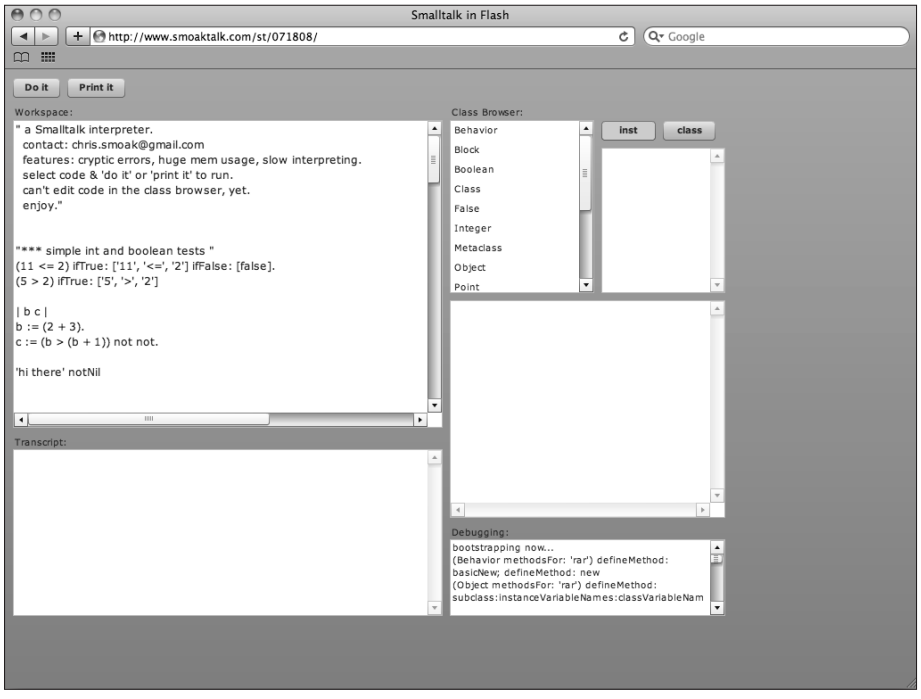
Smalltalk's influence is evident in other languages, including Ruby, Perl, and Objective-C. Objective-C is closest in spirit and implementation to the original Smalltalk design ideals. It blends them with the features of standard C to create a very powerful and productive environment. Objective-C can run C code without changes, but it adds features and concepts imported from Smalltalk. Specifically it supports objects and messaging using some of Smalltalk's syntax and conventions.

From Objective-C to NeXTStep

In 1988 Steve Jobs left Apple and started a new company called NeXT. NeXT licensed a version of Objective-C from StepStone, the owners of the language, and began creating its own compiler and libraries. NeXT's computing hardware products were widely acclaimed, but because they were too expensive for mainstream users, they sold poorly. NeXT reluctantly dropped out of hardware development and concentrated on developing the compiler, libraries, and SDK as a marketable product that could be licensed to potential users. The libraries, known as NeXTStep, included data management features. They also supported a sophisticated graphical and user interface (UI) environment based on the object-message development idiom inherited from Smalltalk.

Figure 1.2

Smoaktalk on the Web re-creates a Smalltalk interpreter inside a Flash application. You can use it to experiment with Smalltalk's features and syntax.



As NeXTStep became more popular and influential, it evolved into OpenStep, an open industry standard based on NeXTStep's features and design goals. The NeXT desktop became a benchmark for innovative interface development, supporting arbitrary fonts using Display PostScript, at a time when most PCs still displayed generic blocky text.

NOTE

Arguably, OpenStep narrowly missed out on the chance to compete with Microsoft Windows 3.0. In the early 1990s, OpenStep was an obviously superior operating system, and was already being converted to run Intel x86 machines from its original Motorola 680x0 platform. Unfortunately, OpenStep shipped later than Windows, handing Microsoft an early-adopter advantage. Had it shipped earlier and been promoted more aggressively, it might well have overshadowed Windows and given the developer community a worthy alternative platform.

NeXTStep and the Web

NeXTStep may have been directly responsible for the World Wide Web. The original hypertext browser and server system were developed by Tim Berners-Lee on a NeXT system at the European CERN particle accelerator. The Web isn't truly object-oriented or message-based, and HTML remains a simple mark-up language and not a programming language. But hints of Objective-C and NeXTStep are visible in the way that links and HTML fields are specified as objects with specific subproperties.

NeXTStep and Cocoa have certainly been influenced by the Web. Many Cocoa data objects make no distinction between local and remote data. Where most operating systems assume that all data is on a local disk or on a network, Cocoa enables data to be specified and downloaded with a URL. Some Cocoa objects use URLs for all data. Local file paths must be converted into a URL before the data can be loaded.

Moving from NeXTStep to Cocoa

In 1996 Steve Jobs returned to Apple, bringing NeXTStep with him. In the preceding years Apple had tried and failed to create a successor to Mac OS 9. Buying NeXTStep and re-hiring its creator was an obvious solution. After numerous false starts, Apple eventually released OS X 10.0 in 2001. Codenamed Cheetah, OS X 10.0 was built from a blend of OpenStep features and existing OS 9 code. Apple's new interpretation of NeXTStep was named Cocoa. Internally, Cocoa objects still use the letters NS as a prefix — for example, Cocoa's window object is called `NSWindow`.

Although OS X 10.0 was famously unstable — Apple released a free update to OS X 10.1 almost immediately — the NeXTStep-inspired look and feel of OS X was an immediate success with users and developers. Apple aggressively promoted OS X development by offering a free SDK. Since then Cocoa has developed further. In 2005 Apple dropped support for G series processors and moved OS to Intel Mac technology. Currently Cocoa and OS X are moving toward full 64-bit support with touch-based user interface extensions. But the SDK continues to be free. A version is bundled with all Macs, while the most recent update can be downloaded from Apple's Web site.

CROSS-REF

For more information about the Xcode SDK, see Chapters 3 and 4.

Recent versions of Cocoa remain compatible with NeXTStep code, which often compiles and runs on OS X with minimal changes. But Cocoa is no longer identical to OpenStep or NeXTStep. With each OS X update, new features are added and old features are changed or removed. Figure 1.3 shows Snow Leopard, the most recent version of OS X and Cocoa. Internally the NeXTStep legacy remains, both in the code and in the Cocoa design philosophy that influenced

the OS X user experience. Cocoa remains entirely object-oriented, and Cocoa code still uses a recognizable version of the messaging and object model that originated in Smalltalk in the late 1970s.

Figure 1.3

The current Snow Leopard is based on Cocoa technology, which remains similar to the technology used in OS X 10.0. Windows are rendered in a lightly textured gray finish with subtle 3D effects. The look is set by the OS and can't be customized within Cocoa.

Today Cocoa is the most popular framework for application development. Many Mac applications use other OS X libraries to implement media features and manage application data. But these mixed applications are still considered "Cocoa applications," as long as they create a Cocoa look and feel, and are organized around interacting objects and events.

NOTE

This book concentrates on Cocoa, but introduces key elements from other libraries. Even though technically they are not included in Cocoa, it's difficult to develop Cocoa applications without being familiar with them.

Profiting from Cocoa

Developers can learn Cocoa for both fun and profit. The Mac application market is very much smaller than the Windows market, but is also much less saturated, making it potentially easier to reach. Mac market share typically oscillates between 3 and 7 percent of total desktop and laptop sales, and is currently on an upswing because of the influence of the iPhone. Web statistics suggest that around 5 percent of pages are served to users of Safari or the Mac version of Firefox.

Sales of Windows PCs are partly determined by bulk corporate purchases, and this can distort user statistics. In reality, Macs continue to be a disproportionally popular choice for domestic and small business buyers. Even so, the market is limited and prospects for volume sales of niche applications are not good.

The Mac is also a relatively poor choice for games development. Macs are more expensive than performance PCs and more difficult to customize, so high-profile game developers have concentrated on the PC market, but there are exceptions outside of the prestigious high-performance games market. Simple Flash games are popular on the Web, and some developers have done well from straight PC and Mac ports, using Web gaming to market the stand-alone versions. These games have a much smaller development cost than full-scale theatrical 3D gaming experiences, but can be unexpectedly profitable.

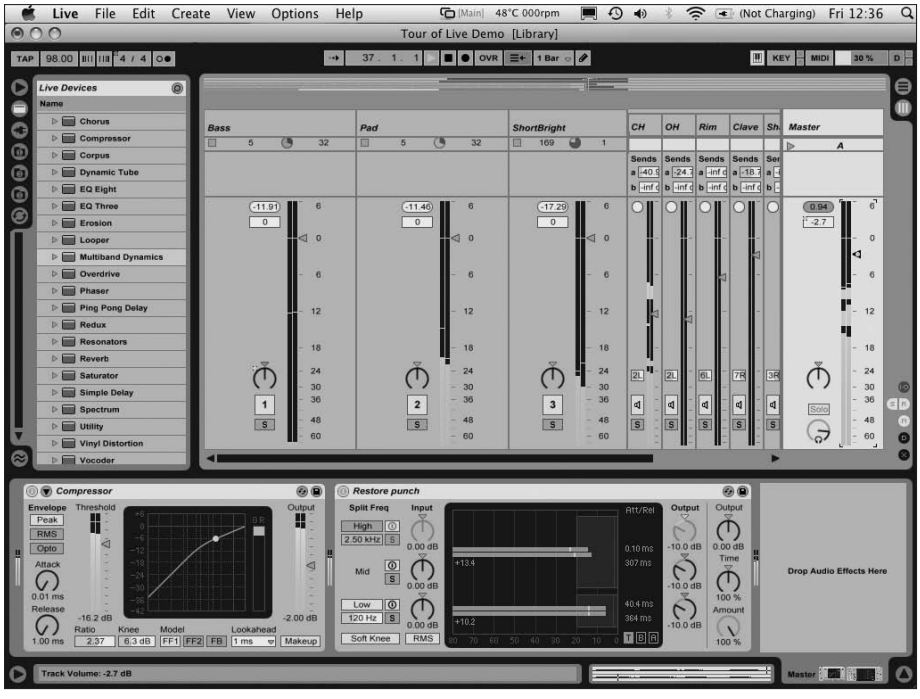
In the creative application market, Apple's own music, video, and photographic products dominate audio and video, supported by a number of other established software products. Elsewhere Adobe's Creative Suite series has become an industry standard for graphic design, photography, and animation. These markets are now mature, leaving little room for competing new products. Occasionally a new product can carve a niche for itself if its features are unusual and appealing enough. For example, Ableton's Live audio sequencer, shown in Figure 1.4, is a popular seller with both Mac and PC users. When it arrived it offered new performance features that weren't available in existing products.

Competing head-on with existing creative products is a poor strategy unless you have a truly market-changing idea. Competing on price is unlikely to be any more profitable. Various budget clones and reinterpretations of Adobe Photoshop have been released on the Mac, but none have successfully taken market share from the original, even when priced much more cheaply. Solo developers should keep in mind that even when products are useful and well crafted, marketing costs can be much higher than development costs.

But head-on competition isn't the only option. There is a thriving market for plug-ins and accessory applications that support the market leaders. For example, software music synthesizer and audio processing plug-ins are steady sellers, and the plug-in and accessory market for video and graphic design applications is similarly robust. Plug-ins and helper applications are smaller and simpler than full applications, and can be developed much more quickly — and profitably.

Figure 1.4

Ableton's Live is one of the few media applications to successfully invent a new application niche. The unique combination of live sample loop playback with interactive control has proved irresistible to musicians and DJs.

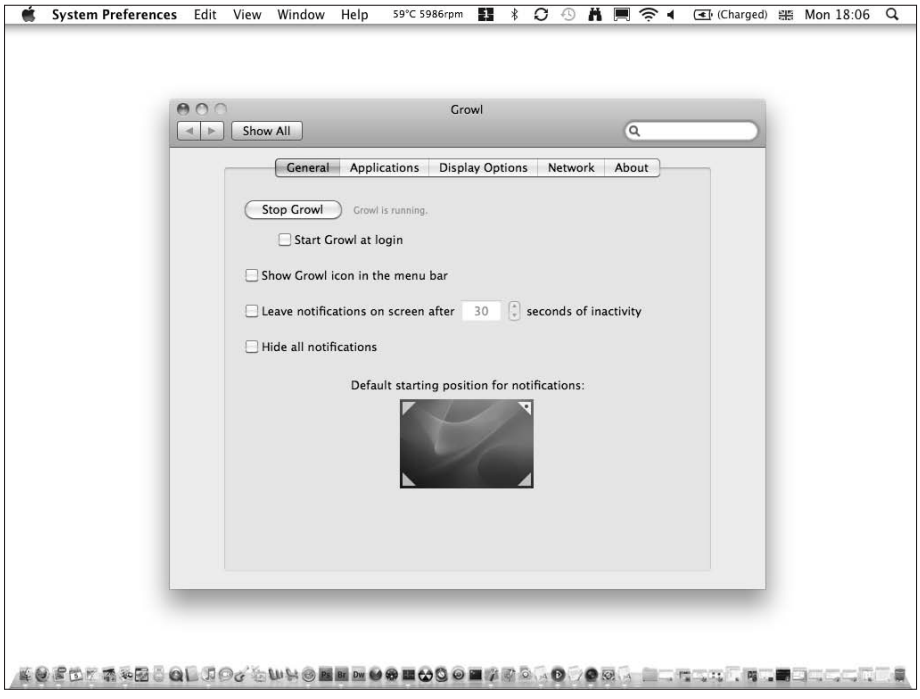


Utility applications are another steady market. Apple's own Finder application has barely been updated since OS X 10.0, and users may welcome a convincing alternative. The standard Mac Mail application lacks refinement and offers a minimal feature set. Automation is another underexplored market. OS X includes Automator, but it is difficult to use and users may appreciate alternative productivity-enhancing options.

As a rule, opportunities are available wherever Apple's software is difficult to use or deliberately limited. This applies to most of the free applications supplied with OS X, so there are obvious opportunities for innovation. Products that solve a simple problem cheaply are always likely to do well. For example, various inexpensive helper applications, such as Growl, shown in Figure 1.5, add useful extra features or enhancements to OS X. Products that appeal to all Mac users but are simple to code and cheap to sell can be ideal projects for solo developers.

Figure 1.5

Growl is a unique and popular accessory that enhances inter-application communication. Growl is donationware, but it is so popular that donations have successfully funded development.



As a prospective developer, online software sales aren't the only possible market — you can also hire out your services on a freelance basis, developing bespoke applications that may never be sold commercially. Business owners and development houses support a small but significant market for experienced Mac developers and consultants. To succeed in this market you'll need to be fluent in OS X and Cocoa, as well as willing and able to use APIs for popular Web services such as Amazon, Twitter, Google, and Facebook. The Twitterrific application shown in Figure 1.6 is one example of this approach.

Developing applications that connect a Mac to these services — or others that are still being developed — can be a reliable source of income. A successful strategy used by some Apple developers is to approach companies with relatively simple but popular Windows applications and ask them if they'll consider outsourcing the development of a Mac version.

Figure 1.6

The Icon Factory's Twitterrific application is a good example of successful and accessible development. It's small, simple, and cheap, and it aims for volume sales by targeting a very popular market.

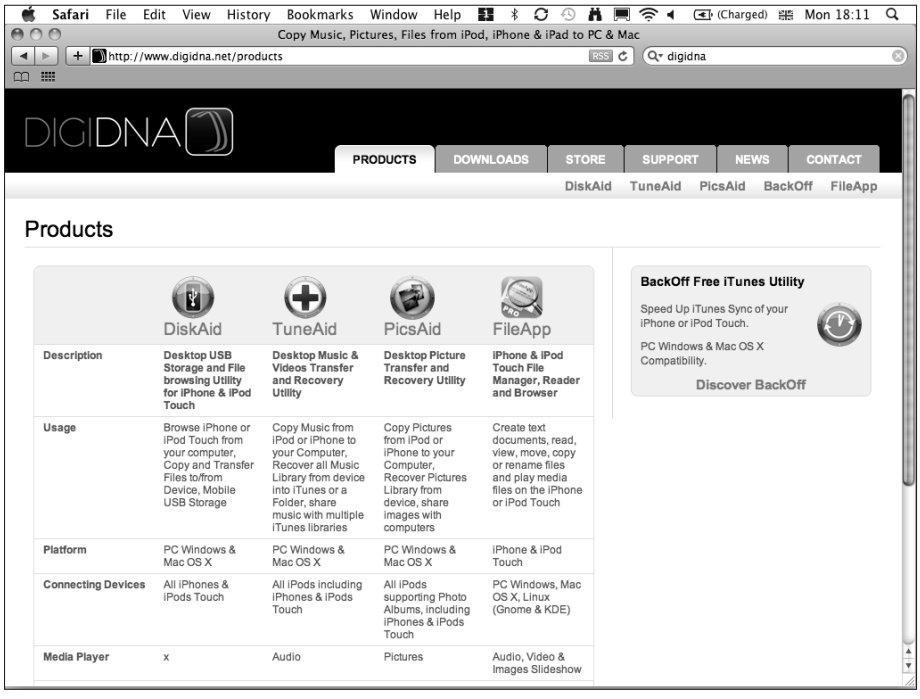
Profiting from the iPhone

Sometimes iPhone development can be even more profitable. There's a small but non-negligible chance that an iPhone hit will make you very successful. It's unwise to be too optimistic about the prospects of overnight success, but some developers have done extremely well by selling relatively simple iPhone apps. For more information, see Chapter 19, which looks at the iPhone's development model and business strategy in more detail.

Other developers have succeeded by offering solutions for common iPhone problems. This is an iPhone-friendly extension of the desktop strategy of looking for opportunities where existing software is limited or frustrating. DigiDNA's Aid series of applications, shown in Figure 1.7, makes it possible to use an iPhone as a USB drive for simple file copying and to simplify file management in iTunes.

Figure 1.7

DigiDNA's range of utilities uses a related model, targeting existing popular Apple applications and creating solutions for some of their shortcomings.



A key difference between Mac and iPhone applications is that Apple has a monopoly on iPhone app sales. Anyone can develop a Mac application and sell it from a Web site without Apple's permission. Apple can only block sales of OS X applications if they break the law in an obvious way.

The iPhone market is much more tightly controlled. Apps must be sold through the App Store and must be approved by Apple before they are listed. Some developers have had unfortunate experiences with the approval process, and their apps have been removed from sale without notice. Apple's stated approval policies aren't always applied consistently. You can expect an app to fail if it uses Cocoa features in nonstandard or unsupported ways, conflicts with Apple's own business plans or those of Apple's airtime partners, or includes illegal or questionable content. Apps that avoid gray areas are usually accepted.

While iPhone development includes an extra element of risk, it also offers access to a huge potential market. Listing in the App Store is a potentially valuable form of free marketing. However, it's not a viable get-rich-quick scheme for developers — a few developers will profit enormously; most won't. But it can provide a significant extra income stream.

Combining Mac, PC, and iPhone/iPad development is an increasingly popular strategy. Applications built from elements across multiple platforms can sell for a higher price, and they also cross-promote each other.

Developing for fun

Not everyone develops professionally, and Cocoa is a rewarding environment for creative media projects. Its features include powerful support for graphics, video, and sound. Adventurous developers who are familiar with accessible Web-centric languages such as Flash and JavaScript are finding Cocoa and Objective-C a natural next step.

Cocoa applications can only run on a Mac. There is no Web-based version of Cocoa, so you can't use Cocoa to create Web graphics. But you can use it to create custom plug-ins for professional media applications; to process graphic, video, and sound files in imaginative ways; and to produce printable ultrahigh-resolution graphics that would be difficult to create in other environments.

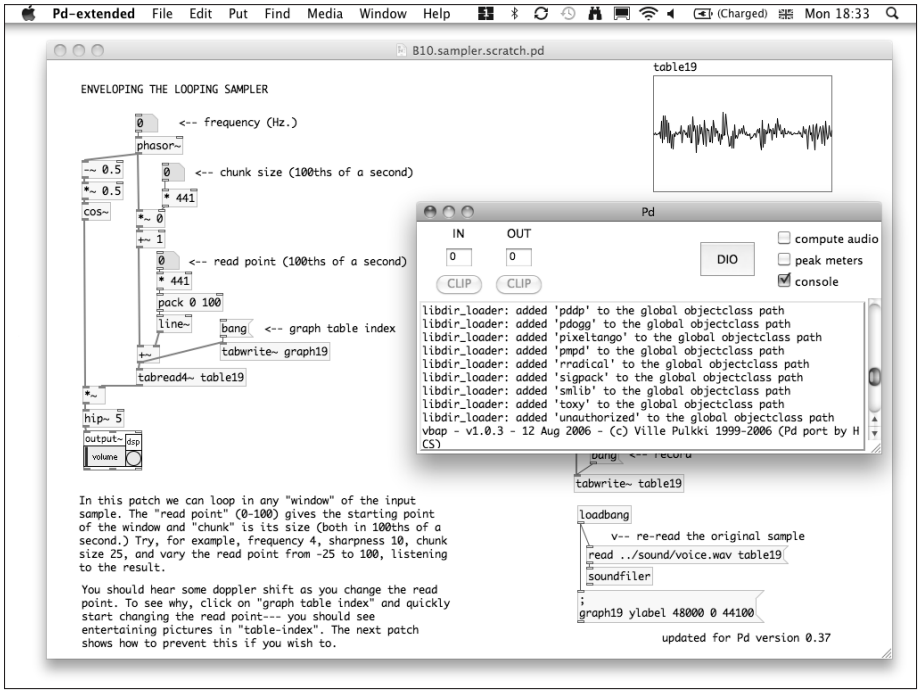
Cocoa isn't as accessible as Flash, Java, or a simplified "toy" language like Processing. It's also not as easy to master as a media construction kit such as Cycling 74's Max/MSP/Jitter product or the Pure Data open-source equivalent, shown in Figure 1.8. If you need a simple and clean but limited environment, Cocoa may not be ideal for you.

Cocoa and Objective-C come into their own when you reach the limits of these environments and begin to explore more complex possibilities. Objective-C is ideal for structured programming of all kinds, including media applications, and the distance between coding in ActionScript and coding in Objective-C and Cocoa is much smaller than the distance between writing code and not coding at all.

While Objective-C is still seen by some as "real" programming, and Flash, PHP, JavaScript, and Perl are considered simpler and less demanding, the distinction is superficial and misleading. As Apple begins to move beyond the desktop/laptop model into new kinds of computing, these technologies will start to collide in interesting and creative ways. Developers who are familiar with all of these environments will have a creative and perhaps professional advantage over those who are less adventurous.

Figure 1.8

Developers and users of nonmainstream applications like the Pure Data audio synthesizer often find that Objective-C and Cocoa development are a natural progression for them once they reach the limitations of these simpler environments.

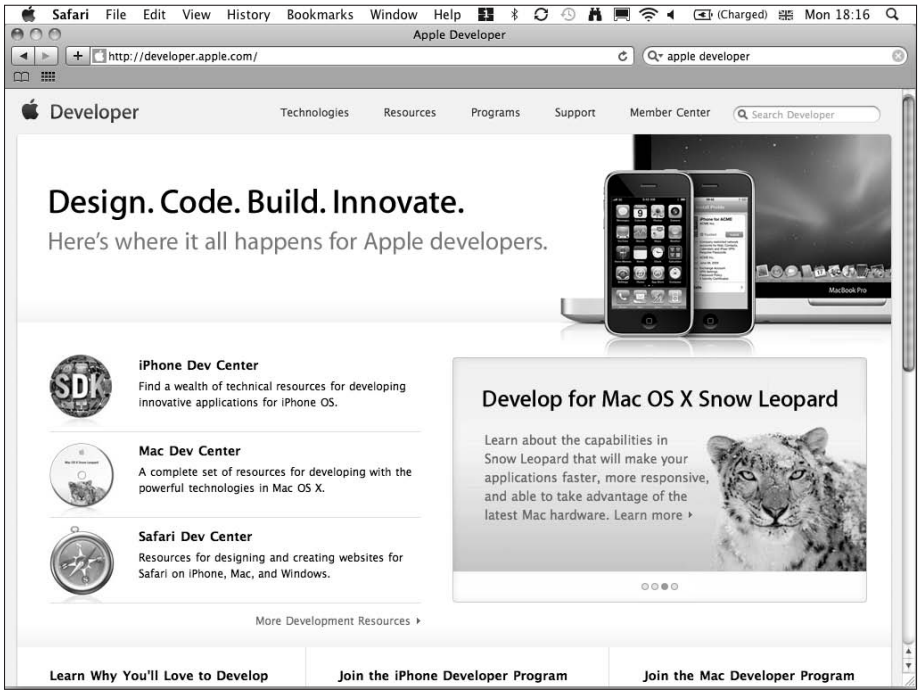


Introducing Xcode and the Apple Developer Programs

If this chapter has inspired you to begin coding, you'll be pleased to know that the Xcode SDK used to create Cocoa applications is available to anyone with an Intel Mac. Entry-level membership requires a simple registration but is free and includes access to Apple's Xcode SDK, which supports both iPhone and OS X development. You can find out more at the Apple Developer Center site at `http://developer.apple.com`, shown in Figure 1.9. For more information, see Chapter 4.

Figure 1.9

Apple's developer centers are open to everyone. Free entry-level registration offers access to the combined Mac and iPhone SDK.



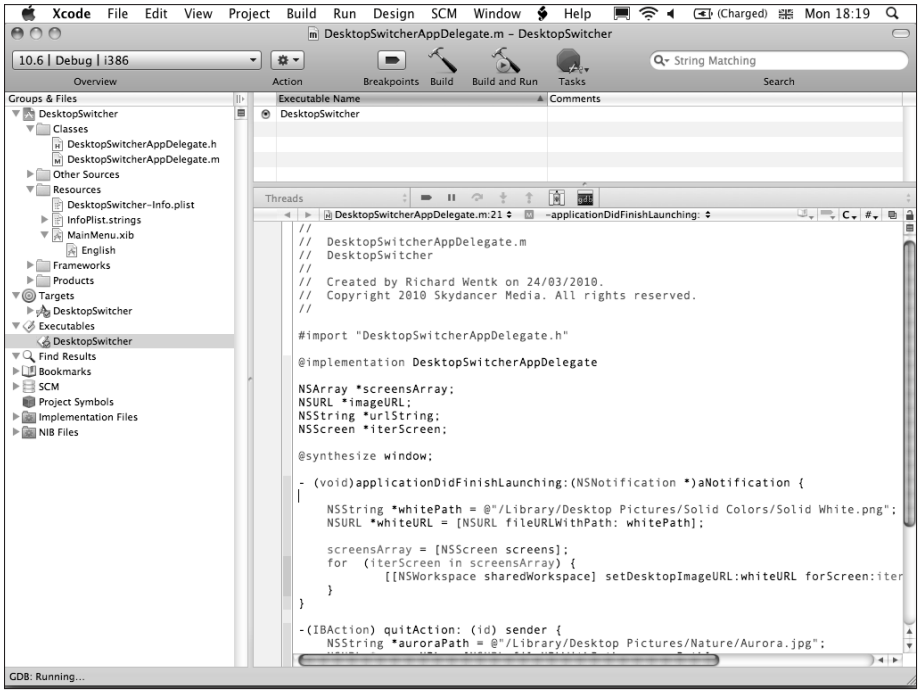
Working with Xcode and Interface Builder

Xcode, shown in Figure 1.10, is a complete suite of development tools; it is described in more detail in Chapter 3. In addition to an editor and compiler, it also includes Interface Builder (IB), shown in Figure 1.11.

At first sight IB — as it's often known — looks like an interface design tool. In fact, the name is misleading. When you create an application, you can use IB to list the objects it loads as it starts. Visible objects appear in the interface, but you can also load data objects that don't appear there. In reality, Interface Builder is an object hierarchy editor that can also design interfaces, and you can use it to define the entire architecture of your application.

Figure 1.10

The Mac and iPhone SDK are built around the Xcode code editor and compiler. Xcode can be used in a simple one-click-to-build way by beginners, or it can be customized and extended almost indefinitely by more experienced developers.

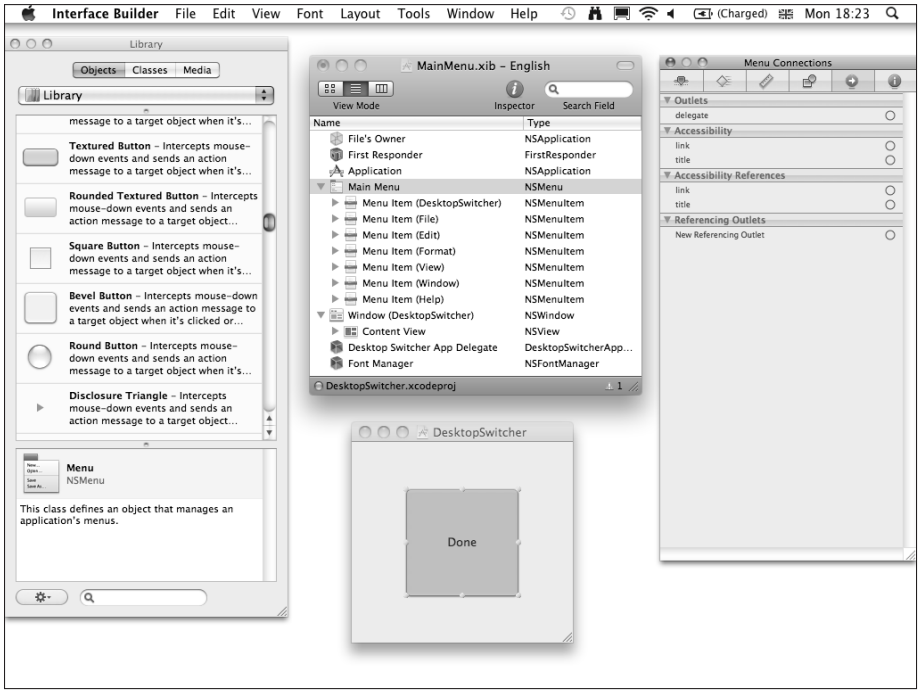


Working with Safari

Apple also offers the free Safari Developer Program for Web applications. On the Mac and PC, Safari remains a minority interest for now. On the iPhone, you can use Web-app technology to create Web-based apps that have many of the features of conventional iPhone apps but are simpler to create and can be sold directly from a Web site. With HTML5 and CSS3 (Cascading Style Sheets 3) due soon in Safari and other browsers, Web apps are likely to have an interesting future. The Safari development tools also include a powerful JavaScript debugger that you can use in other projects. Although this book isn't about Safari, it can be worth exploring the Safari environment as a halfway house between JavaScript and Flash Web development and the more complex challenges of Objective-C and Cocoa.

Figure 1.11

Interface Builder includes a complete list of objects that you can add to an application. It also includes an object hierarchy view, an interface preview, and an Inspector window used to display and edit objects and their settings.



Summary

This chapter introduced the history of Smalltalk and Objective-C and explained how the development process that created Cocoa began with Smalltalk, moved to Objective-C and NeXTStep, and culminated in the integration of the Cocoa libraries into Mac OS with the release of OS X.

It also explored various commercial business models for iPhone and OS X development, and it introduced the Apple Developer Program and the Xcode SDK.

Think Cocoa!

Although it's possible to learn Cocoa by rote, laboriously listing its objects and trying to memorize their key properties and code interfaces, Cocoa comes alive when you understand its design principles in a more open way. Some of these principles derive from Objective-C. Others are related to Aqua — the OS X look and feel built on top of Cocoa — and are described in more detail later in this chapter. It's much easier to master Cocoa by understanding its design philosophy and filling in coding details, as needed, than by working backward from Cocoa code examples.

Designing for Cocoa

Apple products emphasize creative design; a successful design includes the following three elements:

- **Outstanding aesthetics.** Applications should look inspiring and attractive, and users should feel curious and fascinated as soon as they see the interface. But design should enhance features without overpowering them.
- **A sense of fun and creative possibility.** This element isn't always appropriate — for example, it's not usually emphasized in office applications — but even the simplest or most straightforward applications can include elements that inspire users.

TIP

Applications sometimes try to fake fun and creativity by including templates and canned content. This can be popular with users, but it's more rewarding — and more difficult — to create an application that empowers users to be completely original.

- **A clean and intuitive mental model.** Users should be able to understand the application's features with as little conscious effort as possible. The application should be invisible; users should never have to think about how to perform an operation.

CAUTION

Users may have existing ideas about how features should work. A clean and intuitive mental model can still fail if it doesn't match those expectations. For example, users sometimes find iTunes difficult because they assume that an iPod can work as an external USB drive. As a model, an external drive is simpler than the way iTunes really works, leaving them frustrated.

In This Chapter

Designing with Cocoa

Creating Cocoa applications

Moving to Cocoa and Objective-C from other platforms

Applications with complex features always require a learning curve. But it's important to think about these goals and allow them to shape the design of every Cocoa application, even if the result isn't perfect or ideal.

It's easier to approach these aims when creating simple applications, so Cocoa's design philosophy is particularly important on the iPhone and iPad. These devices are optimized for stripped-down and streamlined applications with simple, beautiful, clear interfaces. The mainstream Cocoa environment is likely to trend in this direction in the future, so it's useful to consider mobile aesthetics when designing desktop applications.

Understanding Aqua

Cocoa is part of Aqua. Aqua implements the visual styling of Cocoa applications, including the glassy buttons and gray window borders. Aqua also defines some specific expectations and requirements, as shown in the list that follows.

TIP

You can find the guidelines online by searching for "Apple Human Interface Guidelines." The URL occasionally changes, so it's better to run a search than look for a specific URL.

- **Applications use multiple windows.** Windows typically float over the desktop and not inside a separate application window. There is a single menu bar at the top of the screen. Different windows do not have different menus.
- **Applications should start and quit quickly.**
- **Applications must support the OS X Dock.** Full-screen modes are allowed, and users are responsible for arranging windows on top of the Dock.
- **OS X supports multiple users.** Users should be able to use the same application on the same machine in different sessions without conflicts. OS X supports fast user switching, so users should be able to log out and log in again without losing data or settings.
- **Applications should support internationalization.** Cocoa includes features that simplify multi-language and multi-alphabet support.
- **OS X includes standard features such as preset keyboard shortcuts, color pickers, mouse actions, menu options, drag-and-drop options, and file selectors.** Cocoa applications should use these features.
- **Windows are designed with standard elements.** These elements include a data source at the left, a toolbar at the top, scroll bars at the right, further options at the bottom, and so on. Applications should follow the same outline design.

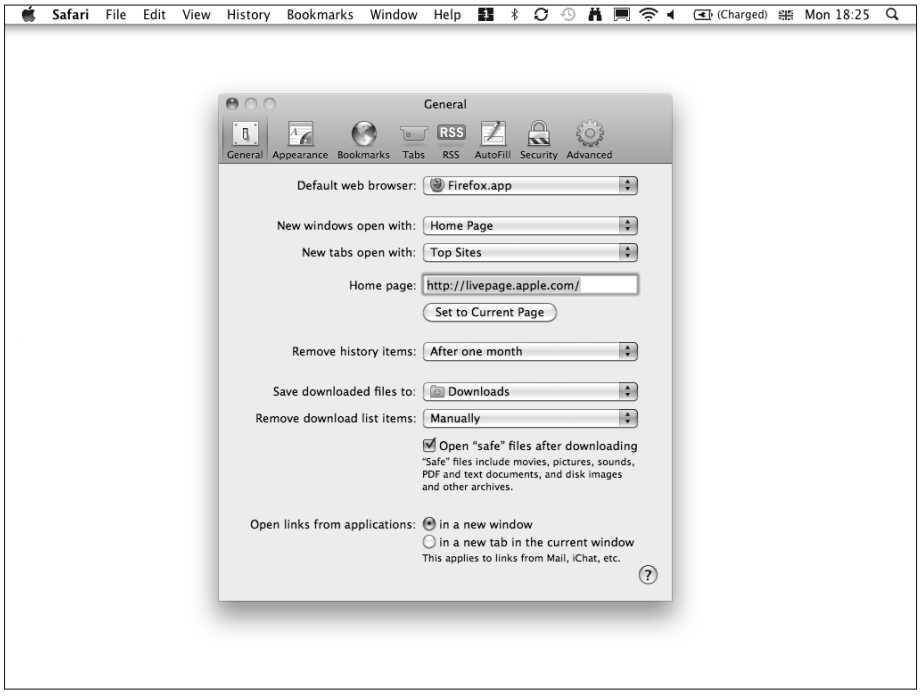
NOTE

If you're new to Cocoa, coming from a different development environment, review the design guidelines for Aqua windows. Elements such as Bottom Bars may be unfamiliar. Not all applications use all elements, but you should be aware of the different options.

- **The spacing of elements within windows is also standardized.** Standards for margins, positioning, grouping, and justification are listed in the Apple Human Interface Guidelines. For example, preferences panes, such as Safari's preferences shown in Figure 2.1, are weighted so that they appear centered in the window, but aren't simply center-justified in a mechanical way.
- **Fonts are partly standardized.** Most applications use the system font, but you can allow users to select other fonts in certain circumstances.
- **Applications should include help files,** with topic browsing and open searches.

Figure 2.1

Safari's preferences are center-weighted rather than center-justified. The layout as a whole appears balanced, even though none of the individual items is placed on the exact vertical center line. Items are also grouped vertically for further clarity.



Using Aqua with Cocoa

Many applications fail to follow all these design guidelines. Because OS X applications can be created and sold by any developer without Apple's permission, developers can create their own interface standards.

Aqua is an ideal choice for mainstream productivity and lifestyle applications, but creative and media applications sometimes take a more individual approach to interface design and feature access. Although parts of Cocoa are closely tied to Aqua, it's possible to use Cocoa to create a completely individual look and feel. Aqua is a set of a guidelines, but Cocoa only enforces them to the extent that it includes a library of standard objects that can be included in an interface. You can customize the appearance of most of these objects if you choose to, and still use Cocoa to define how they respond to user actions.

Creating Cocoa Applications

Cocoa is one library among many others in OS X and isn't a synonym for the OS as a whole. Technically, Cocoa is a *layer* in OS X — a group of libraries that can be used to implement related features. Some care is needed when discussing layers because Apple's documentation describes layers in different ways. For example, in parts of the documentation, Cocoa is described as an element of the *Application Framework* layer — a set of four libraries that can be used to build applications. Elsewhere Cocoa is described as a complete layer in itself. Elsewhere again it's described as a group of separate *frameworks* — code libraries.

Understanding layers and frameworks

You'll find it helpful to understand that some parts of the documentation sketch a functional outline of OS X, while others refer to groups of actual code libraries. Confusingly, the functional outline is only loosely related to the structure of the code. The functional view is best used as an OS X orientation summary and jargon buster, not as a development aid.

Table 2.1 illustrates a functional summary of the layers in OS X. It's an extended version of Apple's own overview, with added comments.

Table 2.1 Functional OS X Layers — An Outline View

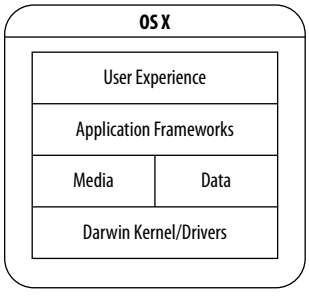
Layer	*Elements*	*Comments*
User Experience	Aqua Spotlight Accessibility	These elements implement or define the most visible parts of OS X in a high-level way. They are either APIs (Spotlight, Accessibility) or concepts and guidelines (Aqua). Simpler applications can ignore the APIs. More advanced applications can use them to add certain standard Mac features.
Application Frameworks	Cocoa Carbon Dashboard & WebKit POSIX and X11	These code libraries are used to create application skeletons. They include window management features, data handling, application setup and teardown, interface creation, and so on. For more details, see Table 2.2.

Layer	Elements	Comments
Graphics and Media	Core Animation Core Image Core Video QuickTime OpenGL Quartz Core Audio	These elements include still image, animation, and video and audio support. This list is a shortened summary, and the names listed here don't map exactly to the names of the frameworks that implement these features. But these names are often used as generic descriptions for groups of related libraries.
Data and Data Management	Core Data Address Book Calendar	Use these elements to manage data within an application and to exchange data with Apple's own built-in applications. The Application Frameworks have their own separate internal data management features. Many applications use both kinds of data management.
Darwin, Kernel and Driver	UNIX system calls Driver libraries	OS X is built on the BSD (Berkeley System Distribution) variant of UNIX and includes all the usual low-level UNIX system calls. This level handles direct — high risk — access to hardware features, such as disk hardware or the temperature sensor. It also handles calls to the underlying Darwin version of the UNIX OS.

Figure 2.2 shows how these elements are related to each other graphically. The most sophisticated and abstract layers are at the top of the diagram. As you travel away from the User Experience layer, you move into the internals of OS X, eventually reaching the underlying UNIX code.

Figure 2.2

This simplified functional view is useful to orient you, but a code-level view of frameworks and layers is more relevant in practice.



Cocoa, Aqua, and the user experience

Aqua is a key part of the user experience. While Coca is a code library, Aqua is a look and feel, defined in part by the glass-like interface graphics pioneered by Apple. As a developer, you don't program an Aqua layer, and there is no Aqua framework. Instead, the visible parts of Aqua are implemented for you automatically. Whenever you add a standard button graphic to your

application, it appears as an Aqua button. In theory, Apple could replace Aqua with a different look in a future version of OS X without breaking any existing Cocoa code.

CAUTION

You can replace the default Aqua graphics features with custom graphics if you choose to. But Aqua is popular with users, and if you're planning to customize the look of an app with your own graphics, you may find it difficult to create a look with equal or greater user-appeal.

Aqua is much more than a set of glassy graphics. As a layer, Aqua also defines a set of explicit design guidelines, and an Aqua-compliant application must meet these guidelines. This is covered in more detail later in this chapter.

The other parts of the user experience layer are built into OS X. Spotlight implements speedy searches. Dashboard displays mini-applications called *widgets*, and is described in Appendix A. Accessibility is used to create accessible applications for partially-abled users. Support for Spotlight and Accessibility is recommended but not obligatory. Depending on their functions and user interface, applications can sometimes leave out Spotlight features without disappointing or frustrating users.

Cocoa and the application frameworks

As Table 2.2 shows, Cocoa is not the only application framework included in OS X. Developers can also use low-level POSIX calls and X11 window management to create Unix-compatible applications; a framework called Carbon to create and manage OS X windows and interface features through C function calls; and Java and JavaScript to create Web-based or Web-like applications.

Table 2.2 OS X Application Frameworks

Framework	*Interface*	*Application Type*
Cocoa	Objective-C objects and methods	Mac-style windowed desktop applications and command line applications that use Cocoa data types.
Carbon	C functions and data structures	Mac-style windowed desktop applications and command line applications using C language data types, structs, and functions. This layer includes a number of special libraries, for example, speech synthesis and speech recognition.
POSIX and X11	C functions and data structures	UNIX-style applications with X11 windowing and low-level command line input. Use this framework to create cross-platform UNIX applications that can run on a Mac and on a different UNIX platform with minimal changes.
Dashboard and WebKit	Java and JavaScript functions and data types	Web-based applications, including WebKit apps for the iPhone, and Dashboard widgets for the Mac desktop. For details and examples, see Appendix A.

Although most new Mac applications are Cocoa applications, the other frameworks are still in use; for example, the iTunes application is built with Carbon rather than Cocoa. This book concentrates on Cocoa and includes very little on POSIX and X11 development. Dashboard applications are described in Appendix A.

NOTE

As with the functional overview, this information is only included to explain some of the terms you'll see in the documentation. You don't usually need to refer to it while creating a Cocoa application.

Cocoa and Carbon

Carbon is a special case because it competes directly with Cocoa and is still widely used. Carbon was originally included in OS X to simplify the move from Mac OS 9, making it possible to rewrite older OS 9 applications with updated but compatible OS X code — a process called *Carbonization*.

Carbon is arguably more difficult to work with than Cocoa and is programmed via a legacy C language interface. While Carbon is still officially supported, its status is becoming ambiguous. Apple has withdrawn a full 64-bit port of Carbon from future versions of OS X, leaving a patchwork of support that makes it difficult for Carbon developers to move to the new 64-bit application model. New developers should use Cocoa almost exclusively.

Cocoa and code layers

Table 2.3 is a code-oriented overview of OS X, showing the relationship between Cocoa and the other OS X frameworks. This view is closer to the organization of the OS X code libraries. It also illustrates how each layer is broken down into specific named frameworks that solve implement-related features.

You'll find these frameworks listed in the developer documentation, with information about adding their features to your applications. A complete list of the frameworks in each code layer would cover many pages. Table 2.3 shows selected key frameworks that represent the most useful elements in each layer.

Table 2.3 OS X Application Layers — Code Frameworks

Layer	*Selected Key Frameworks*	*Applications*
Cocoa Layer	Application Kit NSAnimation Preference Panes Security Interface	Drawing and managing application windows and user interface elements; applying animations; working with supporting data such as strings and enumerators; managing application preferences; implementing basic Web and application security.
Graphics and Media Layer	Quartz and Quartz Core OpenGL and OpenAL AudioUnit QuickTime Core Animation	Drawing and animating text and 2D graphics; creating 3D graphics; recording, playing back, and processing sounds; creating, recording, playing back, and processing video; creating programmable animations for windows and interface elements.

continued

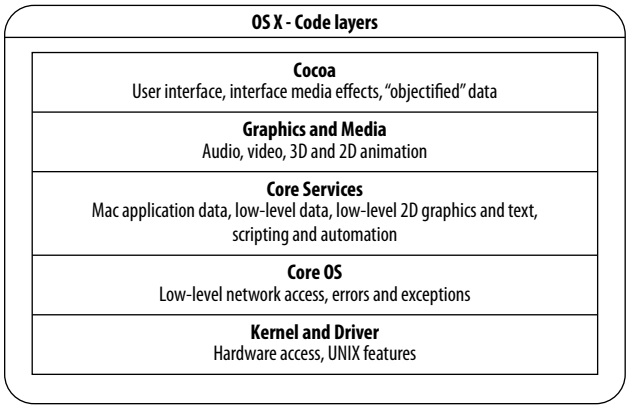
Table 2.3 Continued

Layer	*Selected Key Frameworks*	*Applications*
Core Services	Address Book Application Services Calendar Store Core Data Core Foundation Core Graphics Scripting Bridge Automator	Accessing user data, including Mac Calendar and Contacts; creating and managing application-specific databases; managing scripting and automation events; drawing text and graphics with low-level primitives.
Core OS	System configuration OpenCL Exception Handling	Managing network access; managing device-independent multiprocessing; handling errors and exceptions.
Kernel and Driver	Kernel System CoreWLA IOBluetooth I/O Kit	Using low-level access to the underlying Darwin OS Kernel via a Unix interface; managing Bluetooth, WiFi, and FireWire drivers and data transfers; high-risk direct control of hardware features such as disk controllers; access to the values of various hardware sensors via BSD's `ioreg`.

Figure 2.3 illustrates the code overview graphically. A "Cocoa application" is usually a mix of features and code from different layers.

Figure 2.3

This code-level overview of the layers in OS X summarizes how the layers work in practice. Most applications use features from most of the layers.



Developing features across layers

Divisions between layers aren't clean or simple — there's no discrete "'graphics framework" or "animation framework." Instead, animation features are spread across the Cocoa layer, the Media Layer, and Core Graphics. Similarly, you might expect Core Graphics to be in the Media Layer, but for historical reasons it's included in the Application Services framework in the Core Services Layer.

So adding a feature to an application usually means mixing code and features from different frameworks. It's not unusual to use Cocoa to create a user interface, Core Graphics to draw dynamic elements in the interface, and Core Animation to animate the graphics.

There is also some duplication between layers; for example, you can play sound using the Core Audio or Audio Toolbox layers. This may seem redundant, but these layers offer different features. Audio Toolbox implements simple file player objects with limited features, while Core Audio includes more complex objects that are more difficult to work with, but can be used for more challenging applications such as real-time sound synthesis.

TIP

The organization of the frameworks would be very confusing if you had to use all of them all the time. In fact you only need to master a handful of frameworks. Once you know how to create and control a window, add controls, manage mouse clicks and keyboard events, and draw graphics and perhaps add sound, you've covered the basics — and they're not difficult. The rest is useful and interesting, but optional.

"Toll-free bridged" layers

Some of the layers offer more fundamental duplication. For example, `NSString` in Cocoa is closely related to `CFString` in the Core Foundation framework. Some objects are explicitly described as *toll-free bridged* — Apple's way of indicating that objects and their interfaces are largely interchangeable, even though they exist in different layers.

In fact, many Cocoa objects are simply wrappers for lower-level functions that are listed in other frameworks. You can ignore this if you develop exclusively in Cocoa, and you can also ignore the lower-level functions unless you need to work with them. But it's useful to be aware of the relationship because even when class features are duplicated, they're rarely duplicated exactly. Related frameworks always offer a different balance of features and accessibility, and it's sometimes possible to use a lower-level framework to solve a problem that can't easily be solved in Cocoa.

Using frameworks and layers successfully

If you're new to Cocoa development, you'll have realized now that the layer model is only a very rough guide to OS X. In practice you'll often find yourself digging through the framework reference documentation to combine features in different layers. But with the overwhelming number of objects and libraries in OS X and the slightly chaotic organization, it can be difficult to get a feel for how to approach problems when starting out with Cocoa.

TIP

It's easy for new developers to feel overwhelmed by the number of frameworks and objects. No one with a normal human IQ can remember the entire OS X documentation set. A more practical learning strategy is to master an outline of the relationships between the most useful frameworks in OS X, and then find the rest of their details in the documentation when necessary. You don't need to remember *everything* — but it's helpful to remember where to look when you need to find something.

Table 2.4 shows a simplified view of the OS X layers that strips them down to essentials. This model doesn't match Apple's own classification scheme, but it does match how you're likely to work with layers in practice. In a typical application you'll use the Cocoa Layer to build an application skeleton and to manage key application events and data, and you'll use the Services Layer to manage media and data. Many applications ignore the OS Layer completely, or they hardly ever access it. You'll need to work with it if you create or manage drivers or access the Mac's hardware features. Otherwise, you can ignore most of its features.

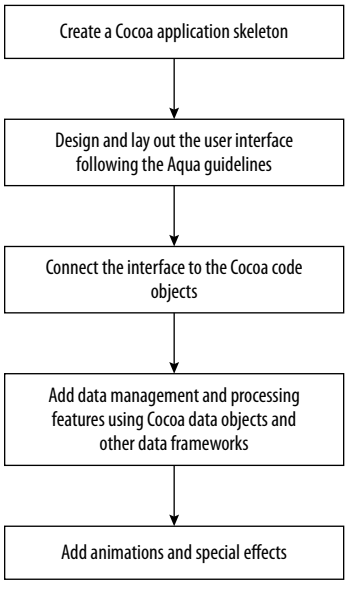
Table 2.4 Simplified OS X LAYERS

Framework	*Interface*	*Applications and Elements*
"Cocoa Layer"	Objective-C objects and methods	Window, interface, and application management; high-level media features; packaged animation effects; object-oriented data; high-level Web and network access; high-level timer and thread control.
"Services Layer"	C functions and data structures	Low-level media features, including basic graphic primitives and customized animations; C-type data structures; low-level Web and network access. This layer duplicates some of Cocoa's features with an alternative C function interface.
"OS Layer"	C functions and data structures	Useful functions that aren't included in the other layers, such as random number generation. Socket-level Web and network access; system and hardware access; low-level process and thread control.

Switching between layers as you work is standard practice. You can simplify development by understanding that most Mac applications are a mix of Objective-C objects and standard C function calls, that there's some feature duplication across the layers, and that the further you get from the user, the more you'll be using OS X's C libraries rather than Cocoa. Figure 2.4 shows a common workflow.

Figure 2.4

There's no single standardized way to create an OS X application, but this outline workflow format is widely used. In practice, most projects iterate around at least some of the steps.



Cocoa and the Framework Story

If the organization of the frameworks seems less than streamlined, that's because it is. The OS X frameworks are a combination of new development and ancient history. In the same way that Carbon was included to simplify backward compatibility, other frameworks were added at various stages in the NeXTStep/OS X development story for reasons that were relevant at the time but may be less pressing now.

Like most operating systems, OS X is an example of computing archaeology. Not only are the legacy layers less complex and less abstracted than Cocoa, but they're also older. If you're finding it hard to understand which frameworks are essential and which are optional, you'll find this information in the rest of this book, which highlights the key elements of Cocoa and OS X.

Occasionally you may use a specialized framework to add a very specialized feature such as speech synthesis to your application. Generally, you can ignore the more specialized frameworks. You can also ignore many of the older frameworks because their features have been duplicated and simplified in Cocoa.

In outline, you'll use the Cocoa Layer constantly, the Services Layer regularly — some elements are essential, while others are barely used — and the OS Layer only occasionally. As you gain more experience, you'll find the older layers become simpler to understand and easier to work with.

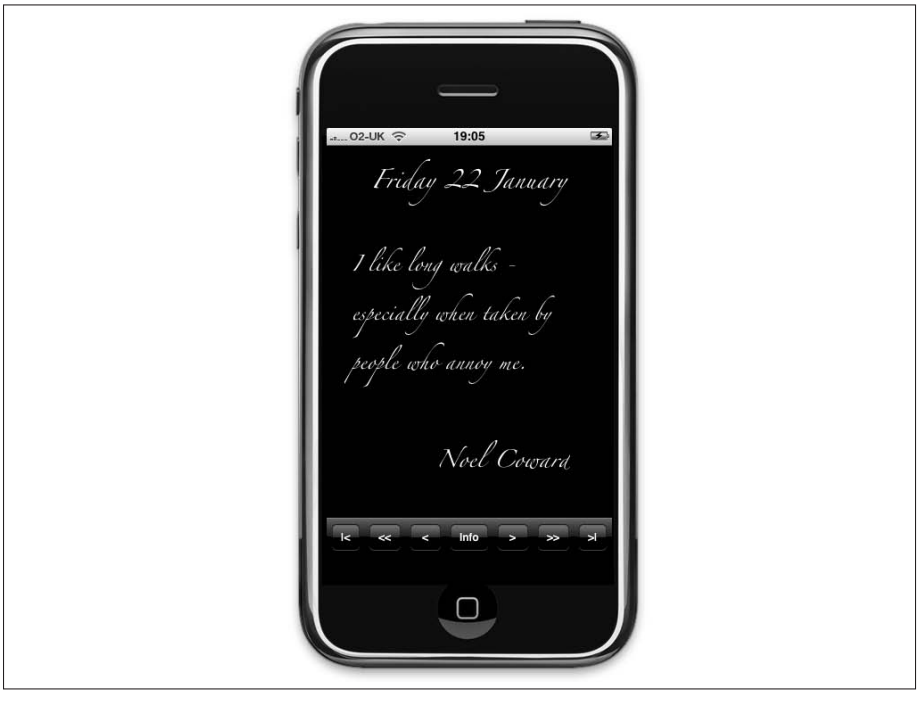
Cocoa on the iPhone and iPad

The iPhone and iPad use Cocoa Touch, which is a development and reinvention of Cocoa. Compared to OS X, iPhone OS offers a drastically simplified collection of frameworks. Cocoa Touch is less complex than Cocoa, with a smaller library of objects, simpler windowing, and fewer data objects, as shown in Figure 2.5.

However, both platforms make similar assumptions about application design and use the same concepts and design ideals. Cocoa Touch applications are less complex and more limited than Mac applications, but they're not fundamentally different. If you're familiar with one environment, you can move easily to the other. Instead of learning a new language, you can make the change by extending or modifying your vocabulary.

Figure 2.5

iPhone apps are simpler than OS X apps, and may not need to use the C-function layers. This Zettaboom app is built entirely from Cocoa Touch components.



Moving to Cocoa and Objective-C from Other Platforms

The Web has created an explosion of interest in software development. There are many scripted and programmed environments to choose from, but historically, most are object-oriented. Developers are finding that competing platforms use related coding principles and that existing skills are transferable. If you have experience with another object-oriented language, you should find Objective-C rewarding and relatively straightforward.

But Cocoa's code libraries are the result of some unusual ideas about OS design. Cocoa code is easy to follow, but the ideas that inspired the syntax are unique, with no equivalents in other environments. To use Cocoa effectively, you must understand them. Objective-C also has unique features that aren't available in other languages.

Apple's developer documentation lists a number of a formal *design patterns* — relationships between events, responses, code, and data — that have influenced the development of Cocoa and OS X. Apple's list is partly historical and academic, and unfortunately it doesn't fully explain some of the patterns, data structures, and principles that are used in Cocoa and OS X. Experienced developers may be able to pick them up by reading between the lines, but newcomers can benefit from a more detailed introduction.

Objective-C's language features are described more formally in Chapter 5, and Cocoa's elements are explored in the rest of this book. This chapter includes an overview of Cocoa and Objective-C for users of other languages. Use it as a cheat sheet that lists essential features that you must learn and be familiar with. Once you've read this section, you'll find it easier to get started with the official documentation. All the features introduced here are discussed in later chapters with practical examples, so consider this section a first look. More detailed hands-on explanations follow.

TIP

Don't skip this section! You'll find this summary invaluable because it tells you key details that you need to know. The developer documentation is less focused and includes unnecessary detail that will distract you when you're starting out. Although this section is a first-look summary, there's a lot to take in. Feel free to return to it as you work through the examples later in this book.

Working with Objective-C objects and messages

Objective-C is object-oriented like Java, C#, and C++. Objects include both methods and properties, and they are split into class abstractions and specific instances. Inheritance and subclassing are supported and encouraged. The syntax and file structure used to define objects is slightly different from that used in other languages, but the concepts remain familiar and recognizable. If you have experience with object-oriented programming in other languages, you already understand enough about the fundamentals of Objective-C to be off to a successful start.

Cocoa applications rely heavily on subclassing. In Objective-C, subclassing is a synonym for customization. When you subclass an object, you re-implement — *override* — some of its existing features and add new features of your own. Some Cocoa objects are explicitly designed for subclassing and include method stubs that you must flesh out with your own code.

One unusual feature of Objective-C classes is class messaging. In other languages, you create instances of class objects and send messages to the instances. In Objective-C, classes exist as meta-objects in their own right. They can include *class methods* that are run on the class itself. This feature is often used when creating objects: you send a creation request to the class in a message, and the class creates an instance of an object and returns a pointer.

Objective-C messaging

In Objective-C, messaging is a fundamental feature. Instead of running a method on an object, you send it a message that triggers the method. Messages are events, not function calls. They are *asynchronous*; that is, messages can happen at any time and aren't tied to a sequential program flow.

NOTE

It's a useful simplification to pretend that messages are sent and processed instantly. Of course, real code takes time to run. But to a first approximation, Objective-C tries to hide this from you. To a more advanced and accurate approximation, Cocoa code can run in separate threads, and Cocoa includes thread-management features that can control the timing of events and messages. But until you begin working with threads, processes, timers, and performance profiling, it's useful to ignore the time-dependent elements of Cocoa code and assume that events are processed instantaneously.

The following is a list of the most common ways in which messaging is used in Objective-C. Some will be familiar from other languages. Others are less traditional.

- **Getting an object property.** The only way to read an object property is by sending it a message and asking for a return. There is no direct-read feature in Objective-C, because objects are *opaque* — their internal workings aren't visible to other objects. If you want to make a property visible in your own objects, you must add code to make it so.
- **Setting an object property.** In the same way that you must add code to read a value, you must also add code to set it. Typically, setting a property doesn't just save it to memory, it also triggers a behavior, which may ripple through related objects. For example, setting an object's position moves it on the screen automatically. In a less straightforward example, setting a single object's position can automatically change the positions of other objects associated with it so that they move as a group. Cocoa handles many of these interactions automatically. As long as you initialize and arrange your objects correctly, you can use Cocoa to simplify and automate some of their responses.

TIP

Creating setter and getter code — as it's known — can be a repetitive chore. Objective-C includes a `@synthesize` directive that makes boilerplate setter and getter code unnecessary. `@synthesize` works like a macro. It takes a list of properties and automatically creates setter and getter code for them. The code doesn't appear in your source files, but it does appear in the application and is visible while debugging. When an object is synthesized, it appears to have simple readable properties, but in fact these properties are still accessed through getter code.

- **Getting a property or value after processing one or more parameters.** This feature is close to a traditional function call. If you trigger a method and pass a parameter list, the object returns another object or a value. It's relatively rare for an object to act as a simple data processor. More typically, objects respond to parameters in more complex ways and trigger a behavior and optionally return a value. Return values are used to indicate success/failure or report status.
- **Creating a new object.** The only way to generate a new object is by asking an existing object or a class to create it.
- **Initializing an object to some state.** Objects are complex data structures, and it's not possible to use simple assignments for copying or initialization. Most objects include initialization methods that either create useful preset values or initialize the object with data passed from another object.
- **Releasing an object when it's no longer needed.** This memory management feature is built into Cocoa's object management system and is described in more detail in Chapter 12.
- **Handling specific OS events.** When OS X receives a user action, such as a mouse click or movement, it can pass it to your application for handling. OS X also sends messages spontaneously — for example, when memory is low or the system is about to shut down. Your application should include message handlers for these events.
- **Posting information about object status.** Objects can send messages whenever their internal state changes. OS X also sends standard messages to an application as it moves through various states. For example, the `applicationDidFinishLaunching:` message is sent to an application when OS X has finished loading it. You can use it to run custom initialization code.
- **Forwarding a message.** Message forwarding is a key feature in Objective-C. Objects can ignore a message and forward it, or they can process a message internally and then forward an identical copy. The OS X *responder chain* — a list of objects in every application that can process user events — uses message forwarding.
- **Dealing with events that are about to happen.** OS X posts a warning before certain events. For example, your application can be warned before a window closes. It can use this warning to shut down animation code and perform a cleanup.
- **Deciding if events or responses should happen.** Messages replace conditional tests. In some contexts, OS X asks your application for a Yes/No response — for example, to enable or disable certain user actions. Unlike fixed conditional code, the response can vary according to the application state. In a more complex example, Cocoa may enumerate a list of objects automatically, and it may use messaging to ask code in your application to set their properties as it does this.

NOTE

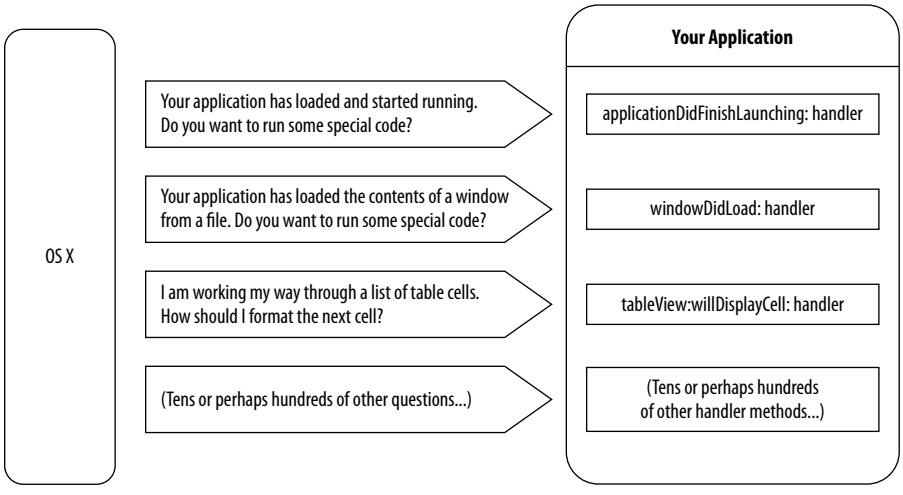
Dealing with events that have just happened, are about to happen, or need to be confirmed before they happen is part of the *delegation* feature in Cocoa. Delegation is described in more detail in Chapter 6. In theory, delegation is about splitting the burden of response between objects, and it's also about extending the possibilities of messaging. In practice, it's often used in the three ways listed above.

Question-response messaging and delegation

You don't need to remember the preceding list. The simplest way to understand messaging is as a question-response dialogue. OS X uses the question-response design pattern, shown in Figure 2.6, to implement application features. When OS X sends a message to your application, it's asking a question. Your application can choose to ignore the question or it can respond with a behavior, with data, or both. Questions are answered by including handler methods for possible messages. Messages without an associated handler method are ignored.

Figure 2.6

Question-response messaging. This slightly fanciful view of messaging is a useful and accessible mental model of the relationship between an application and its OS X environment.



OS X sees every application as a list of disconnected message handlers. It doesn't begin at the beginning of an application and execute the handlers in order. Instead it calls each handler when it needs to. The order in which the handlers appear in your code doesn't matter. It's important that a handler exists, and it's useful to keep related handlers close together for clarity and to make the code easier to maintain. But if you put all of an application's handler methods in a single long file in a random order, the application would still work.

If you have experience with Flash or JavaScript, you'll understand how `onload` and `onclick` messages are sent and processed asynchronously. OS X takes the same principle much further by keeping track of a list of the objects used in an application and sending messages accordingly. This can generate a truly vast number of possible messages for hundreds or thousands of possible events and circumstances.

Some messages can be ignored, while others are obligatory. If you don't include methods that respond to these messages, certain objects won't work correctly; some of their features will be disabled: they'll appear on the screen and do nothing or may not appear at all.

Implicit question-response and delegation

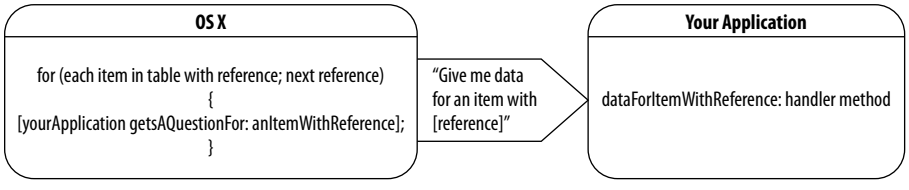
Question-response can be *explicit* or *implicit*. Explicit messaging makes the connection between messages and responses obvious in code. For example, the link between a button and its handler method is usually explicit. Many features in Cocoa make implicit assumptions about when messages are triggered. The context in which OS X is asking for a response may not be visible.

For example, table objects often use implicit question-response to set the appearance of table items and to request data for each element. In a more traditional OS, you might create your own looping code to iterate through a list of cells, and use a function call to set their properties. In Cocoa, the looping is *implied* — it's an internal OS X process you have no control over, managed by implicit code that you can't see or modify.

If you use a table view, your application is expected to implement a method that supplies data for each cell. No obvious code triggers the method, but you can see that it runs — for reasons that seem mysterious until you understand how this feature works. Figure 2.7 illustrates this process graphically.

Figure 2.7

Implicit question-response hides the source of the question within the internal workings of OS X. This is easy to understand for one-off application events, but more of a challenge to follow when the object relationships are more complex or use implied iteration.



Implicit questions are often implemented with *delegate methods* — optional helper methods that aren't always listed in the main class reference for an object. If an object isn't doing what you expect, it's likely that you haven't implemented all of its required delegate methods. Delegation is explained in more detail in Chapter 5, with further examples throughout this book.

CAUTION

When you design your own objects, you can use the same design question-answer pattern to define how they respond to each other. Custom objects aren't allowed to ignore messages. If you send an unrecognized message to an object, your application crashes. Internally, OS X solves this problem for Cocoa object by cheating — every possible message includes a stub do-nothing handler method. When you write your own handler for a Cocoa object method, your code overrides the stub.

Cocoa messaging and notifications

Cocoa includes a separate messaging system with more sophisticated features. *Notifications* are like an internal e-mail or chat service. You can use them to pass information between applications or between objects within an application. Objects can sign up to receive notifications and can post them through a notification center. Notifications can include complex data. They're a system-level feature, while Objective-C messaging is a language-level feature. The two messaging systems are unrelated and are used in different ways.

Objective-C syntax

Now that you've seen some of the ways in which messaging is used in Cocoa, you may be curious about messaging code. Objective-C uses square brackets to indicate messages. For example:

```
[anObject doSomething];
```

triggers the `doSomething` method in `anObject`.

The messaging syntax can pass parameters.

```
[anObject doSomethingWith: anInt];
```

triggers `doSomethingWith:` and passes it an `int`. Parameter fields can have descriptive name strings. For example, in

```
[anObject doSomethingWith: anInt andABoolToo: YES];
```

`doSomethingWith:` and `andABoolToo:` are part of the message syntax. They're effectively code-based tooltips, included for clarity and legibility, and they don't affect execution.

CAUTION

Objective-C treats `aMethod` and `aMethod: thatTakesAParameter` as completely different entities. It's good practice in your own code to keep method names as distinct as possible. Some Cocoa method names don't do this; you'll occasionally see an identical method name followed by a number of different parameter groupings. Objective-C understands the difference, but you may find it harder to remember.

This idiom is often used in Cocoa classes to include hints about the purpose of each field. For example

```
[aWindow setHasShadow: YES];
```

enables a drop shadow effect for `aWindow`. Replacing `YES` with `NO` turns off the drop shadow. This feature helps make Cocoa code easy to read. But sometimes it gets out of hand. Objective-C is chatty rather than terse, and some Cocoa methods have very long names. The full syntax for the `dragImage:` method built into Cocoa's `NSWindow` window object is

```
[aWindow dragImage: anImage
                at:aPoint
            offset:anOffset
```

```
        event:anEvent
  pasteboard:aPasteboard
      source:anObject
   slideBack:aBool];
```

This isn't quite so easy to remember — and if the field names were more descriptive, the method would be even harder to parse. Unfortunately long method names are unavoidable in Objective-C and Cocoa.

TIP

It's good practice to split very long method names across multiple lines to make them more legible, putting each field on its own line.

Fortunately, you hardly ever have to type a long method name by hand. Xcode includes code automation that autocompletes method names for you. You can also copy and paste long method names directly from the documentation and fill in the fields with property names used in your code.

Messages can be nested, and the return from one message can be used as the input for another. A standard example is

```
MyClass *thisInstance = [[MyClass alloc] initWithData:
   initialData];
```

This idiom is used throughout Cocoa. The `alloc` method — short for allocate — is sent to `MyClass` and returns an instance of an object. The returned object is immediately initialized with initial data in the `initWithData:` method. A pointer to the initialized memory area is passed to `thisInstance` and used as the object reference.

Objects and asterisks

Objective-C objects are referenced with named pointers defined with asterisks. Objective-C uses C's asterisk convention for pointers, so you'll see asterisks everywhere in Objective-C code. A standard idiom for creating a new object is

```
NSClass *myObject = [...aCreationMessage...];
```

The left-hand side of the assignment specifies that `myObject` is a pointer to an instance of `NSClass`. You can also predeclare pointers and reference them later in the usual C-like way:

```
NSClass *myObject; //Declares myObject as a pointer to an
   instance of NSClass
[...otherCode...]
myObject = [...aCreationMessage...]; //Creates an instance of NSClass
   and assigns it to myObject
```

Until you create and assign an object to a pointer, it remains NULL — but it still has a type.

Casts are widely used as "code tool tips" while passing parameters in messages to remind you which type of object to pass in a parameter field. For example:

```
…thisIsAParameterField:(NSClass *)myObject… //Reminds both you
   and the compiler that myObject should be an instance of
   NSClass
```

Objective-C "objectification" with the @ character

Objective-C uses the @ character as an objectification operator that converts whatever follows into an object. This is most often used in text string definitions. In other languages, you delimit a string with quote marks. In Objective-C, you must also prepend the @ character to convert the string into an object. For example:

```
NSLog(@"This is a string");
```

This prints "This is a string" using Cocoa's NSLog class, which logs output to a console window. You can also use the C-function equivalent:

```
fprintf (stderr, "This is a string");
```

This line creates exactly the same result and is a functional synonym in conventional C without objectification.

In Objective-C, you *must* convert literal strings into objects wherever they appear. This is true whether they're text strings, file and path elements, or used for some other purpose. If you leave out the @ character — which you often do, until you get used to typing it — the compiler won't always report an error. In some circumstances, forgetting the @ can cause a crash.

Managing data in Cocoa and Objective-C

Objective-C supports all the standard C data types, including arrays and custom structs. Cocoa adds its own collection of Cocoa objects that store and manage data. The most important objects are introduced in Table 2.5 and described in detail in Chapter 5.

Table 2.5 Cocoa Data Objects

Data Objects	*Description*
NSArray	Stores a numbered list of objects. Supports enumeration. NSArray is much more complex than a simple C-type array.
NSDictionary	Used throughout Cocoa. You must master NSDictionary to work with Cocoa successfully. Stores *key-value pairs* — pairs of named values — and can perform automatic lookup on a key to return its value.
NSSet	Stores an unordered collection of objects. NSSet is faster than NSArray when you need to test if an object is in a collection.

Data Objects	Description
`NSValue`	Works as a wrapper for conventional C or other Objective-C data types. "Objectifies" them and provides a standard interface for accessing them, copying them, and comparing them. `NSNumber` is a subclass of `NSValue` that deals exclusively with numerical objects.
`NSString`	The Cocoa string object is vast and complex with many features, including support for non-Western character sets. It has very little in common with a simple C string array.

These data objects and their supporting methods are used throughout Cocoa to exchange information and return data. Typically you initialize a data object with useful values and pass a pointer to it to another object. When OS X sends a message to your application, it often passes one or more of these Cocoa objects in the parameter list. Similarly, you often need to pack data into one of these objects before you send it in a message. This can be inconvenient when you're passing a single data item rather than a group, set, or list, but it's often unavoidable.

CAUTION

One of Cocoa's strangest features is that by default, Cocoa data objects are read-only. You can't edit the contents of an `NSString` or change any of the key-value pairs in an `NSDictionary`. Cocoa includes separate editable versions of each object that are *mutable*; for example, the editable string class is named `NSMutableString`. You can't make an object mutable after it's created. If you want to be able to change data, create a mutable object and initialize it with your starting data.

Cocoa includes a much wider collection of data objects, and the complete list is defined in the Foundation Framework. It features objects that handle Web access, manage date and time information, control threads, and implement other complex features. Table 2.5 shows the objects that you must master because they're used throughout Cocoa code. The other objects are more specialized and used less often.

The CoreFoundation framework, which is distinct from the Foundation Framework, offers equivalent data types with a C language interface; for example, `CFMutableDictionary` is equivalent to Cocoa's `NSMutableDictionary` and is used to store key-value pairs in a very similar way. Code that works in the "System Layer" and the "OS Layer" below Cocoa relies heavily on these data types. They're used to access user data from the address book and calendar databases, set up and manage media operations, and work with hardware peripherals. The Core Foundation versions of these objects are simpler than their Cocoa equivalents and implement a smaller number of supporting features. But because they're "toll-free bridged," you can — with very little extra effort — use any Cocoa method on a Core Foundation data object.

Copying data objects

Objects are referenced through pointers. A standard mistake is to assume that you can copy objects with a simple assignment:

```
anObject = anotherObject;
```

Other languages support this. In Objective-C, this line compiles correctly, but it simply copies pointer values. After the assignment runs, you have two pointers to the same data instead of one. The data itself remains blissfully untroubled by the operation.

Objects are complex and may include other objects in a hierarchy or tree, so copying them may not be a simple process. Cocoa includes a set of standard `NSCopying` methods to support copying, but different objects implement a method called `copyWithZone:` with varying levels of intelligence and sophistication. When you create custom objects of your own, you must include code that implements the standard copy methods.

Many Cocoa objects include methods that create and initialize objects using the data in another object. This may seem counterintuitive when you encounter it for the first time because it looks redundant. In fact, it's often the best way to copy data. For example, you can copy an array to a new array with

```
newArray = [NSArray arrayWithArray: oldArray];
```

If you need to copy objects, don't look for direct copy methods in the documentation; if "copy" methods exist, they're often disguised as initializations.

CAUTION

This code is slightly more complex than it looks because it's an example of class messaging in Cocoa. Instead of sending a message to an existing array, it sends a message to the `NSArray` class and passes the existing array as a parameter. Sometimes you copy objects by sending a message to an object, sometimes by sending a message to a class. There's little consistency in how Cocoa handles this; you must check the class documentation to find out which option to use.

Comparing data objects

A related mistake is attempting to use a simple C comparison to compare objects:

```
if (anObject == anotherObject)…
```

You can do this in other languages to compare strings. In Objective-C, this code compiles and runs, but instead of comparing data, it compares pointer references, which are almost always different.

To compare the data, you must use a standard comparison method called `isEqual:`.

```
if ([anObject isEqual: anotherObject];)…
```

`isEqual:` is available in typed variants for comparing specific objects, including `isEqualToString:` and `isEqualToArray:`. These variants are optimized and are more efficient than the generic comparison method. When you create a custom class, you'll need to implement a custom `isEqual:` method of your own if you want to compare instances.

CAUTION

Note the square brackets in the code. The brackets aren't optional; `isEqual:` is a message and not a function. You must include the brackets or the code won't compile.

Key-value pairs in data objects

Although it's possible to use objects as keys in certain contexts — for details see Chapter 12 — a key is usually an `NSString` used as an identifier. Keys must be unique because there is no way to distinguish between identical strings. A value is any Objective-C object. C data types must be "objectified" before they can be used as a value, wrapped in an instance of NSValue or NSNumber. Chapter 12 includes sample code for this.

A key-value pair links keys to values, making it possible to read or write the values using the key as a reference. Key-value pairs are used throughout Cocoa. They're one of Cocoa's fundamental data idioms and are accessed through `forKey:` statements:

```
//Set aKeyName to aValue
[myObject setValue: aValue forKey: @"aKeyName"];

//Read the value of aKeyName and copy it to aReturn
aReturn = [myObject valueForKey: @"aKeyName"];
```

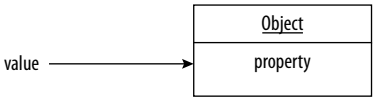
NOTE

You can see in this example how the text strings are "objectified" with the @ character.

All Objective-C objects implement a related but different form of key-value access, which supports named property access. Most object-oriented languages support direct property access, as shown in Figure 2.8.

Figure 2.8

Simple direct property access is implemented in Objective-C. But key-value pairs also allow more complex kinds of access.



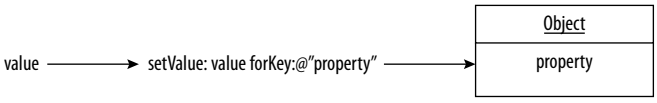
This is also available in Objective-C. But Objective-C also supports indirect property access through the key-value system. If `myObject` has a property called `thisProperty`, you can set it with

```
[myObject setValue:aValue forKey:@"thisProperty"];
```

Figure 2.9 illustrates visually how key-value access works. The key string must match the property name. As long as there's a matching name, you can use this feature to access any property in any object. This feature is called *key-value coding*. It's built into Objective-C, and all objects support it.

Figure 2.9

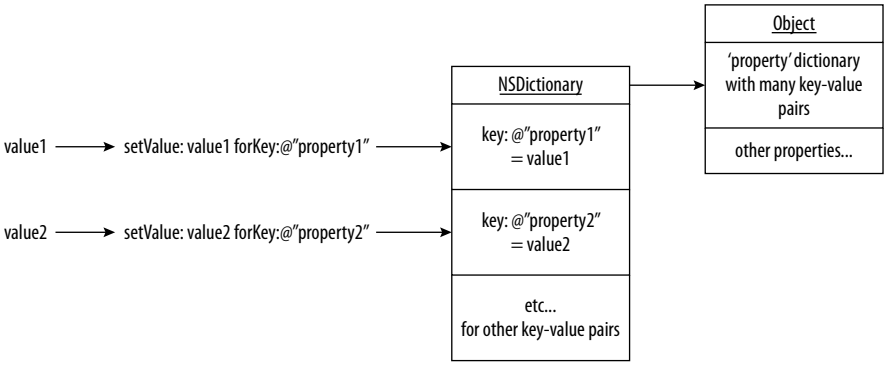
Indirect property access with a key uses a text string. The text string doesn't have to be a string literal; it can also be a mutable string.



Cocoa's data objects support a third kind of key-value storage, in a separate data area. Many Cocoa class definitions include lists of predefined keys. For these objects, dictionaries stand in for a separate list of pseudo-properties. Instead of triggering a response from an object by modifying its properties directly, you create a dictionary, fill it with key-value pairs using the list of defined key constants in the class documentation, and pass the dictionary to the object in a message. When the object receives the dictionary, it unpacks it, reads the values for each key, and responds accordingly. Figure 2.10 illustrates this.

Figure 2.10

Many Cocoa objects use key-value pair dictionaries to implement "pseudo-properties." Some objects also return data in a dictionary, and you'll need to search the dictionary for specific keys to read them as "properties."



In code, this approach looks like this:

```
[myDictionary setValue:value1 forKey:@"property1"];
[myDictionary setValue:value2 forKey:@"property2"];
...
[anObject doSomethingWithTheValuesIn: myDictionary];
```

"Pseudo-property" dictionaries are used throughout Cocoa, most obviously in the media layer, where they're used to control animation, video, and sound features, but also in the network management frameworks and in some user interface objects.

Cocoa uses this indirect approach to save memory. It's an efficient solution when objects have hundreds of possible properties but only use a few at a time. For example, a multiformat video player object might need to support a number of different video file formats, each with a different set of properties. It would be possible to define the player object with a complete set of properties for every possible format, but this would be wasteful. It would also be difficult to maintain and might break backward compatibility.

It's more efficient to define a player object with a single property dictionary. As new formats are supported, the possible key-value pairs for each format in the dictionary can be updated without changing the object's interface. The key-value and dictionary system provides a flexible alternative to a list of fixed object properties, at the cost of some extra setup and teardown.

Objective-C memory management

Garbage collection — automated memory cleanup of unused objects — arrived relatively recently in Objective-C, and legacy code continues to use an older reference counting model to manage memory. If you're developing for the Mac, you can now use garbage collection to clean up memory for you. If you're developing for the iPhone, you don't get a choice — garbage collection isn't available, and reference counting is the only option.

The theory of reference counting is simple. Every time you reference an object, it increments a hidden counter. When you no longer need the object, you release it with

```
[anObject release];
```

When the reference counter reaches zero, the object is removed from memory. It's an obvious and easy mistake to assume that using `release` releases the object immediately, but in fact it doesn't — it tells Cocoa that your code is no longer referencing the object, and that it *may* be released.

In theory, you should always be able to balance references and release calls. In practice, sometimes you can't without obsessively tracing all possible execution paths. In some applications, you may not be able to balance them at all because Cocoa's referencing mechanism may not work as you expect it to. Occasionally it doesn't work as it's supposed to, making memory errors inevitable.

It's often easier to create objects when your application starts up, and release them when it terminates. Officially, this is considered bad practice — resources should ideally be created or loaded when needed and released when not used — but sometimes it can be a practical way to make memory management tractable.

For similar reasons, it's good practice to avoid creating objects inside loops. This applies whether memory is managed manually or automatically. It's more efficient to update or re-use an object inside a loop than to create it and release it repeatedly.

CAUTION

Creating objects repeatedly inside a loop and never releasing them is a good way to waste memory, slow your application to a crawl, and generate random crashes.

Whether you choose to use reference counting or are forced to use it, expect to spend many hours debugging memory leaks and fixing random crashes. Windows, Java, Flash, and Unix all include garbage collection. It can be challenging to have to get used to managing memory without automation. On the iPhone, this is one of Objective-C's weakest and least appealing features. Fortunately the Xcode development tools include a set of applications that simplify testing and make it easier to monitor and track memory leaks.

Objects and conditionals

You can't create objects within a conditional block and refer to them outside it. For example, in

```
if (condition) {
 createAnObject;
 doOtherStuff…
 }
doStuffWithTheNewObject;
```

the compiler can't be sure if the object will exist outside of the conditional. It assumes that it doesn't, and reports an error.

Cocoa file management

Cocoa includes a file selector object that loads and displays a standard OS X file list. Under the hood, file management is less conventional. Files aren't saved but *archived*. Archiving automatically includes a conversion stage that transforms an object or collection in memory into bytes on disk. Loading a file invokes a de-archiver that reverses the process.

This may appear indirect, but it's an example of Cocoa's emphasis on abstraction. When you're saving an image to disk, you always need to convert it to a standard graphic file format before writing it out. Cocoa makes it easy to do this; for example, it includes a JPEG archiver. You can also create your own archiving and de-archiving code for custom file formats.

Cocoa archiving

Cocoa's data objects include built-in archiving. For example, you can write a dictionary to disk with

```
[aDictionary writeToFile: aFilePathString atomically: YES];
```

This automatically archives the dictionary and writes it to a file in a format that can be read back with a corresponding

```
thatOldDictionary =
[NSDictionary dictionaryWithContentsOFile: aFilePathString];
```

NOTE

The `atomically` parameter selects a two-stage write that attempts to write a complete file, and then renames it to the specified file name if the write succeeds. If the write fails, the file fragment is deleted. This guarantees that files on disk are always complete.

Including file write methods inside data objects is unconventional. It's more common to include separate file writing code that takes objects and their contents as a parameter. Cocoa also supports this more traditional option, and it's discussed in more detail in Chapter 5. But keep in mind that Cocoa data objects can read and write their own contents directly to and from disk.

Using NSCoder

A special object called `NSCoder` handles more complex archiving and de-archiving. In many applications, saving a file means saving a list of objects. You can't use direct archiving for this because you'd have to write each object to a separate file.

With an `NSCoder`, you can create your own archiving method in a custom object and fill it with a list of encoding methods. This joins your list of objects together, creating a single packed archive. Optionally, but usefully, you can label each field with a key, making it easy to access it later. The archived object in each field is treated as a value. To reverse the process, implement a corresponding decoder method. For practical examples of `NSCoder`, see Chapter 12.

File paths and URLs

Cocoa makes as little distinction as possible between local and remote data. You can download a file from a Web site as easily as you can load it from disk. In Cocoa, text file paths and URLs are closely related. Paths can be converted into URL objects — and sometimes paths *must* be converted into URLs because some objects can only access local data via a URL.

Although the OS X file system is based on a standard UNIX hierarchy, in practice, system and user directories often require specialized access code. This is even more relevant on the iPhone, where apps have access to a very limited and stylized version of the UNIX hierarchy and data is sandboxed for security reasons.

Exploring other Cocoa features

Cocoa includes a selection of other unique features that developers must be familiar with. Some are introduced in the developer documentation, but the documentation doesn't always explain how these features are designed to be used.

Cocoa network support

Apple's networking system is called Bonjour, and Cocoa includes objects that can find data on a Bonjour network. Specifically, Cocoa can find a list of network services. Unfortunately, most data transfers have to be handled at a lower level of the OS. Web support is more comprehensive. Cocoa includes simple and direct access to online files and can also support online security. For less Web-centric online data access, it's fairly easy — but not trivially simple — to assemble an application that accesses an online API. Chapter 10 has more details.

Windows and views

OS X supports a windowed interface, with support for various interface objects, including buttons, sliders, text fields, file lists, preferences panes, and others. In Cocoa, windows are wrapped

around *views*. A view is an active area in a window that draws graphics and responds to user actions. Cocoa includes an `NSView` object that manages views. In practice, you create an application interface by customizing an instance of `NSView` with your own code. Associated classes can display tables, forms, text fields, groups of cells, and other standard interface features, including smaller objects such as sliders, number boxes, and buttons.

In theory, any item that the user can see and interact with is called a view. This is most obvious on the iPhone, which includes various complex predefined view types, including a Web view that displays Web pages, a table view that displays a list of items in a table, and so on.

One source of confusion is the relationship between individual items in a view and the appearance of the window as a whole. When designing an interface, the entire interface is the window's view. But smaller objects are also views. To avoid confusion, think of the interface as *the* view, and the smaller objects as subviews or as view objects. Very small single-feature views such as buttons and sliders are usually known as *controls*, although occasionally they're also called views, just to confuse you.

Views and objects in Interface Builder

Interface Builder (IB) is the Xcode view design tool. Designing a view might seem to be a simple visual process, but there's a hidden subtlety. It's not immediately obvious to new developers that resource files — called *nib files* — can be loaded automatically. By default, an application loads at least one nib file when it runs. You can override the loading process and control it manually, if you choose to. If you don't, it just happens — no code is needed.

It's possible to set up nib files so that they load other nib files. In fact, you can use Interface Builder to define a collection of objects, views, and data structures across multiple files, and the entire collection will be loaded automatically. If you think of IB as an interface editor, you'll miss this essential feature. You'll also find it difficult to understand how applications are organized. For more information, see Chapter 8.

Model, View, Controller

Apple's developer documentation emphasizes the Model, View, Controller (MVC) design pattern, shown in Figure 2.11. MVC means that data is kept isolated from views, and that controller objects translate messages in both directions. Views never communicate with data objects directly.

This might seem like an unnecessary extra complication, but it increases efficiency. In the same way that a Web browser doesn't need to download the entire Internet to display a single page, MVC means that an application's interface only ever displays a subset of its data. Cocoa applications are designed to use MVC and include predefined controller objects for various data types.

Figure 2.12 shows one possible example. The model is a table with thousands of entries. A good way to make an application impossibly slow would be to copy every cell's data to an editable cell in a list on the screen, keeping all of the visible cells in memory at the same time.

Figure 2.11

Model-View-Controller — MVC — is one of the fundamental OS X and Cocoa design patterns and is built into Cocoa's object hierarchy.

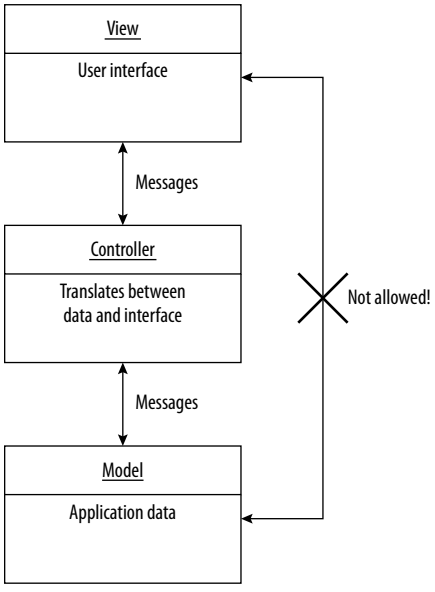
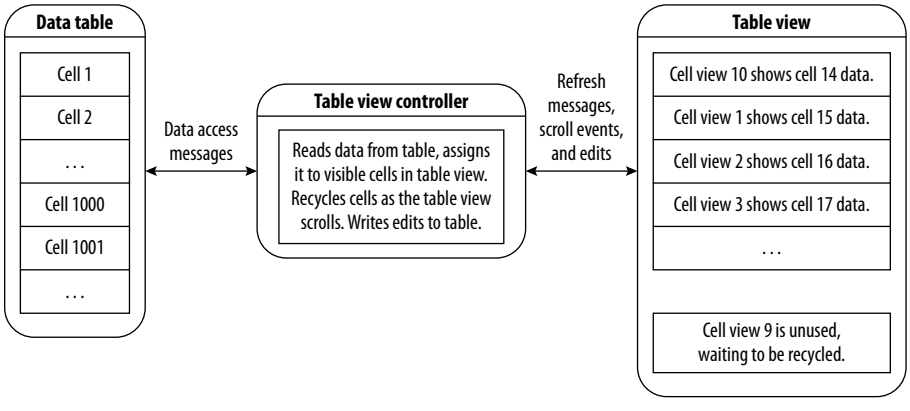


Figure 2.12

Applying MVC to a table display makes it possible to pass data selectively to and from the interface, as required. This is a more complex but much more efficient solution than a direct link between table data and a list of scrollable visible cells.



MVC allows a more efficient implementation. Cells still appear on the screen in a scrollable list, but they have no direct connection to the data source. As the user scrolls up and down the list, the controller removes cells from the top or bottom, refills them with data, and reinserts them at the bottom or top. Instead of displaying thousands of cells, the application can create the same scrollable effect with a handful. The controller object automates cell updates, and the process is transparent and invisible, at the cost of some extra setup and teardown.

Controllers are glue code, but they're intelligent glue code. They simplify access in both directions. Cocoa and Cocoa Touch both include prewritten controller classes to simplify interface design and eliminate unneeded code. You can also create custom controllers for your own applications.

Bundles and plists

A bundle is the list of files included with an application. A bundle typically includes the `.app` executable, at least two icon files, and optional data, graphics, and support files. Applications also include at least one *plist* — a property list stored as XML files that points OS X at key application resources and includes essential application preferences. Cocoa includes objects that can load data from a bundle and both read and write plists.

TIP

In OS X, an application's .app file is really a folder that contains the application's bundle. To view the contents of a bundle, right-click an .app file in Finder and select Show Package Contents. You'll see a Contents folder that you can open to reveal the bundle's files and folders.

Comparing Cocoa to other platforms

Every platform and environment has its own unique qualities. Objective-C and Cocoa are no different. Cocoa's biggest strength is its sophistication. The object library is rich and powerful and it's often possible to implement features with a few lines of code. The learning curve isn't negligible, but the initial frustration is a temporary phase. With experience, Cocoa starts to feel productive and enjoyable.

Setup and teardown

If you're used to a simpler language like Flash, expect to spend some time dealing with new coding overheads. If you're drawing graphics in Flash, a Flash stage is created for you automatically and you can immediately draw on it with ActionScript's drawing functions. This makes Flash creatively rewarding because you can concentrate on results, not on overhead code.

In Cocoa, you must create the equivalent of a stage yourself. Depending on the application, this can take many extra steps using code from a significant number of classes and frameworks. Initially, this can make Cocoa feel difficult and unresponsive. It's not unusual for newcomers to feel that they're disappearing into a rabbit hole of descending classes, frameworks, methods, and constants until they complete their setup code. Sound, animation, and data management incur similar overhead. Because there are no shortcuts, you'll sometimes need to learn a lot about Cocoa and OS X to implement a relatively simple feature, especially when encountering setup for the first time.

The Cocoa approach has advantages: it is flexible and open-ended; is able to handle output to various devices, from printers to screens to files, automatically; and can create advanced effects with very little code. Looked at positively, setup code can often be reused as boilerplate. Once you have your equivalent to a Flash stage, you can recycle the setup code in other applications with only minor changes. Even so, allow for extra setup and teardown before you start; there may be more than you're expecting.

Abstraction and object orientation

Cocoa is highly abstracted and object oriented. Relationships between causes and outcomes are loose and can be customized. For example, an unsophisticated operating system might include a loop that captures keyboard events into a buffer and a minimal framework that can read and process events. Your application might read the events and process them from the buffer.

Cocoa takes a much less direct approach. Keyboard events are abstracted into messages, and your application must include handlers for some or all of the possible messages. For example, Cocoa's user interface manager object `NSResponder` sends various messages as a user types on the Mac keyboard, such as:

```
cancelOperation:
capitalizeWord:
deleteBackward:
willPresentError:
yank:
```

This list of edit control messages includes more than 170 items. The full list of messages that `NSResponder` can send runs into the hundreds and implements mouse control, touchpad control, gesture sensing, graphics tablet control, and other standard features.

Through abstraction, Cocoa implements much of the event processing and some of the application design for you. Although these message names imply text editing, you could also use them in a different application, such as a mathematical equation editor. You can choose which messages your application responds to. If you don't include a handler method for a message, the message and the user action are ignored; if you do include a handler, you can choose what it does and how it works. Cocoa doesn't force you to implement features that match the method names, although it's less confusing for the user if you do.

Abstraction adds flexibility, often in a useful way. It also adds complexity. Whereas a simple OS sets colors with three numerical RGB components, Cocoa encapsulates color in an `NSColor` object. There's no option to set colors directly. You must create and initialize an `NSColor` object, set its properties to the required components, and then use messaging to allow other objects to read the properties — perhaps to draw a line or to set a point. Sample code might look this:

```
NSColor *myTranslucentBlack = [NSColor initWithCalibratedRed: 0
   green: 0 blue: 0 alpha: 0.5];
[aReceiverObject drawSomethingWithColor: myTranslucentBlack];
```

This code calls the `NSColor` class to return a new `NSColor` object with translucent black color values. While it isn't impossibly complex, it's more complex than a hypothetical

```
line (x1, y1) to (x2, y2) with 0,0,0,0.5;
```

in a simpler language. If you're managing memory manually, you must remember to release the color object when you no longer need it. If you're managing memory with garbage collection, you still need to be careful about creating objects unnecessarily or inefficiently. Be especially careful around loops — creating objects within loops is always inefficient, and you should avoid it where possible.

Abstraction is one of the features of OS X that contributes to the initial challenge of setup and teardown. It would be useful if Cocoa offered simplified classes and objects for common requirements. Cocoa sometimes does, but often it doesn't.

Generally, Cocoa tries to pack as much generality into objects as possible. The `NSColor` object allocated previously includes built-in support for color calibration. This feature is free — it's built into `NSColor`, and calling the `initWithCalibrated:` method invokes it automatically. Before it returns a color object, `NSColor` refers to the current calibration table and adjusts the color constants accordingly. Other Cocoa objects are similarly rich and detailed, and they implement sophisticated features automatically. This is often a good and useful thing. But sometimes your application's needs don't coincide with Cocoa's features, and you'll either need to work around them or re-implement them manually.

Moving to Cocoa and OS X from Windows

Windows development has always been pitched at two levels. Visual Basic and the related Visual Basic for Applications provide entry-level access. For more advanced developers, Microsoft has shifted emphasis toward the C# language and the .NET collection of frameworks.

Visual Basic can seem very different than Objective-C, but there are familiar points of reference. Visual Basic applications are event driven, making them similar in outline to the Cocoa environment. But Objective-C and Cocoa are more precise, more demanding, and more complex, with many more features and objects. A good strategy when moving from Visual Basic is to start small and keep it simple. Instead of immediately trying to build an entire application, begin by creating very simple application skeletons. Flesh them out gradually with features — menu support methods controls, more complex views — working back from the interface toward media and animation support, data management, and finally networking and hardware. You'll find this strategy used in the rest of this book.

For more experienced developers, C# is recognizably similar to Objective-C. But there are obvious differences in syntax and concept. For example:

```
aClass.aMethodWithAParameter(20);   //C#
[aClass aMethodWithAParameter: 20]; //Objective-C
```

In Objective-C, class prototypes and implementations are split into separate header and implementation files. This is an Objective-C tradition rather than a language feature, but most developers follow it. Elsewhere, Objective-C lacks C#'s generics; classes can accept untyped

parameters using a generic `id` class, but type recognition isn't automatic. C# also has better support for *boxing,* the ability to disguise and wrap low-level data types into full objects.

There are more obvious differences in the development environments and the associated libraries. Xcode is based on the GNU compiler suite and is a powerful development environment with a homespun feel and almost endless potential for customization. The Windows development suites have a different history and are less open, but some features may feel more professional.

Cocoa, Windows, Yellow Box, and Mono

Yellow Box is — or more accurately, was — a long-rumored but mythical optional element in OS X that would allow Cocoa applications to compile and run in Windows without changes. In the same way that Universal Binaries made it possible for developers to create applications for both G-series and Intel processors with minimal code modifications, Yellow Box was slated to do the same for Windows and Mac applications.

Simple cross-platform development would still be a huge boost for the Mac developer market, and the popularity of the iPhone suggests that an iPhone version of Yellow Box would take the Windows developer market by storm. But so far Yellow Box shows no signs of appearing. Apple no longer mentions it and — perhaps most tellingly — it no longer features in Apple rumors.

Yellow Box is unlikely for practical reasons. Even a simple iPhone version would be dauntingly difficult. There is no good technical reason why many Cocoa features couldn't be made to run under Windows, but complete compatibility is unlikely, and efficiency would suffer. On the Mac, Cocoa hooks directly into OS X. On Windows, many features would need to be translated into their nearest Windows equivalents, with adjustments and corrections. Full-speed execution would be unlikely.

There are also very significant differences in developer tools. In the original plan, Yellow Box would work on a Mac but would compile Windows binaries. Currently, a Windows version of the Mac SDK would be more interesting and useful. But the SDK relies heavily on Unix features and code, and an equivalent version for the Windows environment would be very difficult and time-consuming to develop — especially if Apple continued to give it away for free.

Yellow Box also remains unlikely for political reasons. At the time of writing, iPhone app development is booming. Allowing millions of Windows developers to join the gold rush could swamp the market, perhaps diluting it to the point where tens of billions of apps sold just a few units each. Financially, iPhone development is driving an uptick in Mac sales, and this would disappear if a Mac were no longer essential. So there is no strong incentive to release a Windows version of the SDK.

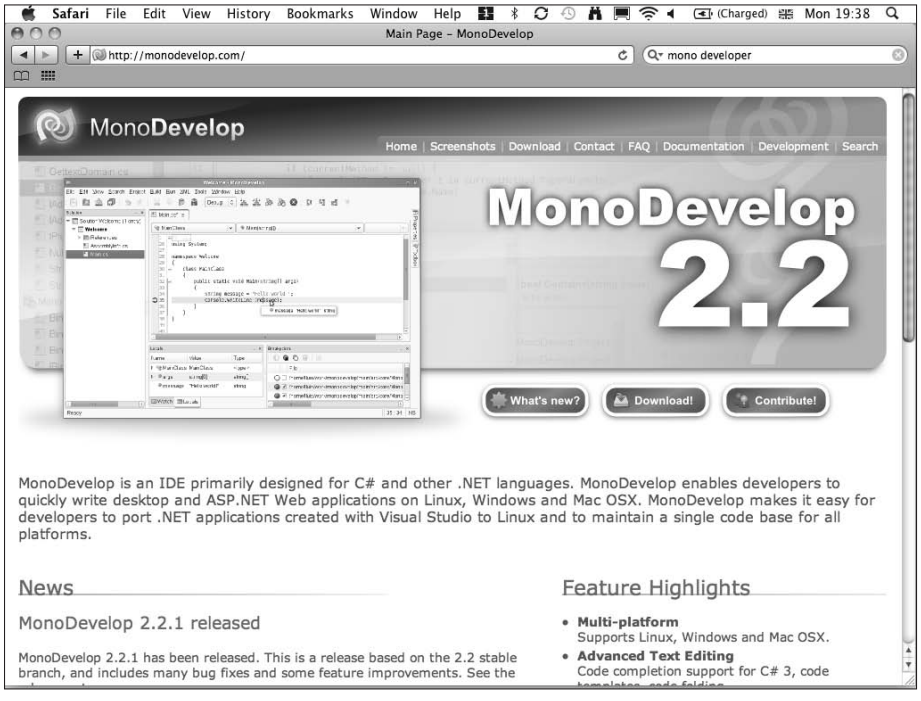
While it's possible that Apple is secretly continuing to develop Yellow Box or a more modern alternative, there's no reason to believe that if such a project exists it will ever see the light of day. At best it would be kept secret as insurance against unforeseen developments. More realistically, it won't appear at all.

While Apple has left Yellow Box on the shelf, competition has appeared from an unexpected source. Novell has sponsored an open-source implementation of C# and the .NET frameworks called Mono, shown in Figure 2.13. The MonoDevelop environment is free and supports cross-platform development across Windows, Linux, and OS X. It doesn't support Objective-C, so you can't port Objective-C and Cocoa projects to Windows. But it does support multiplatform development in C# and .NET, so you can compile applications for three platforms from a single code base. There's also a commercial version for iPhone development, although it currently sells for $399, making it an expensive choice for nonprofessional developers.

As a generalization, Cocoa is smaller and more tightly focused than the Windows libraries. Xcode is at least as productive as Microsoft's Visual tools, but it has a different emphasis and a different mix of more refined and less successful features. But if you're used to C# and .NET, you'll find that you can master most of Cocoa and Xcode quite easily.

Figure 2.13

Mono is an alternative solution for cross-platform development, but as a C#/.NET environment, it doesn't provide access to all of Cocoa's features. It's also banned from the App Store — Apple doesn't support iPhone OS apps created with non-Apple SDKs — but can be used for OS X development.



Moving from Flash

At first sight, Flash, Cocoa, and Objective-C have little in common, but in fact many elements of Cocoa and ActionScript mirror each other with uncanny precision. Table 2.6 shows a short and very incomplete list.

Table 2.6 Object correspondences in Actionscript and Cocoa

ActionScript	*Cocoa*
`Object`	`NSObject`
`MovieClip`	`NSView` in Cocoa `UIView` on the iPhone
`Button`	`NSButton` in Cocoa `UIButton` on the iPhone
`Event`	`NSNotification`
`this`	`self`

There are also obvious differences. `NSButton` includes methods for adding a bezel and border, managing a complex mix of possible button states, and setting up an optional mouse-triggered auto-repeat feature. `Button` in ActionScript has no equivalents.

While it's possible to create graphic effects exclusively in Cocoa, many applications also need to use some of the functions in the Core Graphics and Quartz libraries, with associated setup and teardown code that has no Flash equivalent.

Generally, moving from ActionScript to Cocoa is rewarding and unexpectedly easy. Although Cocoa code can't run in a browser, Cocoa runs more quickly than Flash, and Cocoa objects typically have more features, making it easy to develop complex effects that would be impossible in Flash.

TIP

Flash game developers on the iPhone will want to look at the open source Cocos2D framework at `www.cocos2d-iphone.org`. It implements a simplified language that makes it easier to concentrate on the details of event programming while hiding the more complex elements of Cocoa Touch and Objective-C. Flash video developers will want to explore the Quartz Composer application included in Xcode. Quartz Composer is a simple but powerful programmable video synthesis and processing toolkit.

Moving from Java

Java borrowed heavily from Objective-C, and so many of Java's language features have a direct equivalent in Objective-C. Java is more fluid than Objective-C. Classes can be defined and implemented more spontaneously, so the Java environment can feel less formal and more creative. Java is also more abstracted than Objective-C. It doesn't support low-level access to hardware or to the underlying operating system. Whereas Objective-C is tightly bound to Cocoa, Java developers can choose between various GUI toolkits.

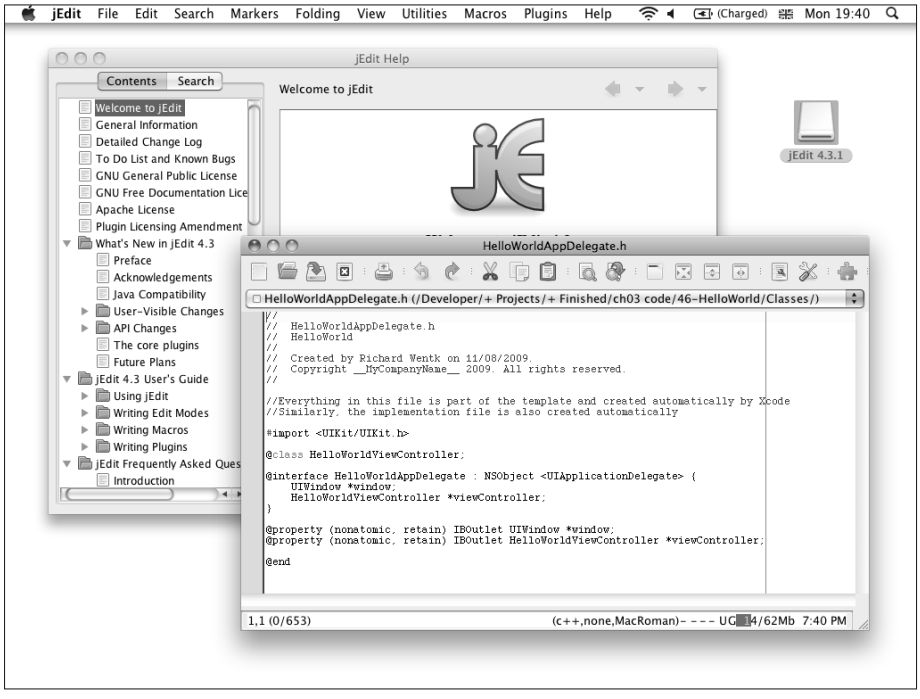
Java has no direct equivalent to Cocoa or to the lower-level OS X layers, so culture shock is likely when moving from Java to Objective-C. Although both languages are object-oriented, Cocoa and OS X are huge and detailed, and so the learning curve can be steep. Java developers can continue using the same grammar, but will need to learn a much larger vocabulary and set of

ideas and idioms. On OS X, Java's SWT (Standard Widget Toolkit) includes links to some of Cocoa's features and can be used as a transitional learning aid.

Java is inherently cross-platform, and OS X includes Java as an application framework, so developers can create Mac applications, such as the jEdit code editor shown in Figure 2.14, without using Objective-C or Cocoa. But Objective-C code is much faster and more efficient, and the Cocoa and OS X layers implement a much wider selection of features. Java remains a good choice for simple applications, but more complex projects need the full performance of Objective-C and Cocoa.

Figure 2.14

jEdit illustrates the flexibility and limitations of Java on OS X. It implements a very useable text and code editor, but lacks the deep links with UNIX and OS X that are built into Xcode.



Moving from C or Assembler

If you're used to traditional procedural programming, you'll find Cocoa more challenging. You'll feel at home with the frameworks in OS X with a conventional C interface. But Cocoa and Objective-C may be more difficult to understand because they make different assumptions about how to approach problems and build solutions.

Although it may not be obvious from the code, Cocoa's philosophy is very visual. It's also event driven. You may find you can improve your understanding of Cocoa development by sketching the links between objects graphically. This can help break old procedural habits and make the overall structure and flow of events in a Cocoa application easier to follow. Examples are included in later chapters.

Summary

This chapter has covered a lot of ground. Don't try to assimilate it all at once. It introduced information explored in more detail in later chapters, with added practical examples. The chapter began with a first look at the Aqua design guidelines and then explored the relationship between Cocoa and the rest of OS X, looking in detail at the OS X frameworks and layers.

Next, it looked at some of the fundamental features of Objective-C and the standard data types and design idioms used throughout Cocoa and OS X. Finally, it examined some of the similarities and differences between OS X and Cocoa and other popular development platforms.

Introducing the Cocoa and OS X Documentation

After the brief tour of frameworks and layers earlier in the previous chapter, the OS X and Cocoa documentation should be easier to understand than if you'd dived into it cold. Apple makes the Cocoa documentation, shown in Figure 3.1, available in two places. Developers can download and install the free Xcode SDK, as described in Chapter 4. In addition to a suite of design tools, the SDK includes a full set of documentation visible in a simple Safari-based viewer. Nondevelopers and prospective developers can view the Cocoa documentation online. Currently the URL is `http://developer.apple.com/mac/library/navigation`.

CAUTION

Apple occasionally rearranges its developer Web site, so the documentation URL isn't guaranteed to remain accurate. If the URL changes, expect a redirect to the new location. If no redirect is available, you can find the location by searching for "Mac OS X Reference Library." The library has always been freely available online and this is unlikely to change in the future. But Apple is moving away from disk-based documentation to online access, so it may not be bundled in full with future versions of Xcode.

Using the Apple documentation efficiently is a key skill, which isn't easy to master. The documentation itself is a slightly chaotic mix of history and new features, with varying degrees of detail. The browser can also be difficult to use.

TIP

You can solve the browser problem by using a free third-party helper application called AppKiDo, available from `http://homepage.mac.com/aglee/downloads/appkido.html` and shown in Figure 3.2. AppKiDo is highly recommended. It transforms and simplifies the browsing experience, and can literally save you hours of browsing time. It also groups and sorts related Cocoa classes in useful ways, making it easier to follow how Cocoa is organized.

In This Chapter

Understanding resource types

Using the documentation

Figure 3.1

The developer documentation is formally known as the Mac OS X Reference Library. There's also a separate iPhone OS Reference Library for iPhone developers.

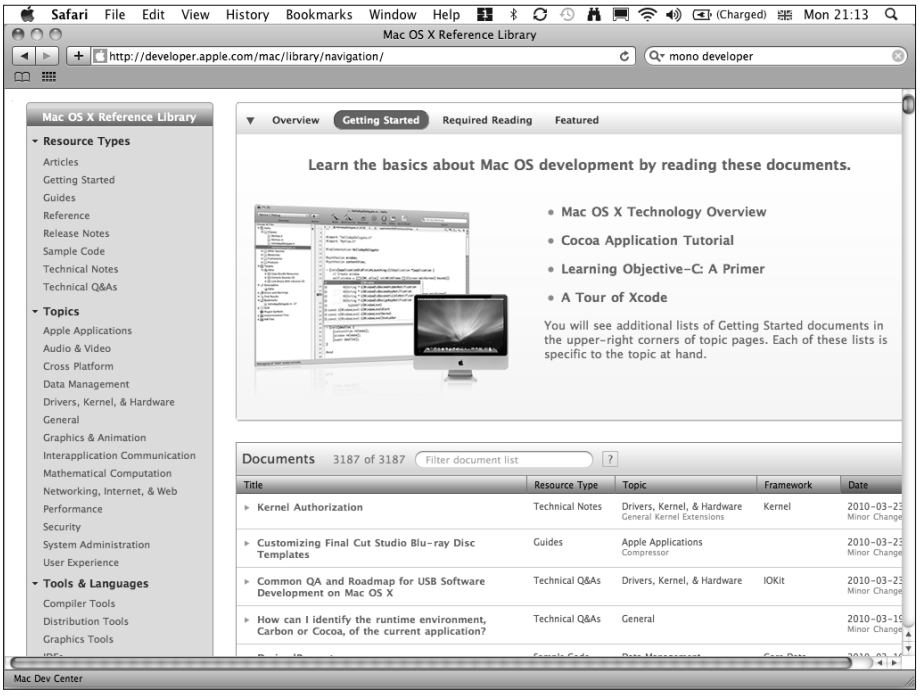
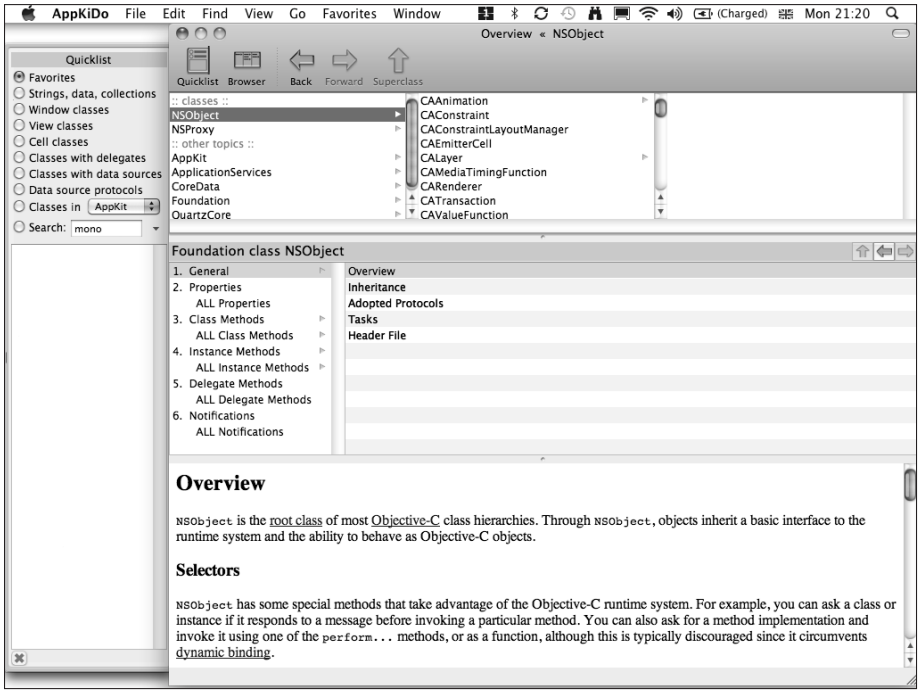


Figure 3.2

AppKiDo reads the documentation content and presents it in a more accessible form, with faster and more efficient class reference browsing. Unfortunately it only includes the Cocoa layer, not the C function layers.

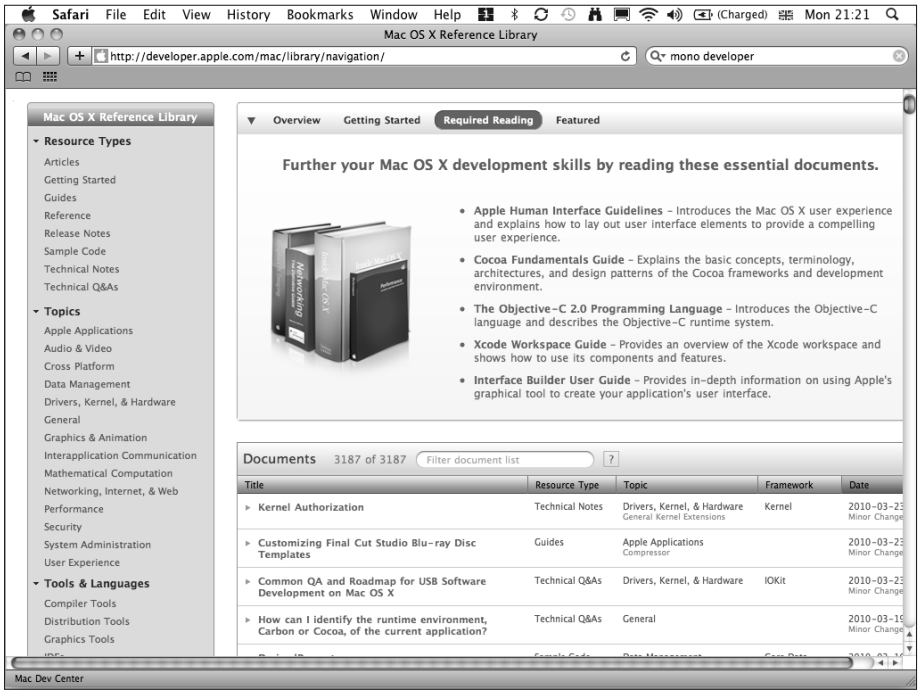


Getting Started with the Documentation

The most important window is at the top right. In Xcode 3.2 it shows four different panes: Overview, Getting Started, Required Reading, and Featured, illustrated in Figure 3.3. The arrangement of the windows and features changes between updates, but you'll usually find elements that are recognizably similar.

Figure 3.3

Required Reading is the best place to begin — specifically with the Cocoa Fundamentals and the introductions to Xcode and Interface Builder.



Although Getting Started is visible by default, the Required Reading section is a better place to start. The guides to Xcode and Interface Builder are essential reading, and the Cocoa Fundamentals Guide is a good overview, although some elements are pitched at a high level of abstraction and may be difficult to follow at a first reading. The Objective-C 2.0 reference is comprehensive but terse, and is best used as a reference rather than as an introductory guide.

The Apple Human Interface Guidelines are a mix of theory and practice. The most useful section is Part III: The Aqua Interface. This has very detailed information about interface elements, including mouse cursors, function keys, key shortcut selections, and visual design standards. You should skim this section immediately and refer to it again when you begin designing an

application. You'll also find it helpful to review the interfaces of selected Apple and third-party applications. Not all of the guidelines are obvious. Even if you're an experienced Mac user, you'll find design suggestions and elements that you may not have consciously noticed before.

The Featured section typically includes two or three selected new articles, often about recent OS updates. The content is aimed at experienced developers and may mention any framework in any layer. It's best to ignore these articles when starting out; they may become more useful when you've gained more experience.

TIP

The iPhone documentation follows a similar format. There's less of it, because Cocoa Touch is smaller and simpler than Cocoa. But the outline design of both platforms is similar, and you can follow the same strategies to use the documentation efficiently.

Understanding resource types

The rest of the documentation is divided into broad resource types. Although you'll spend most of your time looking at the framework reference pages, it's worth taking a few moments to explore the other resources so you can begin to familiarize yourself with the useful resources, while also noting the less useful ones.

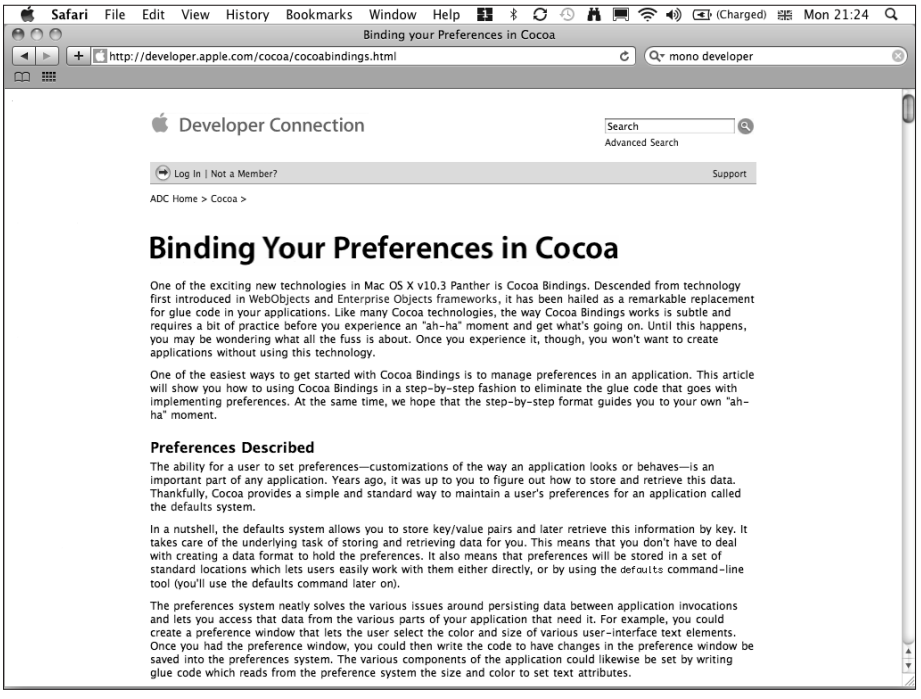
Articles

Articles hold a grab bag of miscellaneous essays and features, such as the Binding Your Preferences in Cocoa introduction shown in Figure 3.4. Many articles are highly specialized and of little practical interest unless you're looking for information about a very tightly focused topic; in short, you can ignore them. For newcomers to Cocoa, the most useful articles are as follows:

- Binding Your Preferences in Cocoa
- Getting Control with Subversion and Xcode
- Installing your Application on Mac OS X: Guidelines for Developers
- Maximizing Mac OS X Application Performance
- Optimizing with Shark 4 — a series of three articles

Figure 3.4

While most articles are irrelevant, a handful, such as this introduction to Cocoa Bindings, are essential reading for developers at every level.

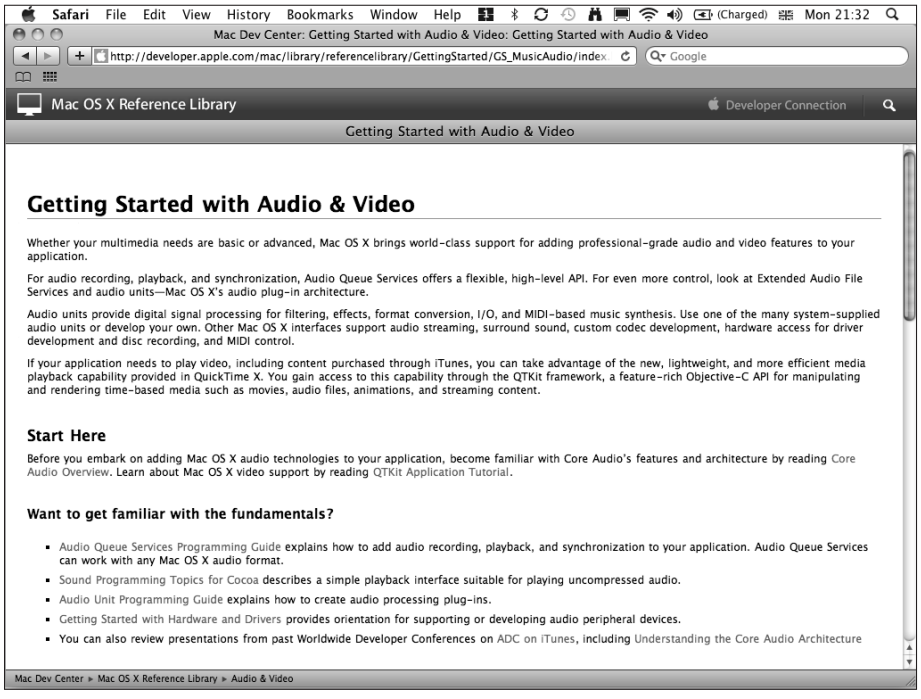


Getting Started

Getting Started includes a list of introductory guides. Each guide is split into fundamentals, examples, and an in-depth collection of links to more detailed information. The Getting Started guides can be useful, but most of the content is a very minimal orientation summary followed by links to in-depth programming guides and to sample code. The Getting Started with Audio & Video guide shown in Figure 3.5 is typical; it lists possible applications, but doesn't explain the relationship between key features, classes, and frameworks. If you're getting started, you won't necessarily want to start here.

Figure 3.5

The Getting Started with Audio & Video guide is typical of the other Getting Started articles. It's a list of links and examples padded with a short introduction, and not a true overview or orientation.



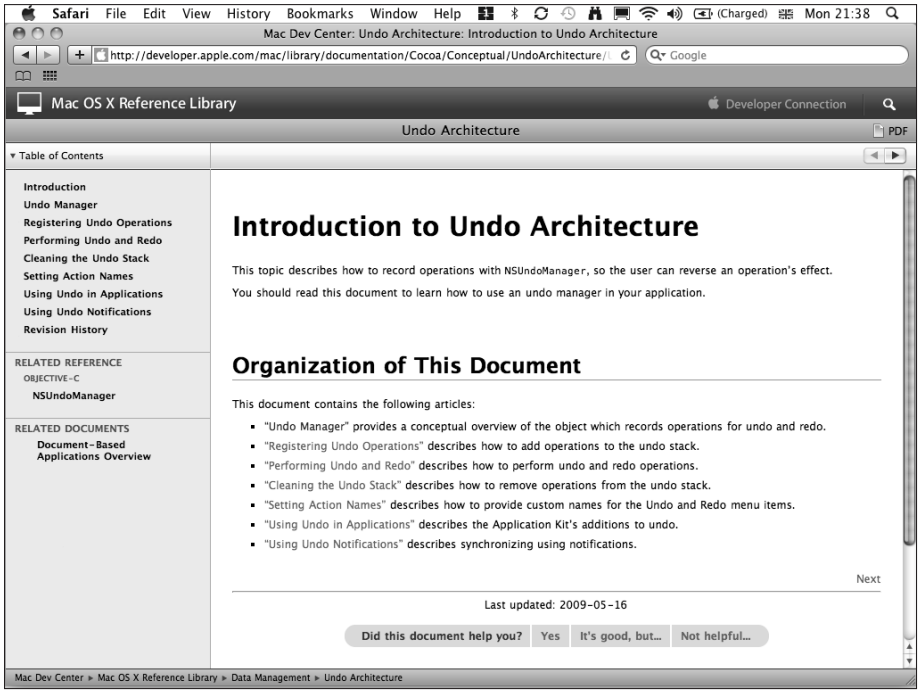
Guides

Guides include information about the coding interfaces to various OS X features. A small number of user interface and style guidelines also appears in this section. Guides vary in quality. Some are detailed and include source-code snippets that you can copy and paste into your applications. Others are very bare single-paragraph outlines of complex features. Only a small number are immediately useful when you're starting out. The rest detail the APIs of extremely specialized features that are rarely used.

Essential guides include the String Programming Guide, the Error Handling Programming Guide and the Undo Manager Architecture Guide, shown in Figure 3.6. You don't need to assimilate the contents of these guides immediately, but you should be aware that they exist so that you can use them as reference material when you're working with a specific Cocoa feature.

Figure 3.6

Guides typically follow a similar format — a short introduction, followed by a selection of mini-articles with more detail. Sometimes the mini-articles include sample code that you can copy and paste into your application.



Reference

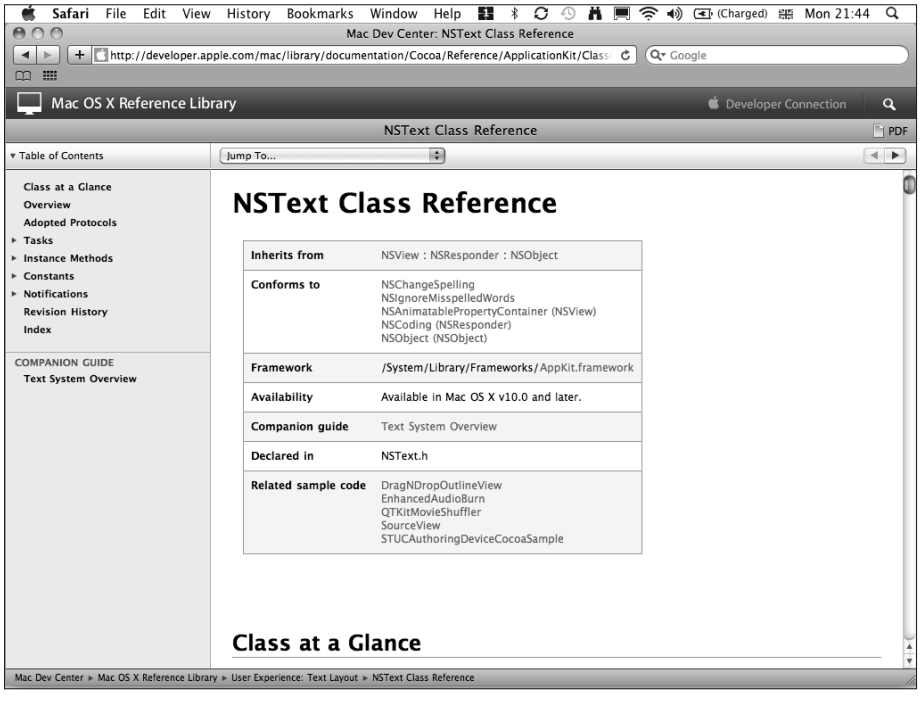
The Reference section takes up the bulk of the documentation. Once you've assimilated the introductory guides, you'll spend most of your time looking at these reference listings. There are two types of reference articles. A handful provide general reference material about broad topics, such as detailed compiler settings. Most are code references, with formal lists of properties/variables and code interfaces. This group includes Objective-C object references, and C function and struct references for all layers of OS X. Code references are grouped into the layers introduced earlier. The class references follow a fixed format, part of which is shown in Figure 3.7. A small number of class references include an extra Class at a Glance overview summary. The Table of Contents can include all or some of the following sections:

- The Overview is a short text article that sketches the function of the class and how it should be used.
- The Tasks section provides a plain list of methods grouped by function. Each method is a link — you can click it to display more information.

- The optional Properties section lists the class properties and briefly sketches their features. Not all classes include a Properties section.
- The optional Class Methods section lists class methods in more detail, with a sketch of their features and functions.
- The Instance Methods section lists the instance methods, using the same format as the Class Methods.
- A final optional section lists other information and may include constants, further optional methods, or information about notification messages generated by the class.

Figure 3.7

The Class References are long documents. The easiest way to navigate them is by using the internal links, to skip forward, and the list of contents at the top left — rather than the back button — to move back to a section header.



CAUTION

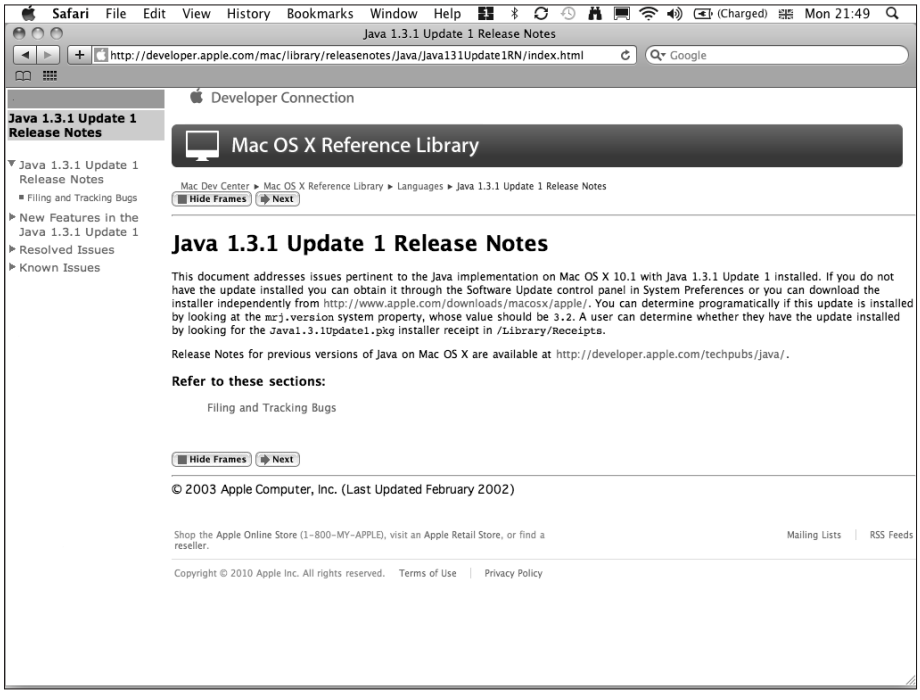
One of the less friendly features of the Class References is the linking. When you click a link, it takes you to a position on the same page. You'll almost certainly click the browser's Back button when you've finished — and it will take you to the previous page you viewed, not back to the original location. There's no fix for this, and you'll have to keep reminding yourself not to do it.

Release Notes

The Release Notes are another grab bag of short articles. Figure 3.8 shows a typical example. The notes list updated properties, constants, and new classes added in each release of OS X. This information is included for completeness rather than for reference. Very occasionally you may find that code isn't working because information listed here hasn't been included in the main documentation. More typically, the main class references are updated as soon as changes are made, and the release notes provide a sporadically valuable change log.

Figure 3.8

It's possible some developers may still need to see the original Java 1.3.1 release notes from 2003, but it's not likely.



Sample Code

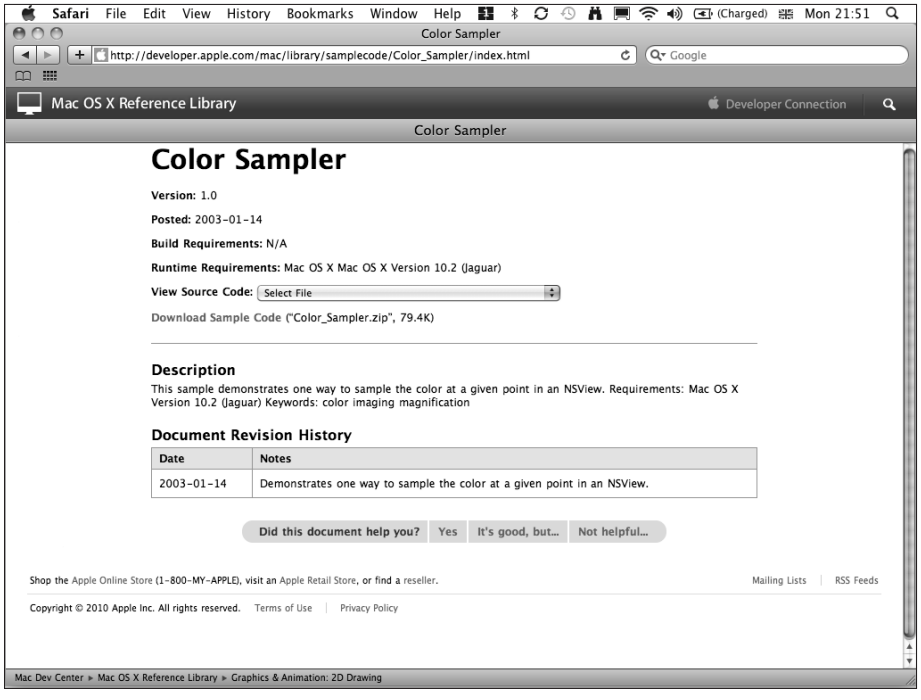
The Sample Code, shown in Figure 3.9, is one of the most useful features of the documentation, and includes a collection of mini-applications with full source code and project settings that you can use as worked examples. The sample code is tightly linked to the Xcode SDK. You can save an example to a project folder, open it in Xcode, compile it, and run it with just two or three mouse clicks.

TIP

The Sample Code resource isn't a complete and definitive list of all available sample code. You can find further code samples online; Apple's online sample code library doesn't always match the library in the documentation. Some of the framework reference documents list further sample code that doesn't appear in this section.

Figure 3.9

In the internal version of the documentation included with Xcode, the Download Sample Code link is replaced with a button that automatically loads the project into Xcode. Online, you must download the project file and unzip it manually before opening it.

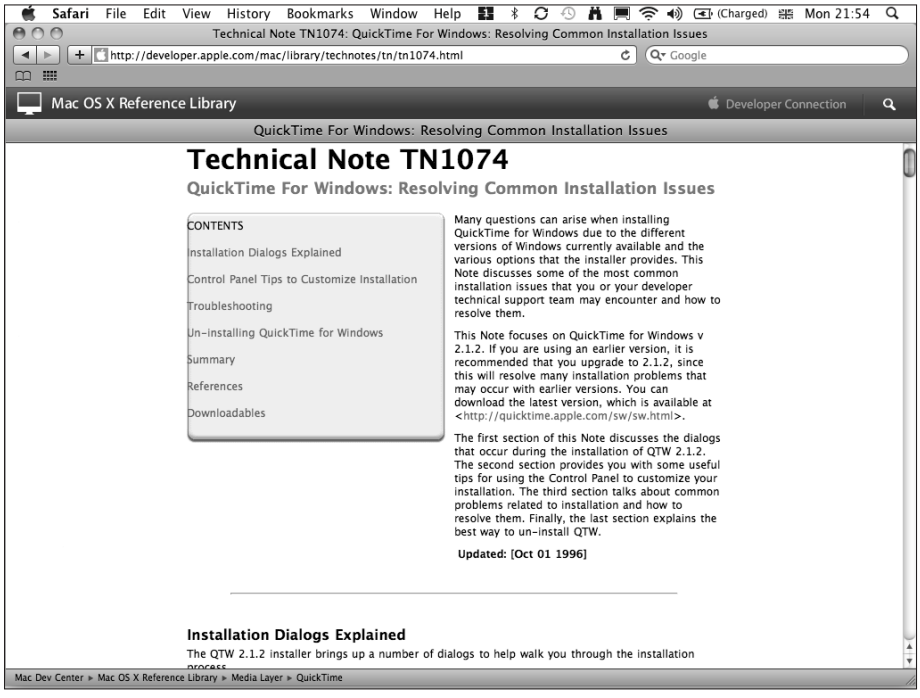


Technical Notes

The Technical Notes section, shown in Figure 3.10, is another grab bag of short essays and mini-features. Many of these items are of historical interest with no useful content. You can find essays here that refer to Mac OS 6, 7, 8, and 9 and mention features and code interfaces that have long been superseded, such as the QuickTime for Windows article in Figure 3.10, which was written in 1996. A small number of articles, such as the New HID Manager APIs feature, may be useful to current developers. Unfortunately, some articles are so out of date, their content is misleading. It can be useful to skim this section for interest, but most of the contents can be ignored — unless you're an IT historian.

Figure 3.10

Some of the Technical Notes are more useful for their nostalgic value than their practical content.

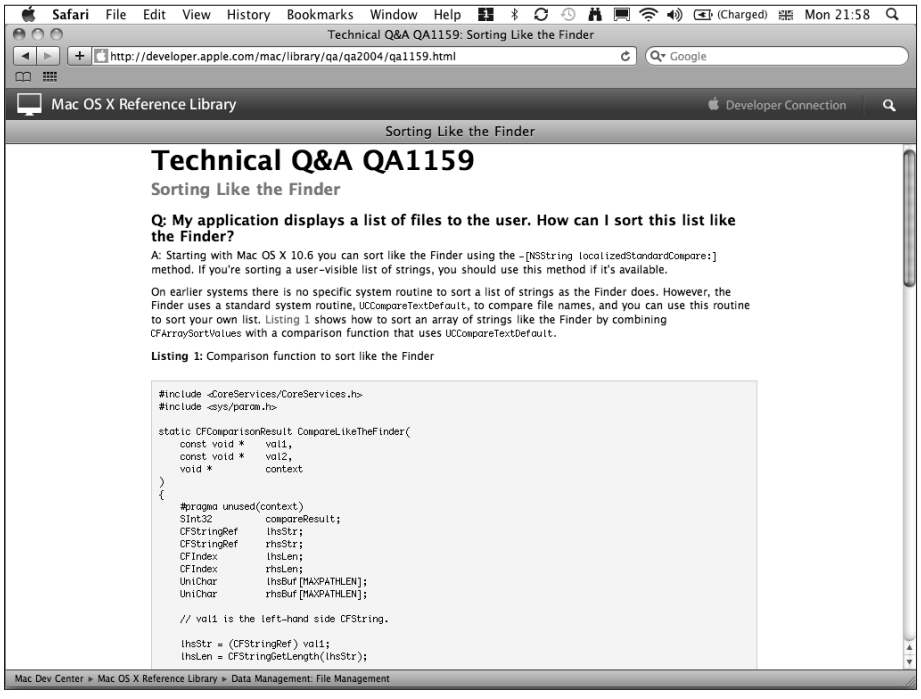


Technical Q&As

The Technical Q&As section is another miscellaneous and somewhat random collection. It concentrates on bug fixes and minor development notes. It includes some useful content, but it also includes a generous selection of ancient articles that are redundant or irrelevant. The most relevant items are the most recent. As a rough guide, items dated from 2000 onward may be useful, such as the Sorting Like the Finder feature, shown in Figure 3.11. Earlier content is very likely to be outdated.

Figure 3.11

The Technical Q&As are more up to date than the Technical Notes. The information about sorting files for display shown here is still current.



Understanding Topics

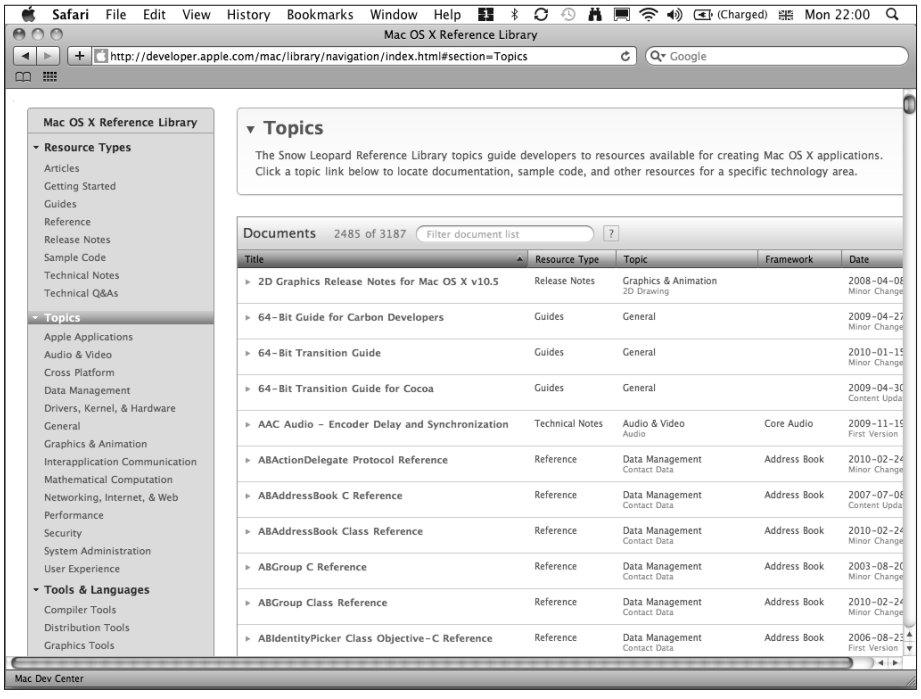
The Topics list, shown in Figure 3.12, is an attempt to group resources by subject. In theory, selecting a subject shows a list of all the relevant articles in the documentation; for example, selecting Audio & Video displays a list of all the documentation resources that may be useful when developing audio and video features.

Navigating the Topics

In practice, because the resources themselves include old and out-of-date information, it can be difficult to make sense of the topic lists. One of the essential features missing from the documentation is prioritization. Features that are essential for beginners are given the same prominence as highly specialized development notes from the mid-1990s. This makes it difficult to find key points. While the Topics breakdowns can be useful, they're an indexing feature and not a useful or complete summary of everything you need to learn to master a topic.

Figure 3.12

The Topic breakdown is the best way to view everything related to a topic, although many of the items that appear here are redundant or irrelevant.



Reading about Tools & Languages

In spite of the name, the Tools & Languages section doesn't include definitive reference information about tools and languages. Instead it supplements the guides in the Required Reading section, adding extra detail and introducing noncore features. For example, you'll find a guide to Quartz Composer, Apple's visual animation and video processing tool. Quartz Composer is a useful and entertaining feature included with Xcode, but you don't need to use it or know about it to create working applications.

It's worth looking through this section because it contains some useful content, including information about using features of Xcode that are hidden behind its GUI (graphical user interface). You can use this content to customize and optimize Xcode if you choose to. Initially, you can treat this section as a collection of extra material and not an essential reference.

Using the Documentation

When you're starting out, the documentation has a poor signal-to-noise ratio and may try to distract you with irrelevant content. This becomes less of a problem as you gain experience, because you'll assimilate the information in the Getting Started guides or find equivalent help elsewhere. You'll also start to remember the frameworks and objects that are used regularly and to memorize some of their features. You can improve your productivity by using the documentation methodically. Random browsing isn't recommended. A more focused approach will help you find useful details more reliably.

Sorting the documentation

Sorting is a key skill when using the documentation. The listing window has five headings: Title, Resource Type, Topic, Framework, and Date. Clicking each heading sorts the contents of the window alphabetically, except for the Date heading, which sorts chronologically.

TIP

The Framework heading doesn't appear when you're viewing the framework reference documentation because it's redundant.

Clicking Title is likely to display a jumble of unrelated features. For a more useful summary, try clicking Topic, Resource Type, or Framework. When viewing the grab-bag sections — Technical Notes, Technical Q&As, Release Notes, and Articles — it can also be useful to sort by Date, to eliminate irrelevant older content.

TIP

It's a good idea to check the date of any resource before using it, because some of the documentation is simply outdated and wrong. There's no formal sell-by date, but as a rule of thumb, any item that's more than five years old may not be accurate.

When clicking Topic to sort contents, the headings include extra subgroups that aren't visible in the main Topic listing. For example the catch-all General topic includes a Memory Management subsection. You can use this feature to find items that you might otherwise miss.

The sorting algorithm groups items by subresource — topic or resource type — and then sorts them alphabetically within each group. The list of titles can appear chaotic, but in fact it's organized so that related items are shown next to each other. One confusing quirk is that in a Topic sort, General items appear at the top of the list. The remaining items are sorted alphabetically in the usual way. A useful search sequence for finding the classes in a framework is

```
Frameworks List ➪ <named framework> ➪ Resource Type ➪
   Reference
```

Working with source code

Source code can be at least as useful as reference documentation. Sometimes you can solve a problem or implement a feature by copying code, even if you don't fully understand how it works. This is an unapologetic cheat, and you should aim to understand the code you use. But even when you don't, it's an approach you can use to gain experience. Figure 3.13 shows my suggested strategy.

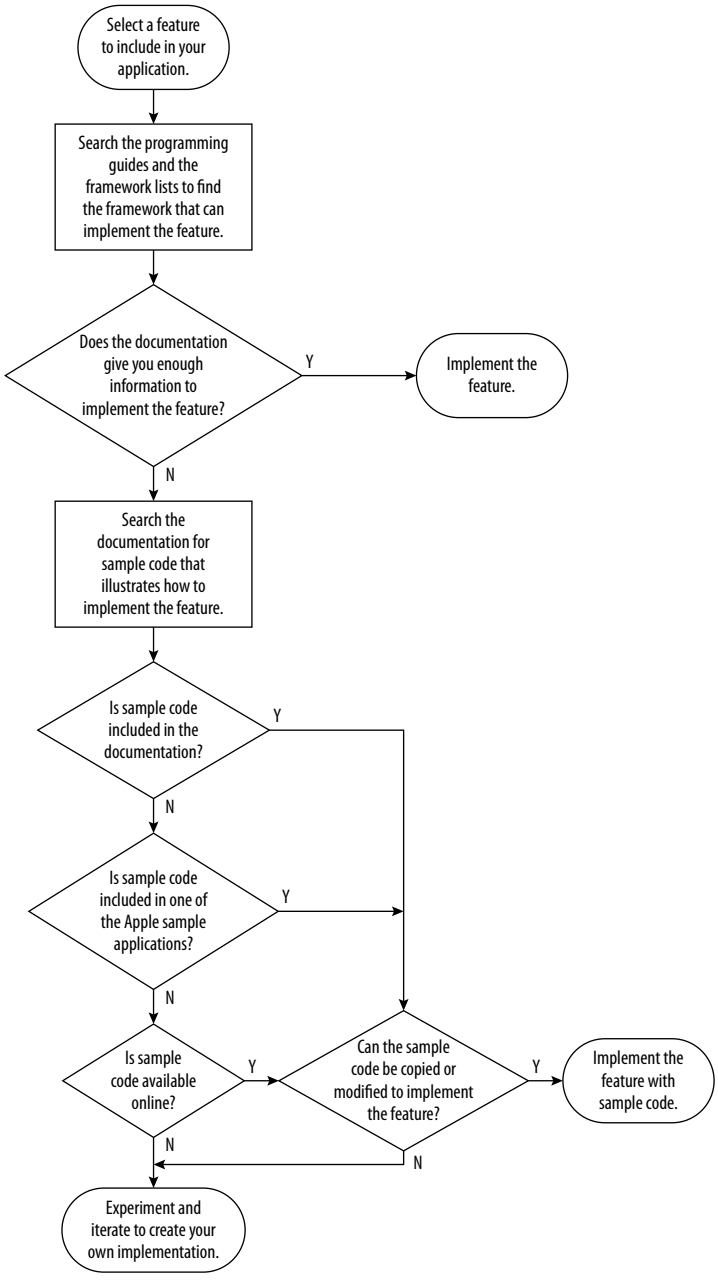
Apple's code samples are often quite dense. It's unlikely you'll be able to use them directly, but stripping out code that you don't need can be a good way to learn about application design. Don't forget that at least some of the application's architecture is defined by its nib files, so you'll need to explore those too.

Summary

This chapter introduced you to the developer documentation. It explained the differences between the various documentation resources and included practical hints for finding the most useful content. It also included some suggestions for working with the documentation as a whole and for making the best use of other online resources and of source code created by Apple and by other developers.

Figure 3.13

My suggested flowchart for the documentation concentrates on finding useful code. When you're staring out, sample code and class references are the most useful resources.



4 Getting Started with Xcode

In theory, Objective-C is as platform-independent as C++, Java, Ruby, and Python. In practice, it isn't. Most developers write code that combines the features of Objective-C and Cocoa, using them as an informal blended language that runs almost exclusively on Apple hardware. Only one toolkit supports this approach — Apple's Xcode SDK (software development kit).

Xcode includes dedicated class and object management features that aren't available in other environments. Developers rely on these features to add new classes to their applications and to define the list of objects that's loaded when an application runs. While it's possible to create working Objective-C code with a text editor and a command-line compiler, this isn't an efficient way to work. It doesn't take advantage of the helper features in Xcode, which make it easier to work with Cocoa objects.

The next few chapters introduce Cocoa object creation and class management in a theoretical way using working example code created in Xcode and Interface Builder. They also outline some typical design processes and illustrate how to add specific features to an application, translating details in the developer documentation into working code.

4 In This Chapter

Getting ready for Xcode

Installing Xcode

Creating a new OS X project

Exploring Xcode's windows

Getting Ready for Xcode

Until the announcement of iPhone iOS 4 in April 2010, Apple offered two versions of Xcode. Mac developers could use an OS X-only version of Xcode, while iPhone developers were supplied with a combined version with templates and documentation for both OS X and iPhone OS projects.

Apple no longer supports the OS X-only version, and all developers must use the combined SDK. This encourages developers to explore both platforms, but the combined SDK is a much larger download — up to 5GB versus 800MB of the OS X-only SDK. After the download completes, it can take three hours to install the combined SDK on the oldest and slowest Intel Macs. An hour is more typical on recent hardware.

Alternatives to Xcode

If you don't own a Mac, you can use GNUstep to experiment with Objective-C and to develop very simple Cocoa applications. This is a valid option for Windows and Linux users, but unfortunately GNUstep is a poor substitute for Xcode. It lacks the C function libraries that support Cocoa on the Mac, it doesn't include any of Cocoa's more recent classes and features, and it doesn't implement the Aqua interface or most of the OS X layers and frameworks. Without Aqua, GNUstep applications have the look and feel of NeXTStep applications from the early 1990s, and there's no support for some of the recent core Cocoa features.

Xcode includes a sophisticated integrated editor, while GNUstep uses the free jEdit editor. Both environments rely on a version of the GCC (GNU Compiler Collection) compiler, but there are subtle differences between the two versions. Code is approximately compatible, but not all of Objective-C's features are implemented in an identical way.

Compared to Xcode, GNUstep is a very limited environment. Some of the examples in this chapter are compatible with GNUstep, but most of the code in later chapters isn't. If you have no other option or if you want to try out some very simple examples, you can use GNUstep to get started with Cocoa. But if you want to develop applications for other users, either for fun or for profit, you'll need a Mac and a copy of Xcode.

OS X developers can create working applications in alternative environments such as Mono. Mono is an alternative set of libraries that have been partly reimplemented with Cocoa elements. But code written in Mono doesn't call on Cocoa's features directly. Mono is a valid alternative for simplified cross-platform development, but it doesn't support many standard Cocoa and OS X features and is an inefficient choice for high-performance applications, or for applications that rely heavily on the Cocoa and OS X media and animation features.

iPhone OS developers have always had more limited choices. Because Objective-C and Cocoa Touch are relatively complex and programmed at a low level, various simplified alternative SDKs and frameworks have appeared, such as Cocos2D, Flash for iPhone, and Ansca Corona. Projects built with these SDKs have been accepted in the App Store.

At the start of 2010, Apple announced a new Xcode-only policy for iPhone apps. Only Objective-C, C, C++, and native Apple Java apps are allowed. Alternative SDKs and frameworks are banned from the App Store. It's not clear how rigidly this policy will be enforced, or whether previously accepted non-Xcode apps will also be banned. But developers should keep in mind that Apple is emphasizing that Xcode and Objective-C are the definitive development environment, and attempting to use an alternative for commercial projects is now very risky.

TIP

If you bought a Mac before Q2 2010, it's likely that you'll have the OS X-only version of Xcode on the OS support DVD bundled with your system. It may not be up to date, and some features may not be an exact match for the descriptions in the rest of this book, but if you're experimenting rather than creating code for release, you can use this version to bypass developer signup and a subsequent download. Look for the `Xcode.mpkg` file in the Optional Installs folder. You can update this version later, if you choose to.

Although Xcode runs on any Intel Mac from a Mac mini upward, compilation is a processor- and disk-intensive process. OS X development is demanding, and the faster your Mac runs, the less time you'll spend waiting for each build. This is less true of iPhone apps, because they're simpler and more compact. After an initial compilation run, recompiling an iPhone OS app after a minor edit can be almost instant, even on slow hardware.

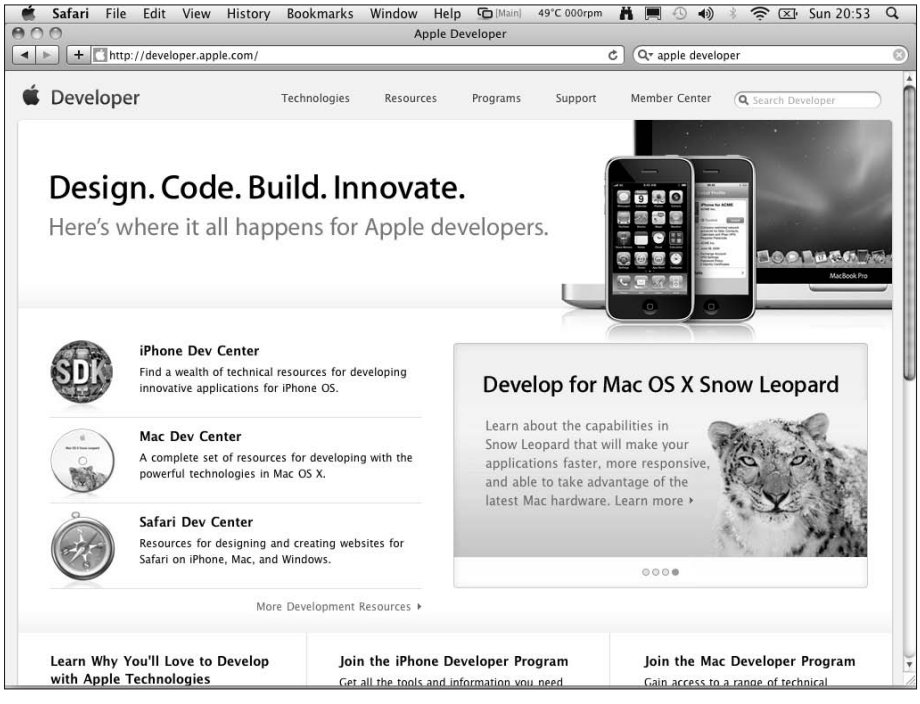
It's useful to have as much screen space as possible. A dual-monitor system is extremely helpful, and a triple- or quad-monitor system is better, although a single large monitor is also viable. The ideal minimum configuration can display a code window on one monitor and documentation on another, which also doubles as a scratchpad for testing and debugging. It's possible to work with a single low-resolution screen, but Xcode is designed to show multiple windows simultaneously. Switching between windows is distracting and takes valuable time, significantly degrading productivity.

Registering as a developer

Although Xcode is free, you'll need to register as a developer to download it. The sign-up process is straightforward, but you'll need a valid e-mail address for confirmation. Begin at `http://developer.apple.com`, shown in Figure 4.1.

Figure 4.1

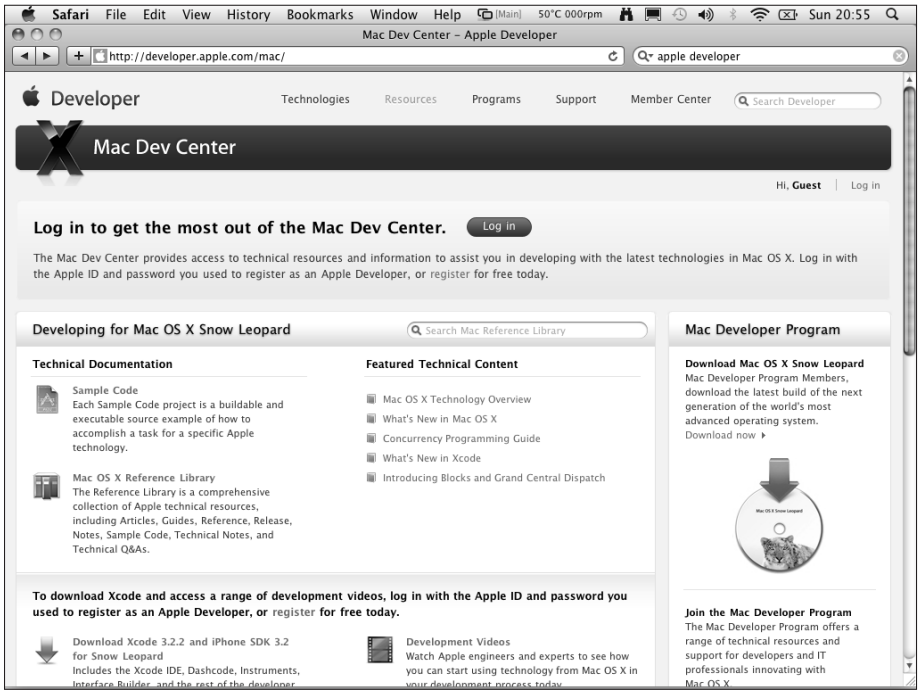
You do not have to register to view Apple's developer welcome page. The design and layout are updated regularly; this version shows Mac OS X Snow Leopard.



Click the Mac Dev Center link to open the Mac Dev Center, shown in Figure 4.2. You can see the download link at the bottom left, but the design of this page changes regularly so it may be in a different location on the page when you view it. The key words to look for are Xcode and SDK.

Figure 4.2

You are not required to register to view the technical documentation on the Mac Dev Center page. However, to download code, you must sign up.

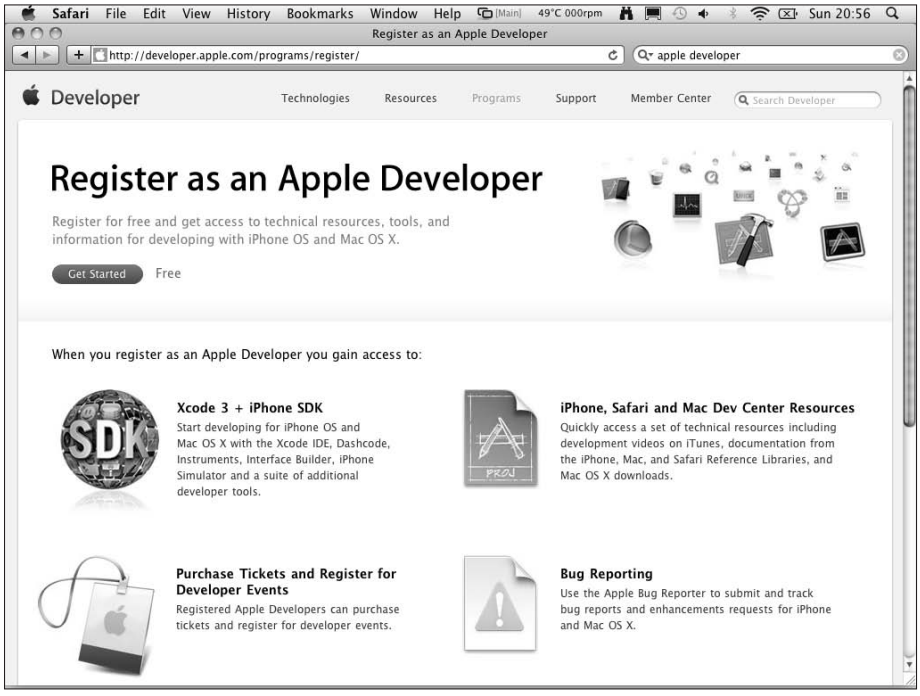


Clicking the download link takes you to the sign-up page shown in Figure 4.3. This page is also redesigned regularly, so it may look different when you view it. Click the Get Started button on the left to begin the registration process.

Figure 4.4 shows the first page of the sign-up process. If you already have an Apple ID that you use with another online Apple product, you can also use it as a developer ID. If you plan to develop commercially on the iPhone, create a new ID here because it will simplify accounting and sales reporting later.

Figure 4.3

The sign-up page illustrates the benefits of registration. Basic sign-up is currently free.



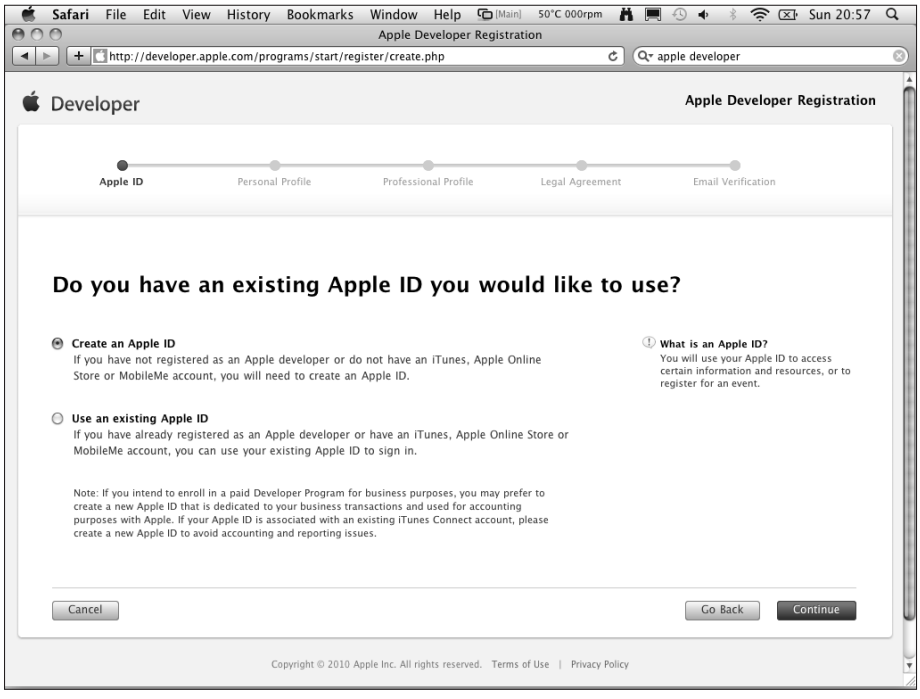
The rest of the sign-up process is straightforward. After specifying an ID, entering your contact details, answering a few simple questions, accepting the Developer Agreement, and processing an e-mail confirmation, your developer account is created. You can then log in again and return to the screen shown in Figure 4.2. The download link is now active and you can begin downloading the SDK.

CAUTION

If you have a slower broadband connection, you may want to leave the download running overnight. It takes around six hours at 1MB/second. On dialup, it can take most of a week to download the file, so the most practical way to get the SDK is to ask a friend with broadband to download it, burn it to a DVD, and post it to you. Apple doesn't sell DVD copies of Xcode. Because access to the developer documentation is being moved online, dialup is becoming an increasingly impractical option.

Figure 4.4

You can use an existing Apple ID to gain initial access to the developer program, and re-register later with a new commercial ID if you plan to release iPhone apps.



Joining the Mac Developer and iPhone Developer programs

Signing up as a free developer gives you entry-level access to the Developer Program. A further level is available to developers willing to pay an annual fee. In 2010 Apple revamped its Mac Developer Program, reducing the fee to $99/year from $499 but eliminating a number of benefits. In the older program, developers were given a discount on Mac hardware and could also send applications to Apple's Compatibility Testing labs for hardware testing. More expensive tiers offered "free" tickets to the Apple Developer Convention and further discounts.

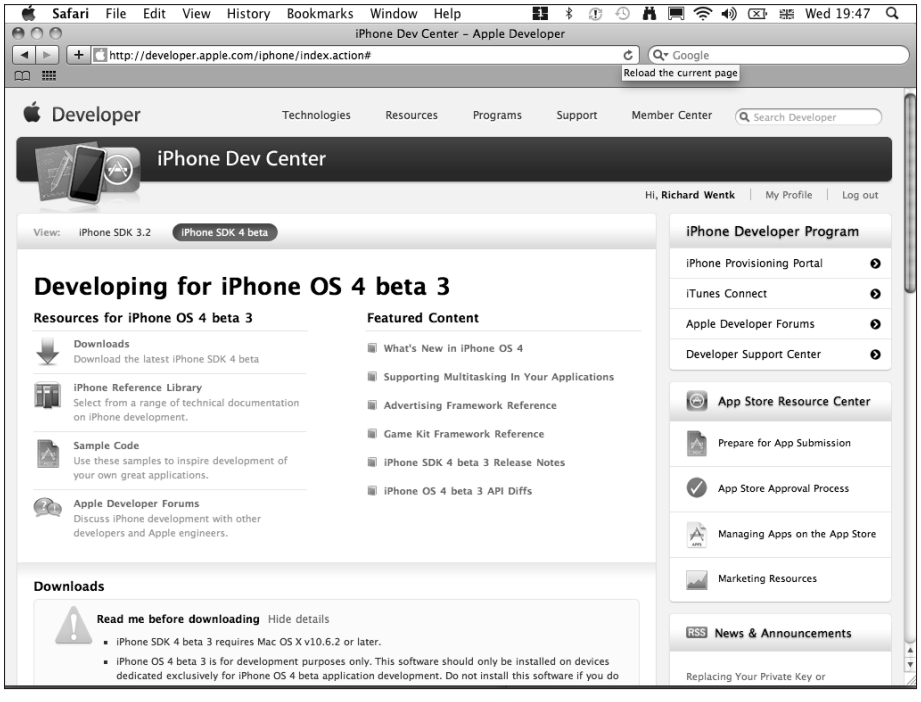
In the new program, the chief benefits are a free copy of the current version of OS X and access to advanced beta — "seed" — versions that can be downloaded in the months before the official public launch of an update. Developers can request two technical support incidents per year, where they discuss development problems with an Apple engineer who reviews their code and offers troubleshooting advice or a workaround. Access to the Apple Developer Forums and to a selection of instructional videos is also included.

If you're experimenting with development on the Mac, you can ignore the Developer Program. None of the benefits are indispensable. Mac applications can be built, run, given away, and sold successfully without them.

The equivalent iPhone Developer Program offers a more useful selection of benefits and is recommended for any serious developer. The annual fee is the same, $99, and developers receive two technical support incidents, access to the forums and videos, and advanced copies of seed releases of Xcode that support forthcoming updates, as shown in Figure 4.5. The iPhone Developer Program also unlocks the hardware testing features built into Xcode. Until this is enabled, apps must be run in a Simulator application with limited emulation of the hardware features of an iPhone, iPad, or iPod touch.

Figure 4.5

During OS seed periods, beta updates are released every fortnight or so. Updates are only available to developers who have paid the annual Developer Program fee.



The program also controls access to the App Store. Only paid-up developers are allowed to sell apps in the store. The iPhone Program is so tightly controlled that apps built in Xcode and installed on a handset expire automatically after a few months. Developers can run a build of their code on their own devices for a limited period. When this expires, the app stops working and it must be rebuilt and reinstalled. An app can only be installed permanently by submitting it to the App Store, waiting for it to be approved, and buying it.

Currently corporate and business developers can sign up for two alternative corporate programs that cost $299/yr. Only businesses with more than 500 employees and a DUNS number — an international business identifier — are eligible. The Enterprise iPhone program allows in-house distribution and development of iPhone applications, bypassing the App Store. Because Mac development is unrestricted, there is no equivalent Enterprise Program for OS X.

Installing Xcode

After downloading the Xcode `dmg` file, double-click it to open it. Installation is conventional, with few surprises. Double-click the `Xcode.mpkg` file in the `dmg` folder and follow the prompts. The SDK is designed to support multiple versions of both OS X and iPhone OS. You can select these using the dialog shown in Figure 4.6. There's no need to support older versions of either OS, but you can install them for compatibility testing.

NOTE

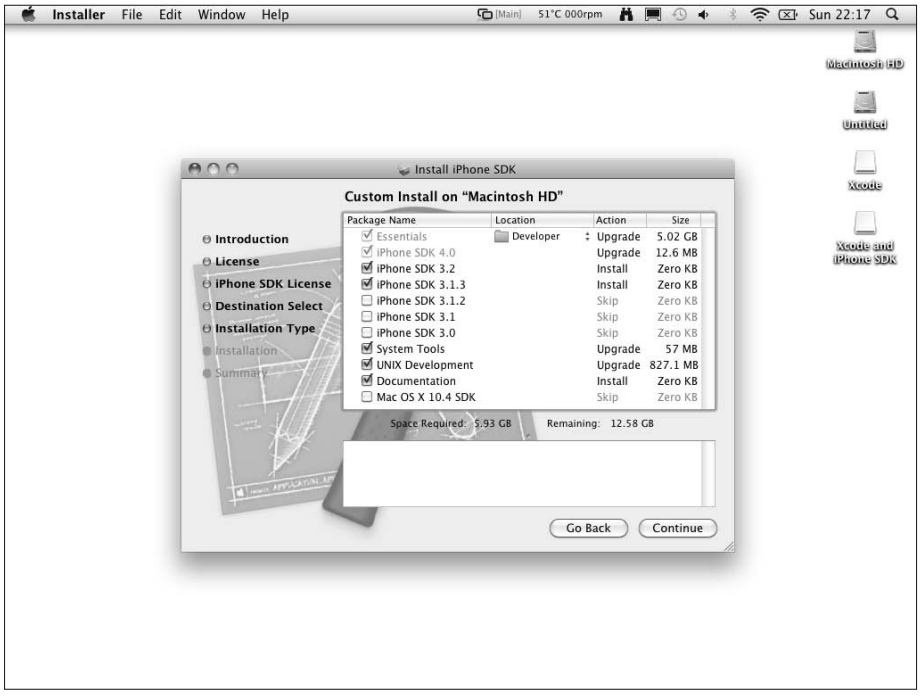
You can use this dialog to install an additional set of Unix command-line developer tools by checking the UNIX Development box. This feature is optional — it's not used by Xcode and isn't required for Cocoa development. It's useful if you're familiar with Unix development and want to use a traditional collection of command line compiler tools, libraries, and man pages on a Mac. The tools are installed into `<usr>`.

By default Xcode is installed in a new `Developer` directory, shown in Figure 4.7, which is placed in the root directory of your Mac's hard disk, not in a user directory. `Developer` is best left where it is — moving it may break the installation. For quick access, it's useful to add `Developer` to the Places list in Finder.

You can view the applications included in Xcode by opening the `Applications` folder. Xcode appears at the bottom of the list. You may want to add it to the Dock, together with Interface Builder. The other applications in this folder are used less frequently.

Figure 4.6

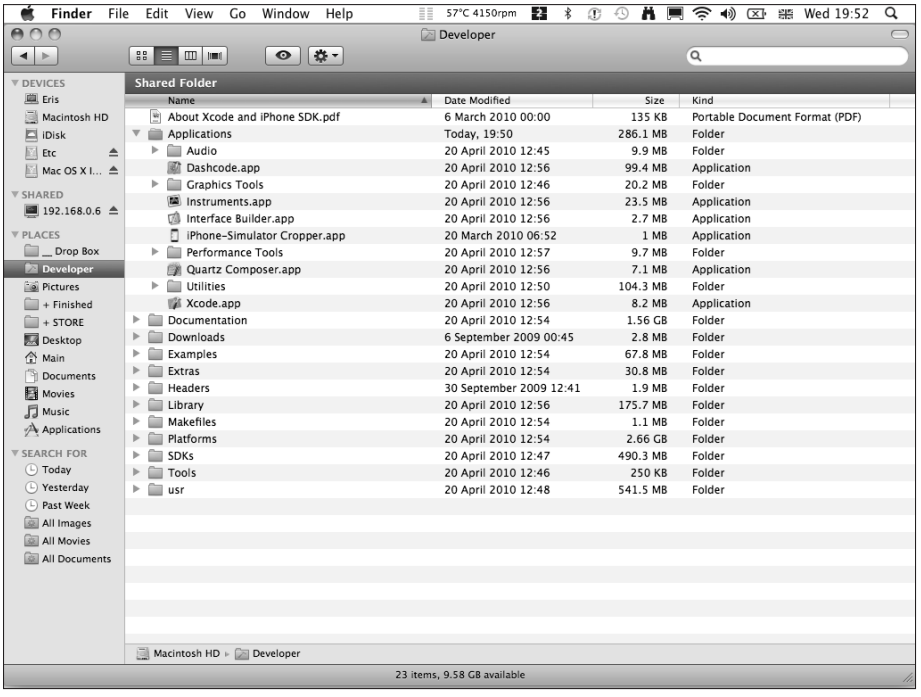
Older versions of both iPhone OS and OS X can be useful for compatibility testing, but most builds target the most recent OS.



It's an excellent idea to create a `Projects` folder. When you create a project in Xcode, it's automatically placed inside a new folder. Most developers generate tens or even hundreds of these folders, so you'll find it helpful to collect them into subfolders; for example, to keep trials and experiments distinct from commercial projects. A separate `Projects` folder also simplifies version control.

Figure 4.7

You can add your own Project folders (not shown here) to the contents of the Developer directory, and then divide projects further into useful subdirectories. Allow at least one folder for random experimentation.



Creating a New OS X Project

To create an application, double-click `Xcode.app`. Click the large Create a New Xcode Project button, shown in Figure 4.8.

CAUTION

This book was prepared with Xcode 3.2.3. A preview version of Xcode 4 was released just before the book went to press. Xcode 4 offers a faster compiler and also integrates Interface Builder, which no longer works as a separate stand-alone application. Otherwise the editing and building process for applications remains recognizably similar. For more information about Interface Builder, see Chapter 7.

Figure 4.8

The Xcode start-up screen displays a list of recent projects, if there are any. It's a good idea to review the Getting Started with Xcode tutorial for a quick overview of some of Xcode's features.

Xcode displays a template window, shown in Figure 4.9. The window shows a list of file and application templates supported by Xcode. Click Application in the template pane at the left of the window. Then click Cocoa Application in the template pane at the top of the window. Click the Choose button at the bottom right.

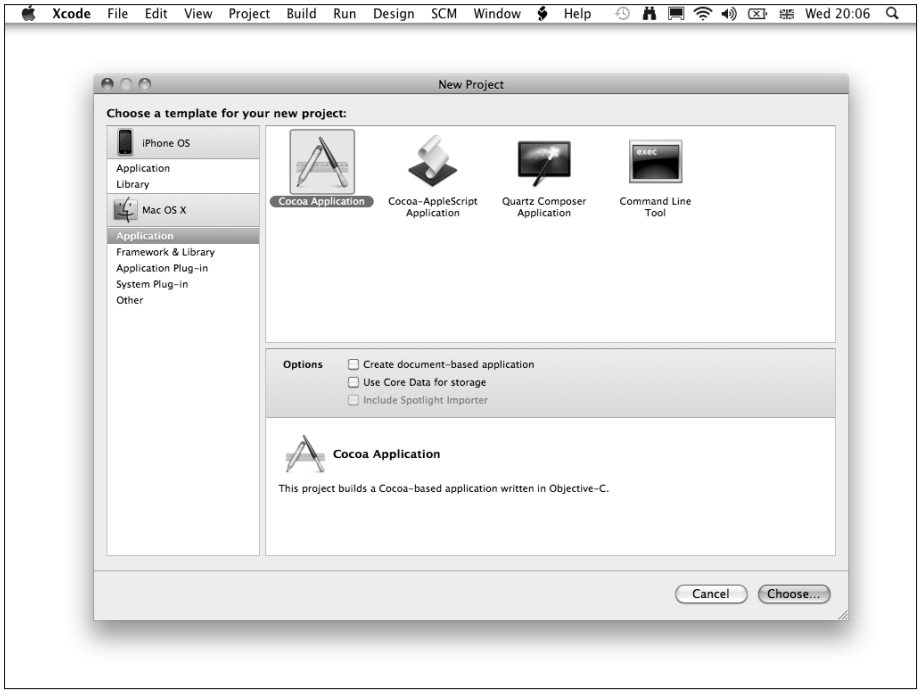
NOTE

If you're using the iPhone version of Xcode, you'll see an extra division at the top left for iPhone templates. Make sure you select the Application option under Mac OS X and not under iPhone OS — unless you want to create an iPhone app. iPhone and OS X templates have a different structure.

A File Selector sheet drops down from the top of the window. Type in a name such as First in the Save As: field at the top of the sheet, as shown in Figure 4.10. Keep the name short and avoid spaces: the name is used as a prefix for some of the files in the project. Using a long name here creates unwieldy filenames in the project. The name you enter also sets the application name.

Figure 4.9

iPhone OS offers a selection of project templates, described in Chapter 19. Cocoa applications start with a single common Cocoa Application template. The other templates are more specialized and used infrequently.

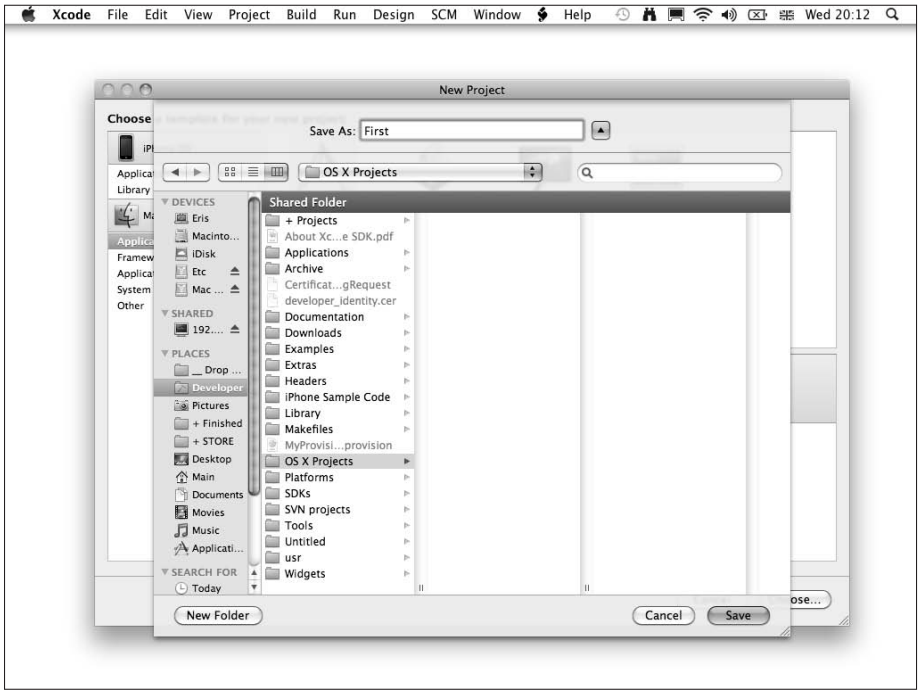


CAUTION

You can change the application name later if you choose to, but it's extremely difficult to rename project files. Xcode treats the build target, the application, the project name, and the project files as separate entities, with independent names.

Figure 4.10

You can save a new project anywhere on disk, but by default this dialog selects the Developer folder. It's efficient to create one or more project folders here rather than elsewhere on disk.



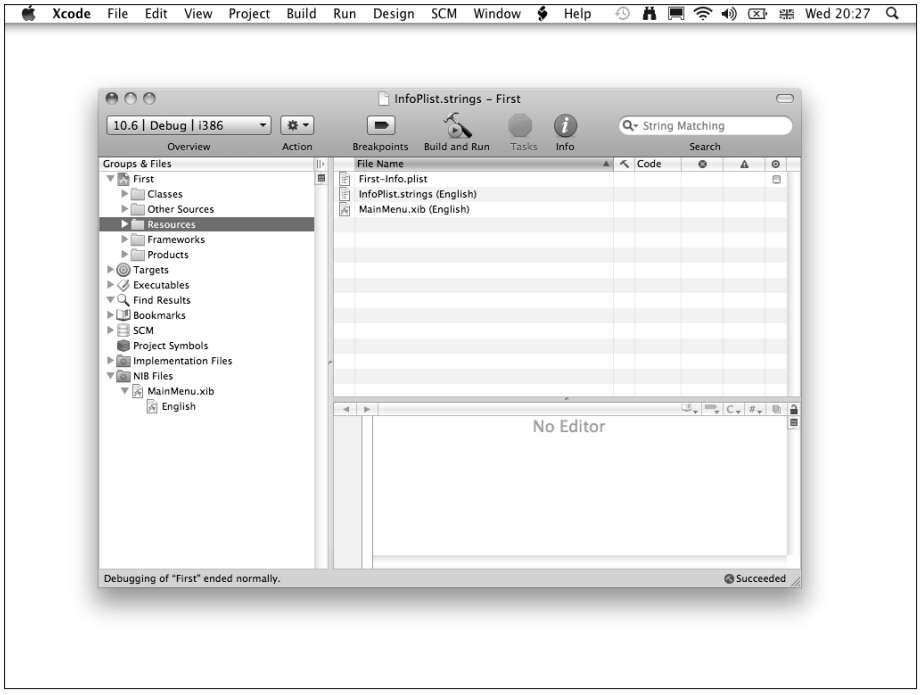
Xcode creates a new template with the name you specify and opens a new project window, shown in Figure 4.11.

NOTE

The project window isn't maximized when it appears. You can either resize it manually or click the green button at the top left of the window to maximize it. For monitors smaller than 26", it's easier to work with a maximized project. For larger projects, you may find it more productive to have more than one project or file window open at the same time.

Figure 4.11

When you create a new Xcode project, it appears in a single window that is a combined editor and file selector. You can have more than one project window open at a time, either switching between them manually or tiling them on a larger monitor for fast access.



Building a project is a single-click process. There's no need to define compiler options or to change any of the default settings. Click the Build and Run button at the top of the window. Xcode compiles and runs the application, creating a floating window and a new menu bar, as shown in Figure 4.12.

TIP

As the application compiles, it posts status information at the bottom left of the main Xcode window. Xcode is an incremental compiler — it only compiles files that have been added to the project or have changed. Depending on the speed of your Mac, it takes 15 to 30 seconds to build and run the template application for the first time. Subsequent edits build more quickly.

Although the application seems to be independent, it's running as a subprocess within Xcode. If you quit Xcode, the application is terminated. You can run an application independently by opening the `Build ➪ Debug` folder within its project folder and double-clicking the `.app` file.

This isn't usually helpful while coding and debugging. Xcode includes separate console and debugging windows, and they can only be accessed when the application is running within Xcode.

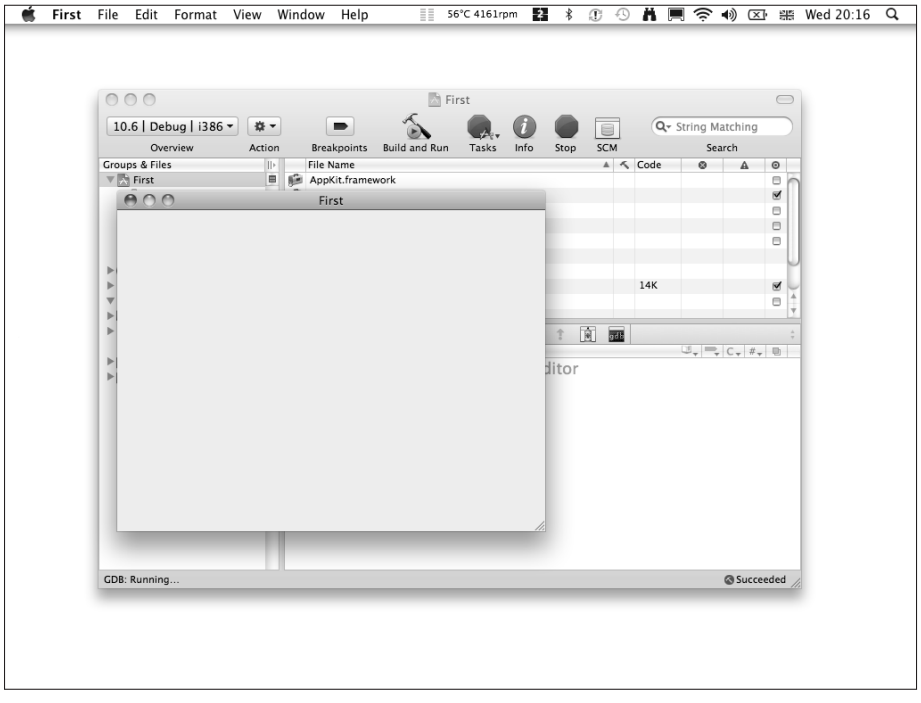
The application isn't complex. It displays an empty window that supports a basic feature set. You can resize the window by dragging the resize box at the bottom right. The red, orange, and green window buttons at the top left respectively close, hide, and restore the window in the usual way. The application adds itself automatically to the Dock when it runs, displaying a default application icon. To quit, choose First ➪ Quit First from the menu.

NOTE

The application isn't set up to quit automatically when you close the window, so closing the window leaves the menu in place and the application running. You can only quit from the menu.

Figure 4.12

The size and position of the default template's window are set within the application. By default, it appears on top of Xcode. Although the window is empty, it's fully functional. It can be dragged, resized, hidden, and so on.



Exploring Xcode's Windows

Now that you have created a first application, you'll want to take a closer look at Xcode. Xcode is a deceptively simple development environment with some unexpectedly rich and powerful features. Newcomers can literally build both Mac and iPhone applications with a single click, ignoring most of Xcode's features. More experienced developers can customize Xcode almost indefinitely, creating custom build phases, using both online and offline version control, and building projects of various kinds.

Understanding Groups & Files

It's important to understand how projects and files are organized, because copying or moving an Xcode project folder in Finder without setting it up correctly can destroy it. The Groups & Files pane in Xcode appears on the left side of the screen. It includes features that look like folders and other items. The folders are called *groups*. They appear to work like Finder folders. You can use reveal triangles to open and close them, drag files into and out of them, and create new folders. But the group structure is superficial and is for your convenience only — there are no equivalent folders in the project directory in Finder.

When you create a new project, a named project icon appears at the top left, and code and resource groups appear "inside" it. Figure 4.13 shows the folders in a new empty project, and Table 4.1 summarizes how they're used.

NOTE

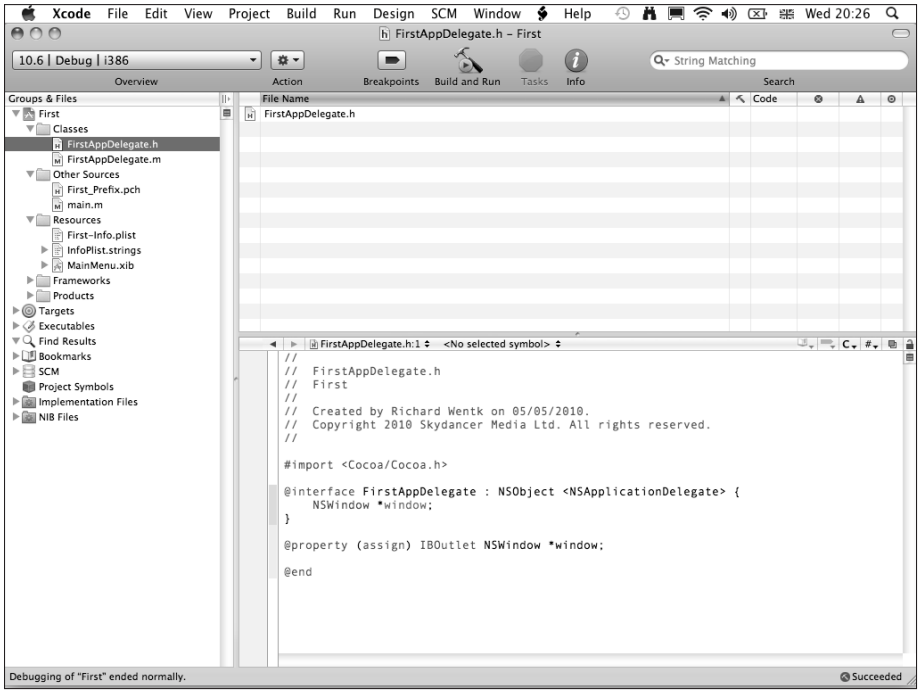
Under the folders is a list of additional features, including Targets, Build Products, Find Results, and others. You don't need to use these features to build and run a project.

Table 4.1 Groups in Xcode's Groups & Files Pane

Application	*Notes*
Classes	Objective-C class files
Other sources	Miscellaneous C files, including `main.c` A `.pch` prefix header file for the project
Resources	Nib files used by Interface Builder Optional graphics and other media files
Frameworks	Frameworks used in the project
Products	A list of build products (Usually there's just one — the finished app)

Figure 4.13

The Groups & Files area at the top left is a key feature in Xcode. You'll spend a lot of time here selecting files for editing and then working on them in the editor window in the lower-right area of the Xcode window.



It's critically important to understand that even though the Groups & Files pane looks like a Finder window, the items in this window are symbolic links to files on disk. Think of them as named placeholder links that point to a file path.

By default, the link name matches the filename — so `file.h` points to a real `file.h` on disk. But in Xcode this isn't always true, and you can rename, delete, and rearrange the links in the Groups & Files pane without affecting the files they point to. A link named `file.h` can actually point to `somethingcompletelydifferent.h` on disk.

Initially this indirect system appears almost completely counterintuitive, but it offers significant benefits. For example, you can include files from any disk location in an Xcode project without having to copy them to the project folder on disk.

CAUTION

Renaming or moving a project folder with Finder can break a project. If you move or rename files in Finder, the links in Groups & Files may point to file paths that are no longer valid. Xcode's default link settings don't allow you to copy a project folder without damage. This is easy to fix — for details see Chapter 18— but for now, don't attempt to move or duplicate project folders using Finder.

Selecting items for editing

To the right of the Groups & Files pane are two more windows. The File List in the top pane displays a list of files in the project. This is a flat version of the view in the Groups & Files pane, without groups and without the extra items that appear at the bottom. However, it adds some extra fields at the right that display information about file sizes and project groupings. You can use these fields to enable or disable compilation of selected files. This is an advanced feature and isn't relevant to entry-level projects.

Clicking any editable file in this window loads it into the text editor window at the bottom of the screen. You can also load items by clicking their names in Groups & Fields. This is one of Xcode's most productive features — all the files in a project can be accessed instantly without saving and loading.

TIP

The text editor window can display graphic files. Clicking any supported image file loads a preview into the window. There are no editing features, but you nominate an external editor in Xcode's Preferences. For details, see Chapter 18.

The text editor window also includes understated but powerful navigation options. Immediately above the text area are two drop-down lists. On the far left is a History list, which displays a list of recently accessed files. You can use this as yet another way to load files into the Edit Window. To right of the History list is a Symbol list that displays method titles and other headings, shown in Figure 4.14. You can use this feature to scroll down to a selected method with a single click.

At the top right of the text editor window are two other essential navigation features. The Split icon above the scroll bar splits the window into two or more sections that can display two different files. Often it's useful to view a header file in one window and implementation code in another. To the left of the padlock icon — which locks a file, preventing editing — is the Counterpart icon. This toggles the text window between a header file and its corresponding implementation file. Both icons are shown in Figure 4.15.

These features are simple, but they make Xcode a very productive environment. Before you start work on larger projects, it's useful to get into the habit of using them. They can literally save you hours of development time.

Figure 4.14

The History list and Symbols list are typical of Xcode's more advanced features. They're not prominently emphasized and it's easy to miss them, but they can be very effective timesavers.

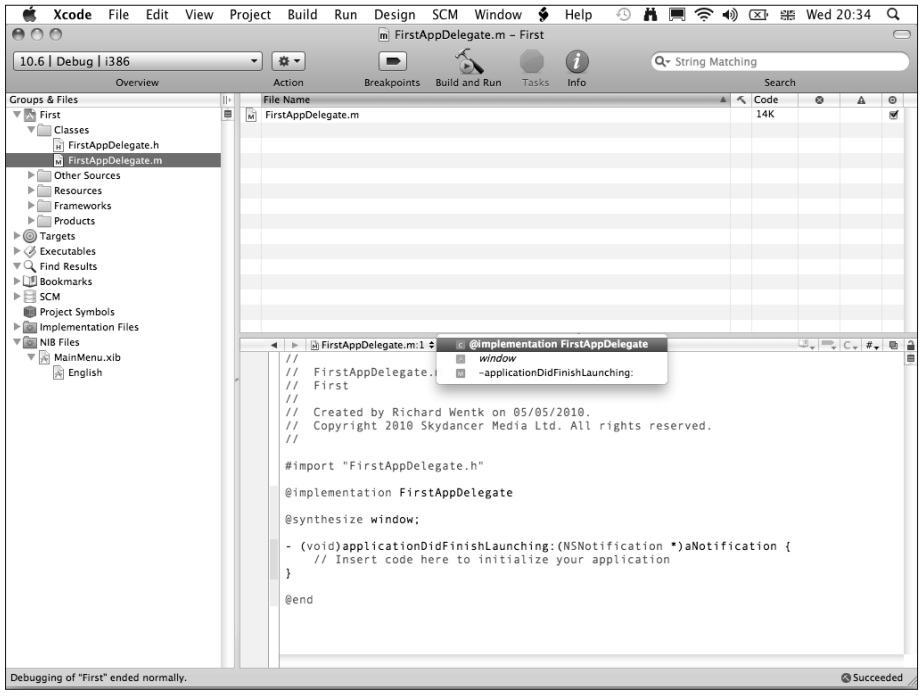
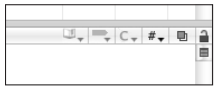


Figure 4.15

Xcode's icons include similarly powerful features you can use to speed up your work.



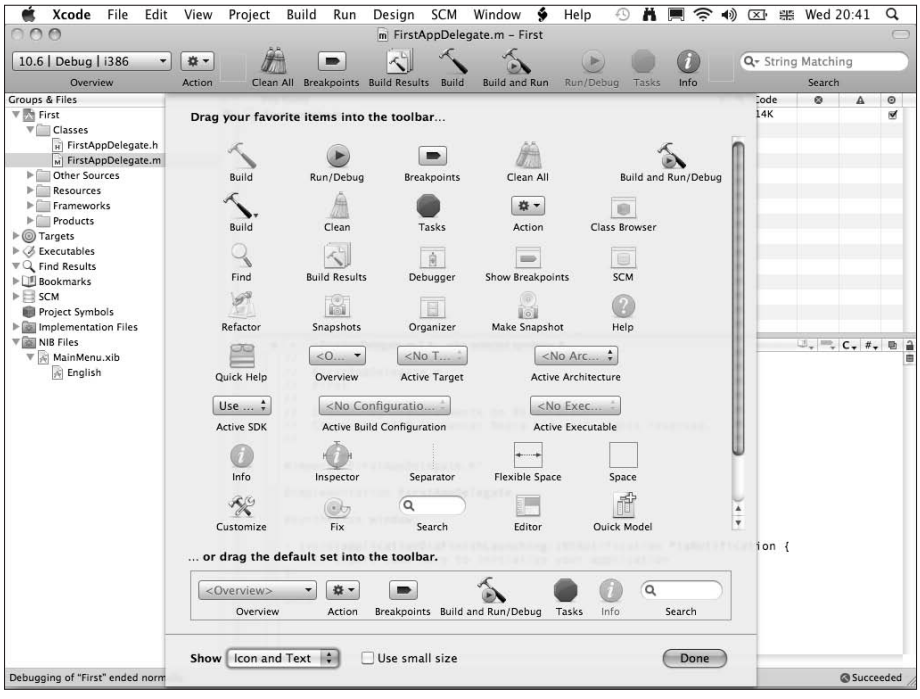
Customizing the toolbar

You can customize the toolbar by right-clicking it anywhere and selecting Customize Toolbar. To remove an item, drag it off the toolbar. To add an item, drag it from the drop-down window onto the central toolbar area. Figure 4.16 shows a suggested customized configuration. You'll find it useful to include the Build option, to check a build without running it, and the Clean All option, to remove existing build products and rebuild a project from scratch. As you gain experience with Xcode, it's likely that you'll want to add other more advanced features here.

Now that you have been introduced to Xcode, you can begin to explore some of the features of Objective-C and look at how they're used to create a working application.

Figure 4.16

Customizing the toolbar allows fast single-click access to some of Xcode's essential features. Unfortunately the options are fixed — you can't create and add your own icons or implement your own features.



Summary

In this chapter you learned about Apple's developer programs and were introduced to the features of the different developer programs and levels. You discovered how to register as a developer, how to download and install the Xcode SDK, and how easy it is to create, build, and run an empty sample project. You were also introduced to some of Xcode's more advanced features, including the editing icons and the toolbar customization feature.

5 Introducing Classes and Objects in Objective-C

Objective-C is object-oriented. Code is assembled from objects defined by class templates; instances are created and released dynamically and controlled by messages. Cocoa is a mix of object-oriented features and conventional C, blended with practical tools and techniques for creating and managing objects.

This chapter is a first look at the theory of object-oriented development in Objective-C and introduces class definitions, messaging, and constructors. The chapters that follow illustrate the practice and explain how to create links between Objective-C code and the contents of a nib file.

Understanding Objects

Objects can represent concrete data or abstract processes and relationships. Some of the advantages of an object-oriented approach include:

- **Abstraction.** Objects make application design simpler by hiding low-level complexity. You can concentrate on creating clean relationships between the elements in your application.
- **Encapsulation.** Objects can contain other objects, linked and grouped in various ways. Grouped objects can be treated as a single object with a simplified interface, or their elements can be accessed individually.
- **Simplified flow control and looping.** Objects can be counted, enumerated, looped, collected, and processed as if they were simple data types. Some of Cocoa's data types include looping and enumeration features. You can use these features to process a collection of objects with a single line of code.
- **Intuitive event management.** Objective-C is explicitly event driven, and the event management features are flexible and powerful.
- **Intuitive but formal data dependency management.** An object can respond to a message by accessing its own data or by interrogating other objects. Data dependency can be managed automatically.

5 In This Chapter

Understanding objects
Designing objects
Creating classes
Using objects

Table 5.1 lists some possible practical applications. *Properties* are data fields, and *methods* are code blocks that can be triggered inside an object to elicit a response.

Table 5.1 Examples of Possible Objects and Applications

Object	*Possible Properties*	*Possible Methods*
GUI window object	Position Size Opacity Front/key window status	Respond to a mouse click Maximize/Minimize/Close Move to position Report position Bring to front
Game object or token	State Position Velocity Texture	Move to position Change state Change texture Redraw
Game playing field	Game state Game score Player number Array of tokens	Update score Start with new player Report high score
Address book	Individual entry (can be a subobject) Entry subproperties	Add entry Delete entry Update subproperty Count entries Save/restore entries Search entries
Music synthesizer or sample event	Synthesizer/sampler settings Note start/end times	Play note Stop note Create note with settings
Generic object	Pointer reference Memory state	Create Destroy and release memory Copy Save/restore from disk

Part of the challenge of object-oriented programming is object design. The ideal design encapsulates powerful features within a clean and intuitive interface that is responsive and easy to work with.

NOTE

Cocoa also includes hybrid data structures that aren't true objects but are partially interchangeable with them. Formally these are plain C-language `structs`, but they can be cast or copied to a related Cocoa object class. In a typical application you use a combination of Cocoa objects, hybrid structs, and standard C data types.

Some Cocoa objects are used as drop-in solutions that implement specific features; for example, the `NSDate` object holds information about the system date and time. Others are designed to be modified and extended. All Cocoa objects are extensions of an object called `NSObject`, which implements essential object creation, memory management, and copying methods. You can extend `NSObject` to create a minimal blank object. You can also modify most of Cocoa's other objects to create sophisticated customized features.

NOTE

It's possible to create customized objects in Objective-C that aren't related to `NSObject`, but this is usually a wasted effort. `NSObject` implements core object management features "for free." A customized alternative has to implement similar features from scratch — which isn't a small project.

Understanding classes

A *class* is a blank template that defines an object's features. For example, Cocoa's NSWindow class defines the features of an OS X window. Class definitions list an object's property fields and the methods it supports, including the code blocks that implement them. In theory, class definitions are blank and static: they hold no data and do nothing. To work with data, the templates are used to create *instances* — working copies of an object. In practice, Objective-C treats the class itself as a single separate instance with unique methods and properties. *Class methods* are often used to create and return information about class instances.

Table 5.2 summarizes the different elements that define objects and make them usable.

Table 5.2 Object Anatomy: Classes, Instances, and Messages

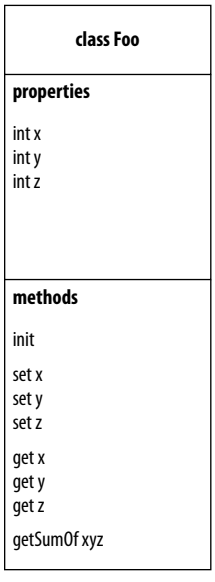
Feature	*Summary*
Class definition	A template that defines the methods and properties in each object. Every object has a single unique class template.
Instance	A working version of an object that can store data. Copies are created and released from memory as needed. Each copy organizes its data using the same template.
Pointer	A named variable that stores the base memory address of an instance. The string is used as a handle for the address. Pointers must be unique within a given scope.
Property	A data field. Class templates define the data type of each field and specify a useful name. Instances can store data in each field. *Properties cannot be accessed directly*. Accessor methods must be included to make them accessible from other objects.
Methods	Code blocks that define how an object responds to messages. Optionally, methods can take parameters and generate return values.
Messages	Events that are sent to an instance to run a named code block. In Objective-C, messages can also be sent to the class template.
Accessors	Methods that read or write properties — explicit setter (write) and getter (read) code. Accessors can be generated automatically with Objective-C's `@synthesize` directive. Optionally, accessors can force an update of an object's internal data or state while implementing a read/write.
C features	C isn't used in object-to-object messaging, but can appear in object method code, and C data types can be used as properties or wrapped into objects. C function calls are often included within objects as helper code that implements low-level features. Parts of Cocoa use C structs as proto-objects that are partly interchangeable with full objects.

Introducing classes and instances

Figure 5.1 shows a stylized sketch of a simple class with three properties and eight methods — the `set` and `get` methods are the accessors. In a Cocoa application, all objects must have an `init` method that initializes properties and private internal values to defaults. The list of properties and methods that can be accessed by other objects is the *interface*.

Figure 5.1

Anatomy of a simple class. In this example the properties are `ints`. In a more complex class, the properties might be pointers to other objects, and the methods might access data in other objects.



NOTE

Objects may include private variables, that are hidden from other objects, and private methods, that the object can run on itself. Private features don't appear in the object's interface. By definition, features in the interface should be public.

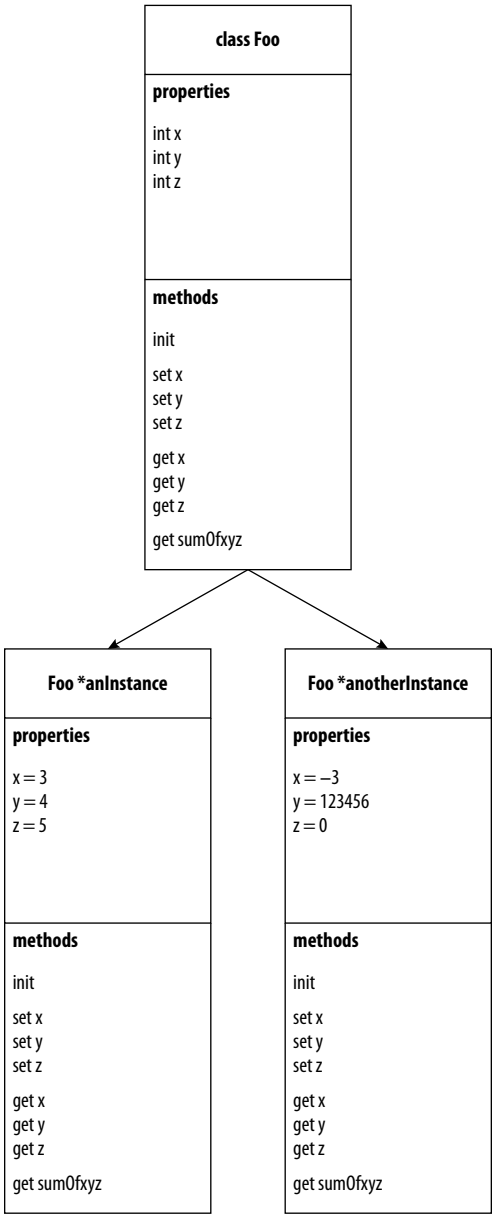
Figure 5.2 illustrates how a class can have two instances. Unlike the template, which is blank, the instances contain working data. Because the internal values of x, y, and z are different in each instance, the `sumOfxyz` method returns a different total. When the objects have their properties set as shown, you can trigger the `sumOfxyz` method to return those values; for example:

```
int aNumber = [anInstance sumOfxyz]; //Sets aNumber to 12
int aNumber = [anotherInstance sumOfxyz]; //Sets aNumber to
   123453
```

An object that receives a message is called the *receiver*. In this code `anInstance` and `anotherInstance` are both receivers.

Figure 5.2

Creating instances makes it possible to *use* a class, filling its fields with data and calling its methods to access and process that data.



Creating implicit and explicit instances

It's critically important to understand that an application can create object instances *explicitly* with Objective-C code or *implicitly* by loading them from a nib file at run time.

This can seem mysterious, because not all objects have supporting code. For example, the blank template application introduced in the previous chapter includes a working menu. The code in the template doesn't create or display the menu, or handle standard events such as Quit. Instead, the menu is defined in the application nib file and is loaded when the application launches. The nib also includes initialization settings for the menu that bind items to standard events.

When you run the application, the menu objects are created automatically and some of the menu options are set up to respond when you select them. In this example, this happens without supporting code. You can choose to add code to access objects loaded from a nib. If you don't, the objects are still loaded and created in memory, but you can't directly access their properties or methods.

Loading objects from a nib is an example of *dynamic loading* in Objective-C. Objects can be loaded and released at run time, and it's possible to reconfigure the architecture of an application as it runs. Typically, Interface Builder manages part of this process, and the rest is implemented with code.

Designing objects

Properties can be traditional C data types, custom or predefined structs, or objects. There's no limit on the number of properties in a class, but objects with hundreds of properties are unwieldy and may need to be redesigned.

There is also no restriction on the complexity of a method. A method interface can hide deep complexity, and method code can access the properties and methods of other objects. Good method design hides complexity by creating interfaces and results that are easy to understand, and which encapsulate a useful return in a simple way.

It's good practice to define class, property, and method names that are explicitly descriptive and easy to understand, even when this means extra typing. This improves code clarity and can help simplify object design. For example, extended variations of `sumOfxyz` might return the result as a string:

```
sumOfxyzAsBinaryString
sumOfxyzAsHexString
sumOfxyzAsDecimalString
```

Cocoa often uses descriptive names for its objects and methods. It's bad practice to create ambiguous or imprecise names:

```
sum //Sum of what?
Total
sumAsString //What kind of string?
returnValue
```

Naming conventions

By convention, class and function names use `UpperCamelCase` while instance, method, property, and general data-type names use `lowerCamelCase`. Spaces are not allowed, but underscores are. It's useful to explicitly include the type of class of an object or data structure in its name.

NOTE

Supposedly CamelCase gets its name from the double hump of the two capitals. It has a more formal name — *medial capitals* — but this lacks alliteration, has twice as many syllables, and doesn't mention an animal, making it harder to remember. CamelCase isn't enforced by the compiler, but it is a standard naming convention.

Cocoa classes and features are named using `UpperCamelCase` with an optional additional framework prefix. For example, the Cocoa/NeXTStep classes are prefixed with `NS`, an abbreviation for NeXTStep:

```
NSWindow
NSViewController
NSBrowserDelegate
```

Other OS X libraries have their own prefixes. Because of overlaps, the naming scheme is slightly random. For example, CoreAudio items begin with `Audio` and not `CA`, which is used by the Core Animation library.

CAUTION

Not every Cocoa feature prefixed with `NS` is an object. Most are, but there are common exceptions. For example, `NSInteger` is a wrapper for a standard `int`. For details of the exceptions, see the Foundation Framework Reference in the developer documentation. The Other References group lists the C functions, C data types, and constants that are used in Cocoa but aren't defined as objects.

Because names are descriptive, they can be long. For example:

```
CFAbsoluteTimeGetDifferenceAsGregorianUnits
```

is a function in the Core Foundation Time Utilities library. Constants in Cocoa are prefixed with a `k` followed by a library prefix, followed by a long name, for example:

```
kCFAbsoluteTimeIntervalSince1970
```

It's helpful to follow the conventions in your own class definitions. If you create your own custom frameworks, it's essential to add a unique abbreviation prefix.

There's no formal designation for developer-created objects. Many developers use `a`, `my`, and `this` as prefixes to indicate instances created in their code.

```
aWindow
myView
thisDictionary
```

This is informal, but works well for simple projects.

Alternatively, developers can use their initials or a unique two-character or three-character acronym to identify their classes. This is helpful when working on collaborative projects because names such as `aWindow` may already be in use. For example:

```
RWWindow
CDRObject
CDRView
```

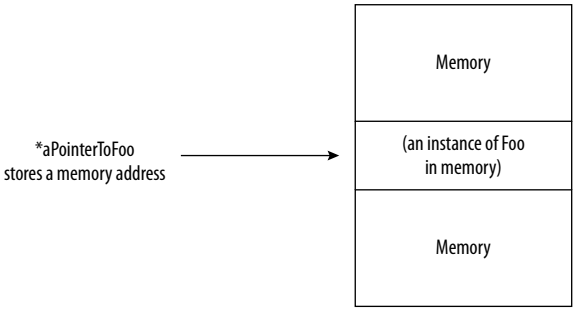
Object constructors and pointers

Objective-C uses C's asterisk pointer convention, as illustrated in Figure 5.3. A pointer name can be any string that is unique within its scope, but it's useful to include the class name as a suffix. For example, a pointer to an instance of class `Foo` is declared like this:

```
Foo *aPointerToFoo = …//This is traditional
```

Figure 5.3

`aPointerToFoo` is a variable that holds a memory address. The exact data type depends on whether the application uses a 32-bit or 64-bit memory model. It's legal, and occasionally useful, to compare or manipulate pointer addresses directly.



A unique pointer is generated whenever a new instance of a class is created. Objective-C doesn't support formal constructors. Instead objects are created using a standard code idiom that runs two methods called `alloc` and `init` to return a pointer to an instance of a class. `alloc` creates an instance of an object, and `init` initializes it. The methods are always nested, as shown here:

```
Foo *aPointerToFoo = [[Foo alloc] init]; //Do this

Foo *aPointerToFoo = [Foo alloc];
aPointerToFoo = [aPointerToFoo init]; //Don't do this
```

The first version appears throughout Cocoa code. With occasional exceptions, described next, you can use this code as-is to create Cocoa objects.

```
//Create a new instance of NSView
NSView *aView = [[NSView alloc] init];
```

Understanding alloc, init, and nesting

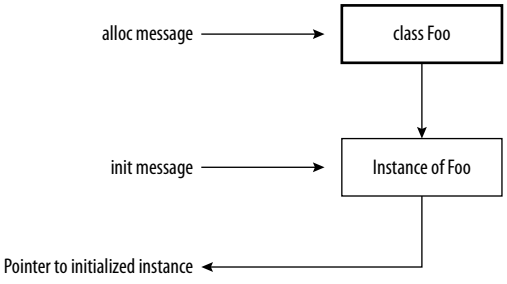
Figure 5.4 illustrates how `alloc` and `init` work together. `[Class alloc]` returns a pointer to an instance of `Class`. But because the code is *nested* — the return from `[aClass alloc]` is passed directly to `init` — this pointer doesn't need an explicit name.

CAUTION

`init` here means any initialization method. Some `init` methods take parameters. This doesn't alter the idiom; as long as the `alloc` and `init` methods are nested, the code is correct.

Figure 5.4

The `alloc` message runs on the `Foo` class and returns an instance. The `init` method runs on the instance and returns a pointer to an initialized instance, which may — sometimes — be different than the pointer returned by `alloc`. Nesting `alloc` and `init` guarantees that the final pointer is usable.



Generally, you can use nesting to avoid creating temporary pointers. It's good practice to include one or two levels of nesting. It's bad practice to nest more than three levels because the code becomes difficult to follow. Unfortunately, because of Cocoa's syntax, deeper nesting is sometimes unavoidable.

Sample code that uses the new instance might look like this:

```
[aPointerToFoo setx: 1];
[aPointerToFoo sety: 2];
[aPointerToFoo setz: 3];
int aTotal = [aPointerToFoo sumOfxyz];
```

You might create another instance with

```
Foo *aDifferentPointerToFoo = [[Foo alloc] init];
```

followed by similar code using the `aDifferentPointer` instance:

```
[aDifferentPointerToFoo setx: 15];
//Etc…
```

It's critically important to understand that pointer variables aren't identical to instances. A pointer is a `size_t` integer that points to the memory used by an object instance and is used as a reference to it. Misunderstanding this can result in code that leaks memory or crashes. If a pointer is overwritten before an object is released from memory, the object's memory is lost. This becomes very important on the iPhone where memory is managed manually. But overwriting a pointer can create a memory leak in a Mac application, even with garbage collection.

Class and instance methods

A class template is equivalent to a type definition. In the same way that two named `int`s are different entities, the two instances of `Foo` are completely distinct and independent. They use the same template, but there is no other connection between them.

Most objects are designed to be used as instances, and the class itself is a template with no active features. But some Cocoa objects include *class methods* that are run on the class and not on an instance. Certain class methods return an object without using `alloc`, for example:

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

This reads the date and time from the system date object, writes it to an instance of `NSDate`, and returns a pointer to it. It would be more consistent if the code looked like this:

```
NSDate *now = [[NSDate alloc] init];
```

In fact, this is also valid. `NSDate`'s init method automatically returns the current date and time. But the first version is used more widely, apparently for historical reasons.

NOTE

Class methods are often thread safe. Instance methods may not be thread safe. The documentation doesn't always detail the differences, so if your code supports multi-threading, you may need to experiment to discover which option runs reliably. For information about creating and managing threads, see Chapter 11.

Class methods are typically used to create objects — `alloc` is a class method — or to return a pointer to a system object, such as the shared application manager `NSApplication`. For example, running the `sharedApplication` class method on `NSApplication` returns a pointer to the current application:

```
//Get a pointer to the current application
//and send it a terminate message to quit
[[NSApplication sharedApplication] terminate: nil];
```

Subclassing and inheritance

Cocoa is a library of prewritten classes that you can drop into your code. When you *subclass* an existing class, you create a new version of the class and use it as a starting point for your own additions. You can then add new features of your own, or *override* — write code that re-implements — some or all of its existing features.

When you create a subclass, it *inherits* all the features of its parent. Before you begin to modify it, the subclass is a direct copy of the existing parent template, with all its properties and methods. It has a new name, but is functionally identical to its parent class.

Inheritance and subclassing save you time. You don't need to reinvent the wheel, or the window — you can reuse an existing class, extending it or modifying it as needed. Cocoa is designed to encourage this kind of customization. You can subclass almost any class, and some Cocoa classes are deliberately minimal with bare outlines of features, rather like a car chassis without a body, wheels, or seating.

For example, the `NSView` class — `UIView` on the iPhone — includes a `drawRect:` method that can fill the view with graphics. By default, `drawRect:` is empty and does nothing. If you don't add code to it, it continues to do nothing when the application runs. When you want to draw custom graphics in a window, you subclass `NSView` and fill out `drawRect:` with custom code. The new `drawRect:` overrides the old empty version.

TIP

Sometimes it's easier to modify an existing class without creating a subclass. Objective-C includes a *category* feature that makes this possible. The name is misleading — no categories are categorized. Instead, an existing class is extended with new properties and methods. Unlike a subclass, the name doesn't change, and the new features are added "in-place." For a practical example, see Chapter 8.

When you create a subclass, it becomes a new class in its own right. You can subclass it again, and then subclass the subclass indefinitely, without restrictions. You can also create multiple subclasses with different features from the same source class.

Cocoa uses this feature to organize classes into a hierarchy or tree. At the root is the minimal object template class called `NSObject`. As illustrated in Figure 5.5, classes spread out from `NSObject` in various branches. Each branch implements different but related features such as window and UI management, networking, data management, and so on. The further away from `NSObject` a class gets, the more specialized it is.

The complete `NSObject` hierarchy includes more than 100 objects. The developer documentation includes information about each class's place in the hierarchy and the classes it inherits its features from. You don't need to memorize the class relationships, but you do need to be aware of them, and an outline understanding of the class hierarchy is useful.

Figure 5.5

A very small part of the `NSObject` subclass hierarchy. For a complete view, see `http://developer.apple.com/mac/library/documentation/Cocoa/Reference/Foundation/ObjC_classic/Intro/IntroFoundation.html`.

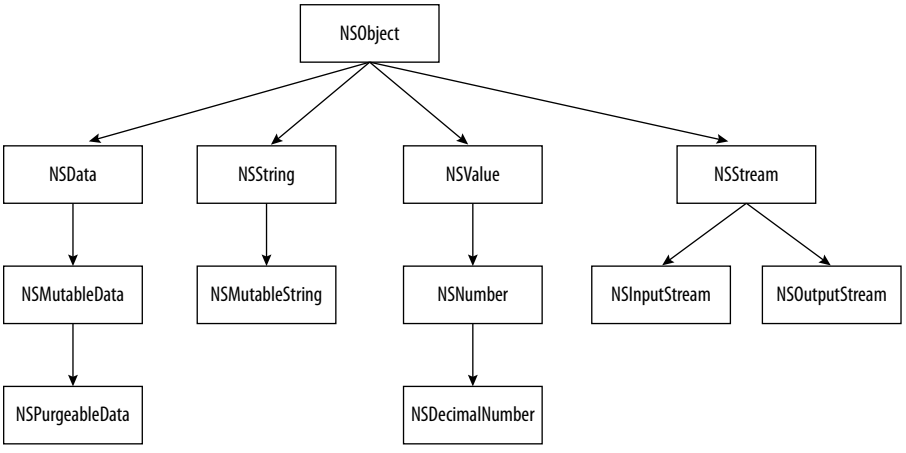


Figure 5.6 uses `NSButton` as an example. Under the reference title you can see an `Inherits from` field with a list of classes. This list tells you that `NSButton` includes all the properties and methods listed in its own class reference, and that it *also* implements all the properties and methods built into `NSControl`, `NSView`, `NSResponder`, and `NSObject`.

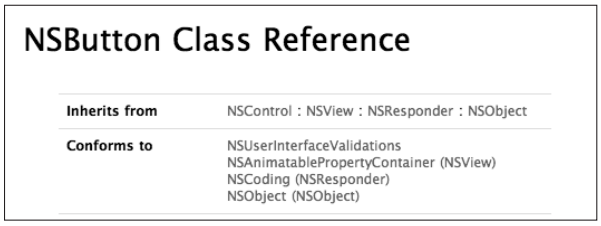
These extra methods and properties aren't listed on the `NSButton` page — to find them, you have to read the reference pages for the other classes. It's good practice to do this, especially when you're new to Cocoa programming, because it helps to fix an outline of the hierarchy in your memory. It also expands your ideas about what's possible. For example, you can use `NSView` methods to animate instances of `NSButton`, changing their size to make them pulse, or changing their position to make them jiggle in an iPhone-like way. Working through the subclassing hierarchy isn't only a technical process, it also can be a creative way to discover new possibilities for your application.

NOTE

The `Conforms to` field lists a separate set of optional methods that can be used in subclass code.

Figure 5.6

A close-up of the `NSButton` class reference showing subclassing and inheritance information



Subclasses, superclasses, and the root class

`NSObject` is Cocoa's *root class*. When you create a subclass, the original class is known as the *superclass*. You can send messages from any object to its superclass using the `super` variable. `super` isn't a separate object. It's used to "unoverride" method names, accessing the unmodified and unextended originals. For example, if you subclass `NSView` and call `[super drawRect:...]` your code runs the original unmodified `drawRect:` method. This is more of an illustrative example than a useful one, but the same technique can be used to run unmodified methods in objects that have useful features. `super` is often used in `init` methods, described in Chapter 7.

Creating classes

Class definitions are split across two files — a header file with a `.h` extension and a code file with a `.m` extension. The header is called the *interface*. It declares the methods and properties that are visible to other objects. The code file holds the *implementation* — the code that runs when each method is triggered. The implementation can also contain private variables and methods that are invisible to other objects. A feature is only public if it appears in both files.

TIP

As you'll see in Chapter 6, Xcode makes it easy to create new classes with a selection of default templates. You don't have to type in the contents of the interface and implementation files from scratch — you can start with a template file and modify it.

Defining a class interface

The contents of an interface file are organized like this:

```
#import <any required headers>
@interface MyNewClass : NSObject <protocols>{
(a list of instance variables)
```

```
}
(a list of public properties prefixed with @property
   declarations)
(a list of public methods that can be accessed by other objects…)
(a list of public setter and getter methods, if used)
@end
```

The `@interface` directive tells Objective-C that the following lines define the class interface. The rest of the line includes the name for the class, which can be any unique string followed by the name of the superclass from which it inherits its features.

In words, this line tells Objective-C to create a new class called `MyNewClass`, copying methods and properties from `NSObject`. To subclass a different class, replace `NSObject`. For example, the following subclasses `NSView`:

```
@interface MyNewViewClass : NSView <protocols>{…
```

NOTE

The `protocols` field includes one or more named bundles of optional methods that can be used in the class. A protocol is a slightly more complex form of `import/include`. Protocols are used as a quick way to import groups of related methods to a class. They're optional, but many Cocoa objects use them. For some examples, see Chapter 6.

Instance variables, also known as *ivars*, can be objects with pointers or standard C data types. For example:

```
{
  NSView *aView;         //This is a pointer to a view
  int numberOfGiraffes; //This is a standard C int
}
```

The list of properties between the curly brackets defines private internal variables. To expose them to other objects, the property names must be prefixed with a `@property` directive.

```
@property (nonatomic, retain) NSView *aView; //aView can be
   public
```

NOTE

`(nonatomic, retain)` defines the memory management options for properties that are objects. C data types don't require this extra information. For more information, see Chapter 12.

The methods list defines every public method that can be triggered by other objects. For example:

```
- (void) thisIsAMethod;
//This is a method with no return value and no parameters
- (NSValue *) thisIsAnotherMethod: (int) aNumber;
//This is a method that returns an instance of NSValue and takes
   an int as a parameter
```

Methods are listed as signatures — one-line summaries that list the type, output type, name, and parameters used in the method. Signatures end with a semicolon. They do not include implementation code.

In both the interface and implementation files, Objective-C uses the – character as a prefix for instance method definitions and + as a prefix for class method definitions.

```
- (void) thisIsAnInstanceMethod;
+ (void) thisIsAClassMethod;
```

Defining accessors: setters and getters

In Objective-C, properties are private and invisible unless *setter* and *getter* methods are defined for them. Setters write values; getters read them. Properties can allow either or both modes of access. You can define setters and getters explicitly with code, or implicitly using the `@synthesize` directive.

Use `@synthesize` to generate the setters and getters required for simple read and write operations. The directive works rather like a macro, and it creates setter and getter code automatically. It also implements dot-format property access.

```
anInt = anObject.numberOfElephants;
anObject.numberOfWheels = 5;
```

This syntax is common in other languages. In Objective-C, it is only valid if the properties are synthesized.

Use explicit setter and getter code when reading or writing a property triggers a more complex outcome, such as a change in the state of an object or an operation that accesses other properties and features. For example, a setter can automatically redraw visible graphics when a window's size is changed, or a getter can count the number of times a property has been accessed and write the count to another property.

By convention, getter method names are identical to the name of the property they access, while setters prefix the name with `set`. Define setters and getters in the interface as follows:

```
- (int) numberOfElephants; //A getter for the numberOfElephants
   property
-(void) setNumberOfWheels: (int) aNumber; //A setter for the
   numberOfWheels property
```

To use them, the syntax is

```
anInt = [anObject numberOfElephants];
[anObject setNumberOfWheels: 5];
```

The compiler doesn't enforce this naming convention, but it's used in Cocoa and it helps make the code easy to read. There are no restrictions on the complexity of setter and getter code. If necessary, a single setter method can dramatically change the state of the entire application.

Using self

In a class, `self` is used to access internal methods. For example:

```
[self doSomethingInternal];
```

triggers the `doSomethingInternal` method, which is assumed to be present in the class. Internal methods can be private or public. You can also use `self` to make property access operations explicit.

```
//If aProperty uses a custom getter, this is necessary
aValue = [self aProperty];

//…because this is interpreted as a pointer copy
aValue = aProperty;
```

Defining a class implementation

The contents of an implementation file are organized like this:

```
#import MyNewClass.h
#import (any other classes that are referenced)
@implementation MyNewClass
(an optional list of @synthesize directives for properties)
(an optional list of private variables)
(a list of methods with full implementation code)
@end
```

The `#import` directive loads the interface header file. For historical reasons, Objective-C uses `#import` for Objective-C headers and `#include` for C headers. The rest of the file is a list of implemented methods. Any method listed in the interface *must* be implemented in the implementation. The implementation doesn't have to be complete and finished — a stub will do — but the compiler will throw an error if it's absent.

NOTE

It's possible to combine the header and implementation within a single file. This is bad practice because classes often need to import the interface declarations of other classes. This becomes very inefficient when headers aren't separated from code.

If a class references a class or a framework, it must include a `#import` directive for its headers. For clarity and memorability, it's good practice to include the `#import` directives in the implementation.

To use `@synthesize`, follow it with a list of property names and end the list with a semicolon:

```
@synthesize aProperty, anotherProperty, somethingElse, aView;
```

This runs the `@synthesize` feature on the four listed properties and generates setter and getter code for them. You can then use dot syntax or conventional method code to access these properties from other objects.

```
aValue = [thisObject aProperty]; //Using a synthesized getter
aValue = thisObject.aProperty; //Identical code using dot syntax

[thisObject aProperty: 5]; //Using a synthesized setter
thisObject.aProperty = 5; //Identical code using dot syntax
```

Defining public properties

To summarize, *you must do three things* to create a property that can be accessed by other objects:

1. **Include it in the list of ivars in the interface.**
2. **Declare it again as a** @property **in the interface.**
3. **Add an explicit setter/getter method or** @synthesize **the property in the implementation.**

All three steps are obligatory. If you skip one, the build will fail or the application may crash. If you change a property name, you must change it in all three places.

TIP

It can be difficult to keep these three steps in mind when concentrating on the rest of the code, so it's helpful to use this summary as a check list. The three steps eventually begin to become automatic, but forgetting a step remains a popular and common error.

Defining public methods

To define a public method, follow these four steps:

1. **Declare the signature in the interface.** The signature includes the class/instance type, return type, name, and parameter list. End with a semicolon.
2. **Copy the signature to the implementation file.**
3. **Replace the semicolon with an opening curly bracket.**
4. **Add implementation code and close the curly bracket.**

For example, the following code completely defines a method called `thisMethodDoesNothing`:

```
- (void) thisMethodDoesNothing; //Add this line to the method
   list in the interface
```

```
- (void) thisMethodDoesNothing {
} //Add these two lines to the method body list in the
   implementation
```

The rules for defining parameter lists and return values are similar to those in C. Methods can take any number of parameters. Parameter types must be explicit. For example:

```
//This line appears in the interface
- (AClass *) thisMethodDoesSomething: (AnotherClass *)
   aParameter;
//This block of code appears in the implementation
- (AClass *) thisMethodDoesSomething: (AnotherClass *) aParameter
   {
 AClass *aResult; //This is a private variable for this method
 aResult = [aParameter doSomethingToIt];
 return aResult;
}
```

TIP

`aParameter` is used as a private variable within the method. Name conflicts are common, so it can be useful to prefix parameters with an underscore to ensure that local parameter names don't conflict with global properties.

```
- (AClass *) thisMethodDoesSomething:
  (UsingAnotherClass *) _aParameter;
```

Xcode assumes that if your code references a method, it exists. If the compiler can't find the method at compile time, it logs a warning but allows the build to complete. The compiler generates the same warning if you use a method before you define its implementation.

Method names are only loosely linked to objects, and the Objective-C run time looks up method names dynamically. For example, as long as `aMethod` is available in both classes, this is valid code even if both implementations of `aMethod` are different:

```
aPointer = anInstanceOfAClass;
[aPointer aMethod];
aPointer = aInstanceOfASubclassOfAClass;
[aPointer aMethod];
```

CAUTION

By default, when you create a new project, Xcode assumes you'll be managing memory manually. Usually you'll want to enable garbage collection instead. For instructions on how to do this, see the last section in Chapter 12. Projects in the rest of this book assume that garbage collection is enabled.

Using Objects in Objective-C

This overview outlines a first look at objects in Objective-C, but Objective-C uses objects in unusual ways. A key point is that the relationship between objects, messages, and data is kept as loose as possible. Connections that are defined at compile-time in most languages can be changed at run time in Objective-C. For example, Objective-C allows loose typing, and includes class management features that can read an object's class at run time, or test if it's a member of a class hierarchy.

Table 5.3 introduces Objective-C's more creative elements. This list isn't a complete summary of Objective-C's more creative options, but it lists the features that are widely used in Cocoa applications.

Table 5.3 Key Objective-C Features

Feature	*Summary*
Object-oriented architecture	Objects can be defined, instantiated, and destroyed. Subclassing and method overrides are widely used.
Explicit and implicit object creation	Object instances can be created in code, or they can be defined as resources in an Interface Builder file and loaded automatically when the application runs. The loading process can be spread across multiple files.
Dynamic loading	Objects can be loaded when needed and released dynamically to minimize memory use.
Loose typing in message parameters using `id`	Type information in message parameters can be enforced, ignored, retrieved dynamically, or implied at run time using a placeholder `id` data type.
Loose links between objects and messages	Messages and objects are loosely bound - a named message can be sent to any target object that supports it. Target objects can be selected at run-time. Certain objects allow explicit nomination of a target object for certain messages.
Message selectors	A `selector` data type is available as a container/handle for each method. Code can use selectors to choose messages dynamically at run time.
Automatic message creation	Messages can be generated automatically without explicit code — for example, when an object property is updated.
Self-messaging	Objects can trigger internal methods by sending messages to `self`.
Abstract classes	Cocoa includes a small number of abstract classes, mostly used as bundles of methods that must be overridden manually.
Delegation	Related messages can be collected into *protocols* and adopted by any object, which acts as a *delegate* that implements handlers for the group. Delegate objects can be assigned and reassigned dynamically.

Cocoa relies on this looseness to implement some of its more flexible features. For example, a single object can process messages sent from many different objects with different parameter lists. This is often useful in UI design. An event handler can receive messages from various UI objects without making assumptions about the format or source of the messages. Similarly, data collection objects — arrays, sets, and dictionaries — can store various data types in a single collection. Chapters 8 and 9 illustrate these features in more detail.

Having looked at the theory, you're ready to move on to some practice in the next chapter, which demonstrates how to use Cocoa objects in a simple application.

Summary

In this chapter you were introduced to objects and looked at some of the benefits offered by object-oriented design when compared with more traditional programming models. You explored classes and instances and learned about Objective-C and Cocoa naming conventions.

Next, you looked at the benefits of subclassing and inheritance and discovered how they were used in Cocoa to simplify development. Finally, you saw sketches of the interface and implementation code used to create classes in Objective-C and were introduced to an overview of some of Objective-C's unique features.

6 Getting Started With Classes and Messages in Application Design

Although Cocoa and Xcode both use Objective-C, having a theoretical understanding of classes isn't enough to work with them effectively. Essential practical skills include:

- Understanding the structure of a Cocoa application.
- Finding the right Cocoa object to solve a problem or implement a feature.
- Translating the Cocoa documentation into working code.
- Using Xcode and Interface Builder to add, release, and manage application objects.

Understanding the Cocoa Development Process

Cocoa code often requires detective work. Cocoa objects are complex, with many features that cross-reference other objects and data types. The developer documentation is comprehensive, but the links between classes and supporting features are perhaps not as clear as they might be.

When you use a Cocoa class for the first time, it's likely you'll follow a process that I'll outline in this chapter, with false starts, incorrect guesses, and repeated searches through the documentation. When you discover a solution, it's likely to be elegant and powerful. You can often implement a sophisticated solution with a single line of code, but it may take you some time to find this ideal.

Timesaving help is always welcome, which is why it's so helpful to look for sample code and related online discussions. Occasionally you can find a worked example that solves your problem. More often you'll find references to unexplored or unnoticed features that can point you toward a solution. Generally, you'll save time by looking for worked examples outside the documentation; for example, in Apple's own sample code and solutions posted in online forums.

6 In This Chapter

Understanding the Cocoa development process

Understanding applications

Discovering object methods and properties

Creating subclasses

Introducing Code Sense

Receiving messages from OS X with a delegate

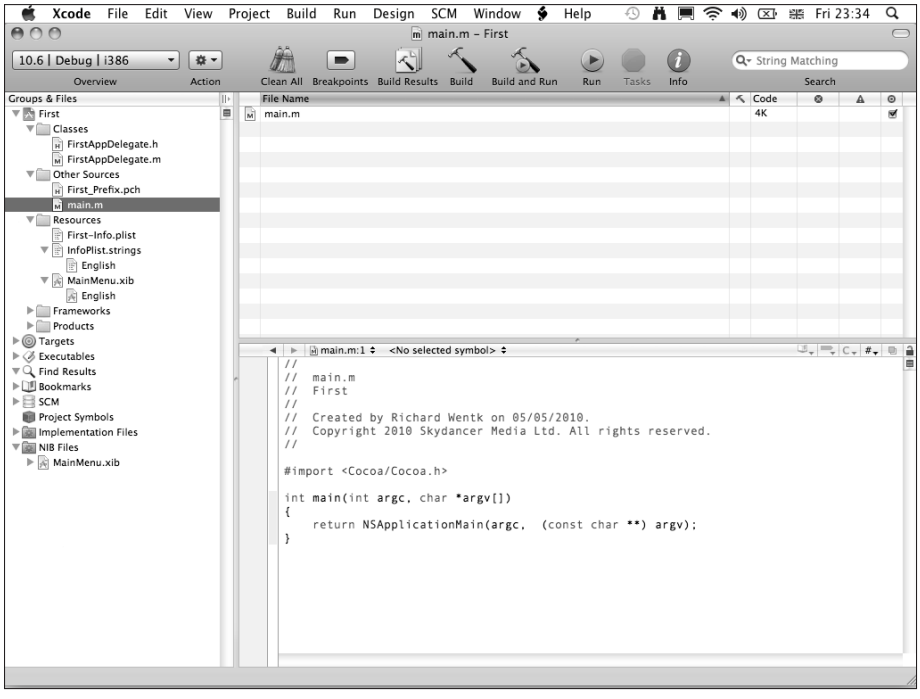
Receiving messages from OS X with NSResponder

Understanding Applications

I'll use the minimal sample project from Chapter 4 as a teaching lab for getting started with application design. Begin by launching Xcode and reloading the project from Chapter 4 called `First`. You'll see three groups — Classes, Other Sources, and Resources — at the top of the Groups & Files pane in Xcode. Click the reveal triangles beside them to explore their contents, and then click `main.m`, as shown in Figure 6.1.

Figure 6.1

`main.m` is included as the launch point for every application, but it is usually left unedited. The `argc` and `argv` parameters are ignored, and the function doesn't return a useful result code.



These three groups define the key elements in an application. Although only some of them are code files, you can think of them as the application's source code. The other items in the Groups & Files pane define how Xcode builds these elements into a finished application.

NOTE

The easy, but unexpected, way to set the developer name and copyright info in the comments at the start of each file is to create a card with personal details in Address Book and choose Make This My Card from the Card menu. Xcode reads the information from Address Book when it creates a new project. You can also enter the following in Terminal on a single line:

```
defaults write com.apple.Xcode
   PBXCustomTemplateMacroDefinitions
    '{"ORGANIZATIONNAME"="<OrgNameHere>";}'
```

Table 6.1 lists the key elements in each folder. In this chapter, you'll concentrate on the contents of the top three groups. All elements are essential — a project won't build without them — but some can be used as is, without changes.

NOTE

The files in a real project include the project name as a prefix. This prefix isn't included in the table.

Table 6.1 Elements of a Cocoa Application

Element	*Explanation*	*Group*
Class files	Source code for the application's class templates.	Classes
`main.m`	Standard application start-up code. (This file can be edited, but isn't usually modified.)	Other Sources
`_Prefix.pch`	Precompiled headers used to avoid repeat compilation of the Cocoa headers. Other common project headers can be added here. For simple projects, this file isn't changed.	Other Sources
`-info.plist`	Default application settings, including the names of the first loaded class and the default nib file.	Resources
`InfoPlist.strings`	Used for localization; a "folder" that holds a file with a list of text strings for each language supported by the application.	Resources
Nib files	At least one nib file, defined in `-info.plist`, is loaded automatically. Other nib files can be defined and loaded on demand, if needed.	Resources

CAUTION

When you add new files to a project, Xcode doesn't attempt to move them to the correct groups. The group structure is cosmetic. It doesn't affect compilation, but it does make it easier to keep related files together.

Exploring standard application elements

In an empty template, there are exactly three source code files. Two are usually used as is, and another is only modified occasionally. The `main.m` file, shown in Figure 6.1, is a standard C `main()` function. All Cocoa applications include this file. It runs a loader function called `NSApplicationMain()`, which initializes a memory manager, and then loads and runs the body of the application. It's possible to modify this file to create a different start-up environment, but you can ignore it in most projects.

Similarly, you can usually ignore the `_Prefix.pch` file. It contains an import directive for the Cocoa library header collections and is used to speed up compilation by making sure that these headers are only compiled once. Occasionally you may choose to add extra header collections here. More typically, you'll leave it unchanged.

By default, your application ignores the Info.plist option for string localization. There are two ways to support non-English languages in a Cocoa application: localization strings and localized nib files. If you're developing for an English-speaking audience, you can ignore these options. If not, you can find out more about localization in Chapter 9.

Introducing the application delegate

Active application code is collected in the files in the Classes folder. If your project is named `First` you'll see two files called `FirstAppDelegate.h` and `FirstAppDelegate.m`.

TIP

For clarity, the Classes folder *only includes code for subclasses*. You can — and often do — use Cocoa classes and objects as is, without subclassing them.

The architecture of a Cocoa application is unexpectedly indirect. An `NSApplication` object exists, but you rarely access it directly. Instead, whenever OS X sends an application-level message, the message is passed to an *application delegate* object. The delegate handles messages that are sent by OS X when the application is about to quit, when the user hides it or unhides it, when OS X runs out of memory, and so on.

Unlike the application object, the delegate is designed to be subclassed. When you create a new project, a delegate subclass with the `AppDelegate` suffix is created for you automatically. To a good approximation, this delegate *is* the application.

NOTE

Using a delegate may seem redundant and unnecessarily complex, but it offers practical advantages. You'll look at them later in this chapter.

This delegate can be extended with handlers for every application control message generated by OS X, but the default template-generated delegate is lightweight and does very little. The code defines a link to a window object and includes a single stub method that is triggered when the application loads.

Click `FirstAppDelegate.h` to reveal it in the Editor Window. The interface looks like this:

```
#import <Cocoa/Cocoa.h>
@interface FirstAppDelegate : NSObject <NSApplicationDelegate> {
   NSWindow *window;
}
@property (assign) IBOutlet NSWindow *window;
@end
```

Table 6.2 breaks down the features in detail.

Table 6.2 Sample Class Interface Feaures

Feature	*Explanation*
`#import <Cocoa/Cocoa.h>`	This line imports the Cocoa framework headers. The default template adds this line — and the others — "for free." When you reference a framework in a class, you must add a corresponding line here by hand to import its headers.
`@interface FirstAppDelegate`	The code that follows is the interface for the `FirstAppDelegate` class.
`: NSObject`	This class is a subclass of `NSObject` and inherits its methods and properties.
`<NSApplicationDelegate>`	The class imports and uses a bundle of predefined optional methods called the `NSApplicationDelegate Protocol`. This particular protocol is part of Cocoa. It defines a set of optional application management methods that may be implemented in the delegate.
`NSWindow *window;`	The class includes an instance of Cocoa's `NSWindow` class, accessed via a pointer named `window`.
`@property`	`window` can be a public property.
`(assign)`	`window` uses `assign` operations for memory management. (Memory management is introduced in Chapter 12.)
`IBOutlet`	`window` is a pointer to an object defined in an associated nib file. `window` isn't allocated in code; it's loaded automatically from the nib file when the application loads. The nib loading process also initializes its pointer.

`FirstAppDelegate.m` is shown next. Table 6.3 breaks down its features.

```
#import "FirstAppDelegate.h"
@implementation FirstAppDelegate
@synthesize window;
```

```
- (void) applicationDidFinishLaunching: (NSNotification *)
   aNotification {
 //Insert code here to initialize your application
}
@end
```

Table 6.3 Sample Class Interface Features

Feature	*Explanation*
`#import "FirstAppDelegate.h"`	This line imports the class interface file. (If you don't include this line or forget to rename the header if you rename a class, the compiler can't find the variable and method definitions it needs to build the implementation.)
`@implementation FirstAppDelegate`	The code that follows is the implementation for the `FirstAppDelegate` class.
`@synthesize window;`	This line generates setter and getter methods for `window`.
`- (void) applicationDidFinishLaunching:...`	This is an implementation for the `applicationDidFinishLaunching:` method, which is a method defined in the `NSApplicationDelegete` protocol. The default implementation included in the template is an empty stub. You can expand it with your own application initialization code.

Functionally, you can add custom start-up code to the `applicationDidFinishLaunching:` method, but none of the other messages that the delegate can handle are implemented — yet. These other methods are introduced later in this chapter. The delegate is a subclass of `NSObject`. In theory this means you can run any `NSObject` method on the delegate. This is often useful for other subclasses for `NSObject`, but not for the application delegate — it's unlikely that you'll want to copy the delegate or release it from memory. In this context, `NSObject` is subclassed because it's the simplest and easiest way to create a generic Cocoa object that can be customized with further features. Its inherited features are redundant. This isn't usually true when subclassing `NSObject`; generally, you do want to be able to copy objects and manage them in memory, but the application delegate is a special case.

Note that although the application includes other objects such as a menu, they're not referenced in the code or loaded explicitly by the delegate. When the application loads, the menu loads with it, without `alloc` events or messages. This is an example of dynamic loading, which was introduced in Chapter 5. The menu is defined in the `MainMenu.xib` nib file and loaded automatically. This file is described in the next chapter.

`window` is defined and loaded from the nib file in the same way. The `IBOutlet` directive tells the compiler to leave the `window` pointer value empty at compile time. When Cocoa's nib loader loads the object, it loads a valid address into the pointer. Your code can then access the features of `window` in the usual way. This process is automatic.

```
IBOutlet NSWindow *window; //Undefined at compile time
//After the nib loader has done its work
//window points to the object called window in the nib file
[window doSomething]; //This is now valid
```

Discovering Object Methods and Properties

When the application runs, `window` appears on the screen and waits. By default it does nothing, because there's no supporting code for it. You can control it by adding code that sends messages to it:

```
[window aHypotheticalMessageThatDoesSomethingInteresting];
```

But you don't yet know which messages `NSWindow` responds to. To discover this information, you have to review the `NSWindow` class reference page in the developer documentation.

Finding and using class references

While you can browse to a class reference by drilling down through the layer and framework hierarchy introduced in Chapter 2, there's a shortcut. In Xcode, choose Help ➪ Developer Documentation. This loads the offline version of the documentation built into Xcode. Figure 6.2 shows the window that appears in Xcode 3.2.3.

NOTE

If you're using a later version of Xcode, you'll almost certainly see a similar page, but it may have different graphics.

At the top right of the window is a Spotlight-like search field with a magnifying glass. You can search for the `NSWindow` class reference by typing **NSWindow** into the search field. But Xcode supports a shortcut that can save you significant time. In the interface file, highlight `NSWindow` with the mouse and right-click it. You'll see a complex floating menu with many options, as shown in Figure 6.3. Choose Find Text in Documentation. This is equivalent to copying the text, pasting it into the search field, and waiting for the results. This is an immensely helpful feature in Xcode, and you'll find it invaluable.

Figure 6.2

The Xcode documentation Quick Start page includes sample videos. This page changes with every new version of Xcode, so you may not see the graphics or features shown here.

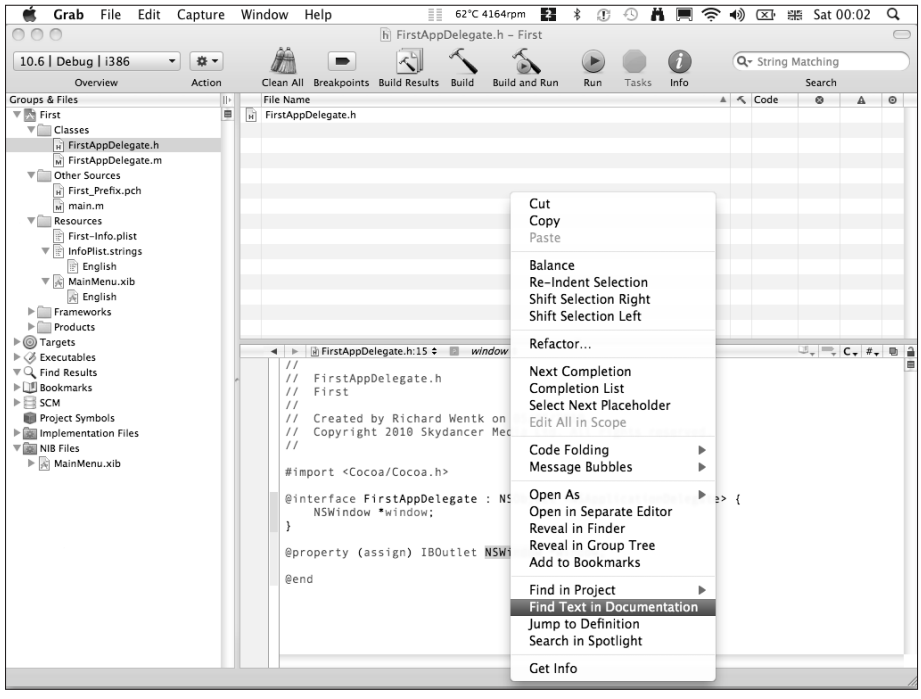
When the search completes, the left-hand pane contains a list of search results that mention `NSWindow`. This list is messy because the search returns all resource types, but you can ignore almost all of it. At the top of the list is an API pane with a C icon next to `NSWindow`. This icon tells you that a document is a class reference. The documentation browser usually — but not always — displays a class reference if it finds one that matches the search string.

TIP

When you type in a search string manually, the search field attempts to search for it immediately, even before you finish typing. This isn't always helpful, but you can use it to search for items that share a starting string. You can also use the Contains, Prefix, and Exact buttons at the top left to further fine-tune the search. You can hide the list at the left by double-clicking NSWindow or the C icon in the API pane, or by dragging the dividing line between the search results and the Table of Contents all the way to the left.

Figure 6.3

The Find Text in Documentation feature can be a huge timesaver. You can use it to search the documentation for any highlighted string.



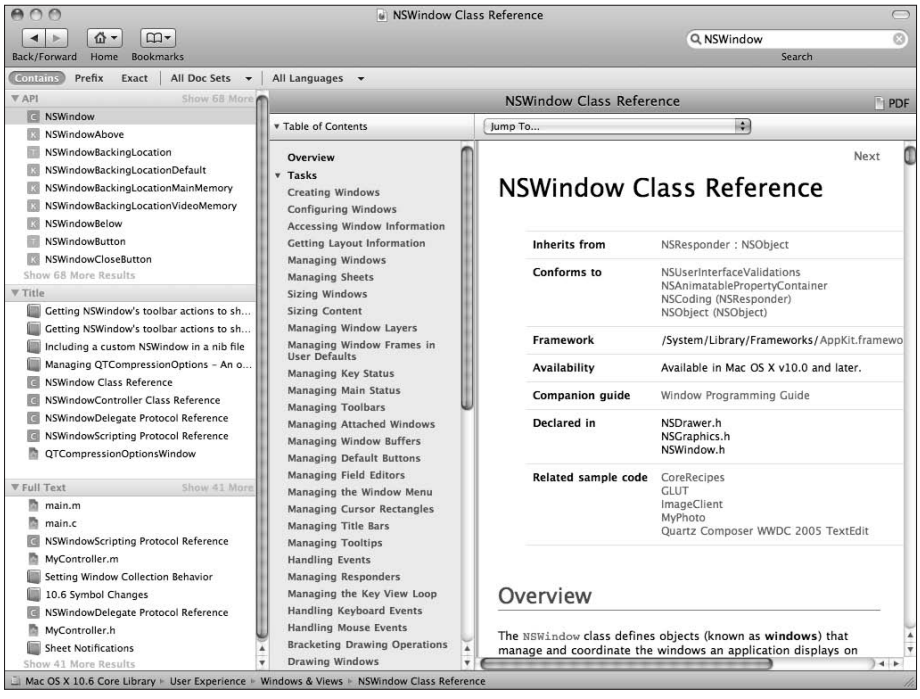
Exploring class references

Class references are single scrollable pages. `NSWindow` is a complicated class with many features, so the reference is long and detailed. The most efficient way to explore it is to use the Table of Contents list that appears in the pane to the left of the main window.

You can use the reference information in various ways. When looking for a list of possible features, it's useful to scan the Tasks list. It includes a complete list of the messages that `NSWindow` responds to, grouped by application and function. You can view the Tasks list by scrolling down or by clicking the Tasks header in the Table of Contents. Clicking the reveal triangle expands it to show a summary list in the Contents pane, as shown in Figure 6.4.

Figure 6.4

Clicking the reveal triangle next to the Tasks header displays a list of solution-based summaries for each class. Use the task list as a quick overview of the class's features.



TIP

At first sight the list of tasks, methods, and other features for some Cocoa objects can appear overwhelming. You don't need to memorize these lists, and it's a given that unless you have developer superpowers, you won't immediately know how to use all the features that appear. Cocoa is a library of *connected* objects and features. Some are self-explanatory, but others won't make sense until you've learned more about how Cocoa objects work together.

For a first project, you'll attempt a very simple task — changing the window's title string. Scroll down through the list of tasks to find the Managing Titles subheading. As you might expect, this subheading includes a list of features devoted to managing the title bar. Each blue item is a clickable link. `setTitle:` looks like a possible solution, so click the `setTitle:` link to read a description of this method.

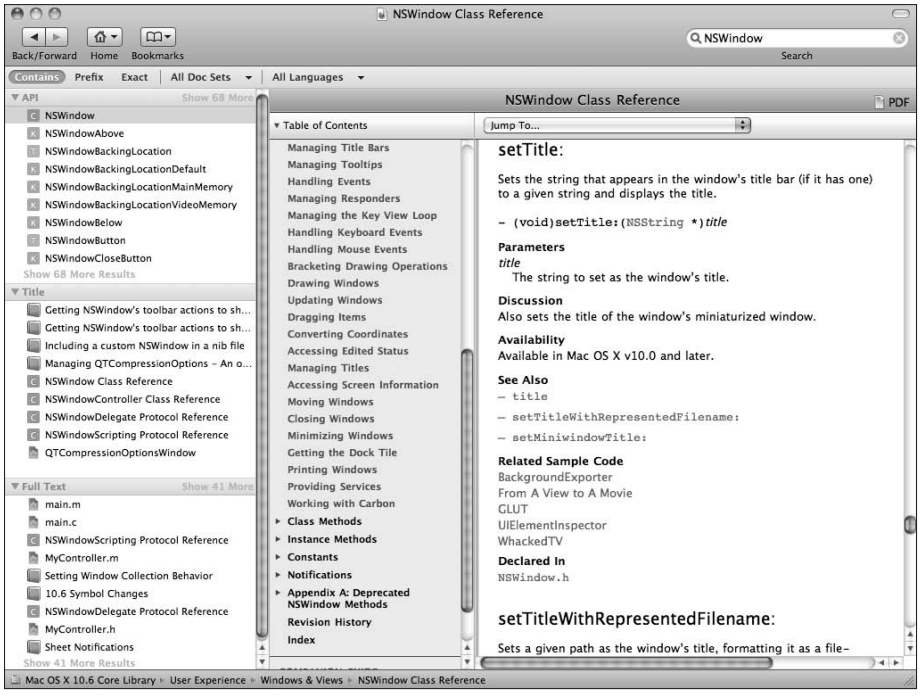
CAUTION

Methods and properties often match their names in an intuitive way, but sometimes they don't. For example, you might expect the `center` method to center the window in the screen. In fact it centers it horizontally, but places it above center vertically for prominence and visual impact. To avoid surprises, check the full description of a feature before you use it.

Figure 6.5 shows the `setTitle:` entry. It has a title followed by a brief description, followed by a code signature. Making sense of the signature is a fundamental skill. Even if you have experience in other object-oriented languages and already understand what each element does, some of the Objective-C syntax may be unfamiliar.

Figure 6.5

The `setTitle:` method description includes a terse list of the method's features. Very few of the class reference pages include sample code or examples.



Understanding signatures

In this example, you want to send the `setTitle:` message to an instance of `NSWindow`. The signature tells you four things:

- The initial "-" character indicates that this is an instance method, not a class method. It must be run on an instance of `NSWindow`, not on the class as a whole. This is good news — it means you can use this method on `window`. For a class method, the initial character would be a "+," and it would have to be run on `NSWindow` itself, not on an instance.

- `(void)` indicates there's no return value. If the method supplied a return value, this field would indicate its type in the usual C-like way.
- The third field is the name of the method and also defines the string used to trigger it — `setTitle:`.
- The colon indicates that a parameter follows the method name. In this example, the parameter is an instance of `NSString`, which is Cocoa's string object data type. When a parameter is a Cocoa object, it appears as a link. If you need more information about `NSString`, click on the link to view its class reference.

Putting this information together, you can guess that the message you need to send to window looks like this:

```
[window setTitle: aString];
```

`aString` can be defined as in-line literal using the `@` objectification character. So your final code looks like this:

```
[window setTitle: @"A Window Title"];
```

Introducing Code Sense

To add this code, type it after the comment line in `applicationDidFinishLaunching:`. You'll notice that as soon as you type the open square bracket and win, Xcode completes the rest of the word window. Similarly when you type `setT`, Xcode automatically expands it to `setTitle:` and also adds a reminder field that tells you that `setTitle:` takes a parameter string.

This autocompletion feature is called Code Sense. Code Sense doesn't read your mind — this may change in future versions of Xcode — but it does try to predict your typing and insert likely items from the application's symbol table. The table includes the full set of Cocoa objects by default.

Code Sense can dramatically improve your productivity, but it can take a while to get used to having an assistant that makes guesses as you type:

- To accept a guess, press the Tab key. The cursor moves past the guess.
- When a symbol has multiple substring options — for example `setT` could be read as `setTitle:` or `setTitleWithRepresentedFilename:` — press the Return key to accept a part-symbol. The cursor moves to the next substring.
- To set a reminder field, tab to it and start typing. Your input overwrites the reminder string. For classes with multiple parameters, tab to each field in turn.

- To see a list of matching symbols, press the F5 key. Scroll up and down the list by pressing the keyboard's up and down arrow keys, or select a symbol with the mouse. Click once to highlight a symbol, and double-click to insert it in the code.
- To ignore a guess, keep typing to overwrite it.
- When you close a curly bracket, the corresponding opening bracket flashes so you can check for balance. Closing square brackets are sometimes — but not always — added automatically. When brackets don't balance, Code Sense makes a warning sound.

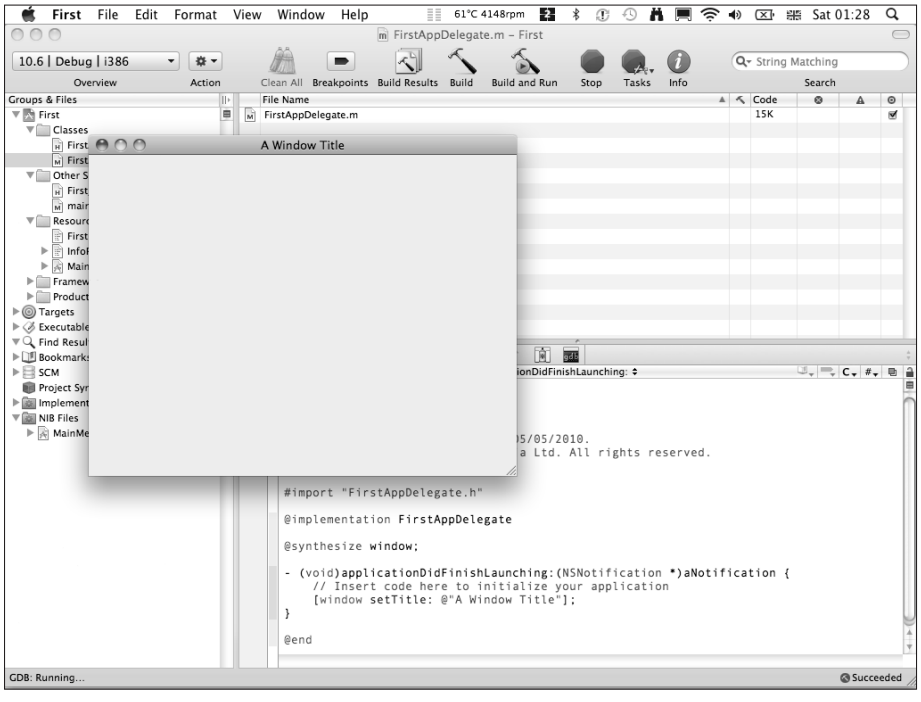
NOTE

Code Sense isn't infallible, and square bracket autocompletion is handled in a somewhat random way. However, you always get a warning when brackets don't balance. Code Sense is most useful when it becomes second nature. You will occasionally need to backspace to overwrite wrong guesses, but it's usually more of a help than a distraction.

Figure 6.6 shows the added line in `FirstAppDelegate.m` and also illustrates what happens if you build and run the modified project. The code does indeed change the title bar. Success!

Figure 6.6

The blank template application with a new window title and the code that creates it.



Working with multiple classes

For a more difficult challenge, you can try to maximize the window as soon as the application runs. Apple's human interface guidelines don't support maximization, and windows have no simple maximization method. The green button on an OS X window resizes the window frame to a default size, but this doesn't have the same effect as a simple maximization.

NOTE

In theory, according to Apple's guidelines, Cocoa applications should have multiple floating windows. But some users find that a single main window with floating subwindows is more intuitive and easier to use — perhaps because it avoids the popular OS X distraction of accidentally losing application focus by clicking outside a window. Because the UI guidelines aren't policed, you can implement whichever solution you feel most comfortable with.

Ignoring any possible controversy, you'll implement a maximize feature. Without a maximization method, the only way to maximize a window is to set its size manually. Scanning the list of tasks reveals a task called Sizing Windows, which looks as if it might solve part of the problem. But it's worth looking at the Introduction to Window Programming Guide for Cocoa, show in Figure 6.7, to see if it contains further hints.

TIP

You can find the Window Programming Guide for Cocoa by scrolling down to the bottom of the `NSWindow`'s Table of Contents. Most classes don't include a Companion Guide, but `NSWindow` is a key class with many features, and the documentation has been expanded to include a guide that introduces them.

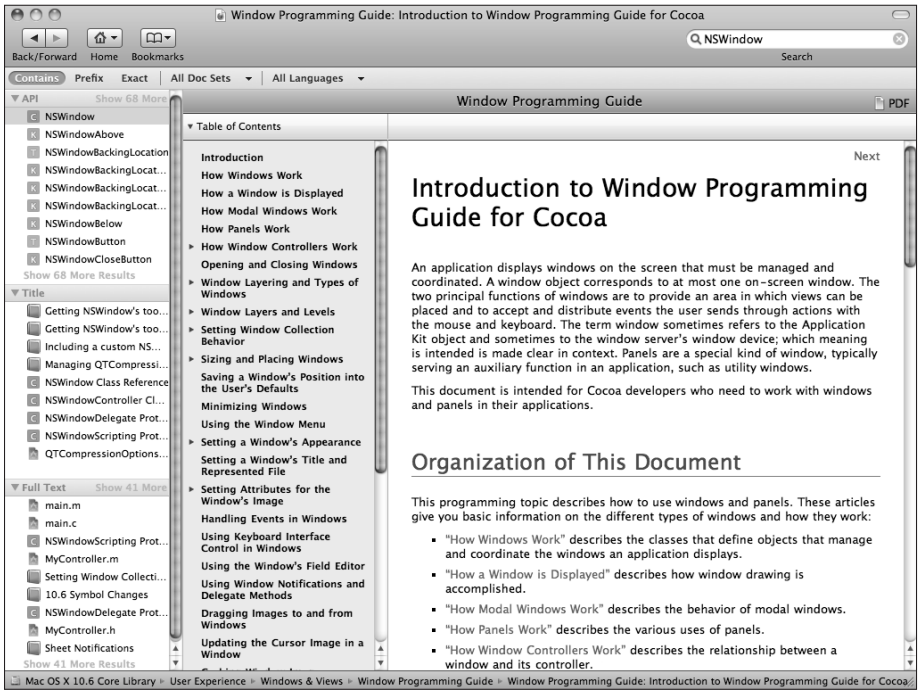
The subsection called How Windows Work introduces the concept of a *frame*. In Cocoa, a frame is the rectangle that surrounds an object and defines its size and position. Returning to the Sizing Windows subheading shows that it features a number of frame-related methods. `frame` returns the current frame but is read-only, and it can't be used to set the frame. `setFrameOrigin:` changes the position of the frame and the position of the window. `setFrame:` does exactly what we want — calling `setFrame:` on a window resizes it.

From the `setFrame:` method description, you can see that its parameters include `display`, which is a Boolean that defines whether or not the window contents are refreshed, and `animate`, which enables an optional animation effect. `display` is irrelevant for an empty window, but you may as well set it to `YES` in case you add other content later. `animate` looks like an interesting parameter to experiment with. So your first attempt looks like this:

```
[window setFrame: aFrame display: YES animate: YES];
```

Figure 6.7

Guides vary in depth and complexity. The `NSWindow` guide is very comprehensive, but some guides are short notes with a couple of terse examples.



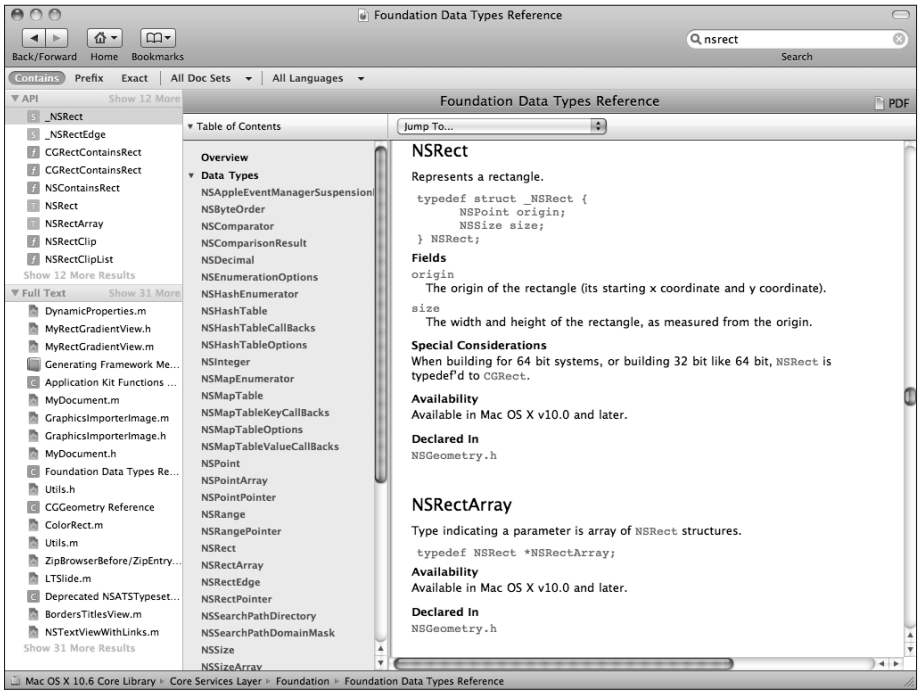
Exploring C-language features

How do you calculate `aFrame`? According to the documentation, `aFrame` should be an instance of `NSRect`. There's no link to `NSRect`, so the only way to find out more about it is to search for it. You can use the Find Text in Documentation feature to search the documentation. Highlight NSRect in the documentation, right-click to show a floating menu, and select Find Text in Documentation. Figure 6.8 shows the result.

This reveals that `NSRect` is a C-language `typedef`, and not an object. You've already fallen out of Cocoa's object hierarchy into an underlying C library that is part of the Foundation layer. `NSRect` is a *Foundation data type*. The Foundation library is still part of Cocoa, but it uses C code and data structures and isn't object-oriented.

Figure 6.8

Cocoa's C language functions and data types don't appear as links in the documentation, but you can search for them manually. `NSRect` is a Foundation data type.



Looking for more information about `NSRect` is unhelpful, because there isn't any. You can drill down further to find out about `NSPoint` and `NSSize`. Or you could create your own custom `NSRect` implementation and use it to convert a group of `floats` or `ints` into an `NSRect`.

There's a simpler solution. The Foundation layer includes a list of Foundation Functions that support its data types. You can use the `NSMakeRect()` function to create an `NSRect` from arbitrary height, width, and position floats.

CAUTION

Cocoa data types are often deeply nested. `NSRect` includes an `NSPoint` made from two `CGFloats`, which are plain C `floats` redefined. It's normal to drill down through multiple data structures to find that the underlying data types are familiar C data types buried under multiple Cocoa redefinitions.

Discovering framework functions and data types

Unfortunately, the documentation fails to introduce the Foundation Functions when you look up information about a Foundation Data Type. You can only learn about them by looking

through sample code or finding them in an online discussion — or reading about them in a book.

Many frameworks offer a selection of helper functions and data types. These are critically useful, but difficult to find unless you already know they exist.

For example, the Quartz 2D graphics library includes a collection of `CGGeometry` features that provide the functions and data types used to implement low-level graphics.

When you start working with a new framework, it's obligatory to check its Framework Reference page to review its functions and data types. This won't always lead you to the features you need to solve a problem because you may need to look for related frameworks to find them. For example, `NSWindow` is part of the Application Kit framework shown in Figure 6.9. Reviewing the Application Kit Functions Reference page won't tell you that it relies on the Foundation Functions, shown in Figure 6.10, for various support functions.

But as a rule of thumb, the Foundation Functions are used throughout Cocoa. It can be helpful to review them whenever you're exploring new features.

Figure 6.9

The Application Kit Functions Reference lists some of Cocoa's less obvious features. You won't find these functions unless you look for them, but they're critically useful to many of Cocoa's features.

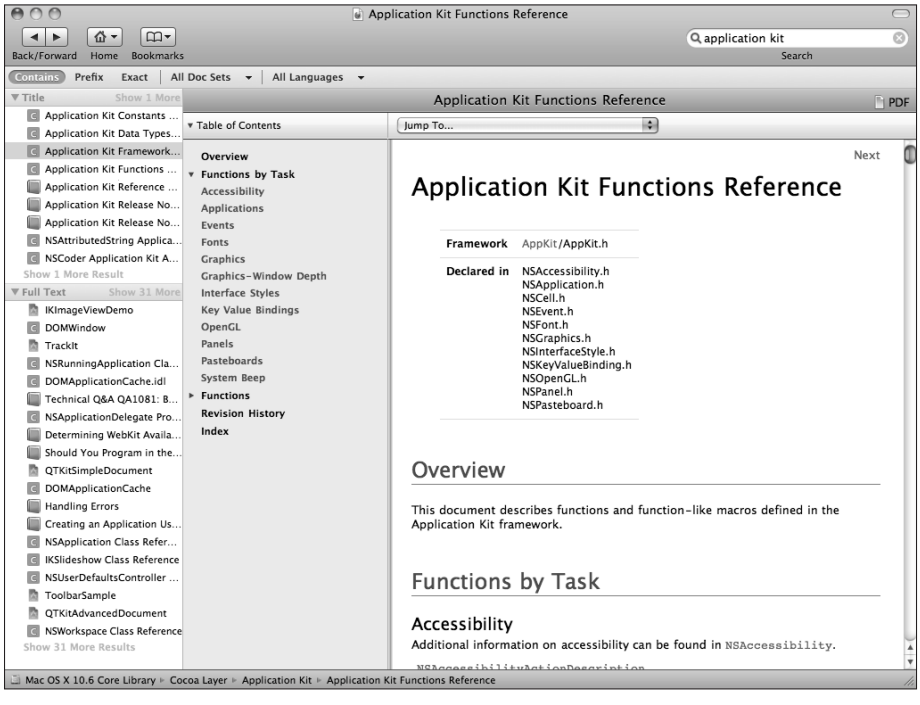
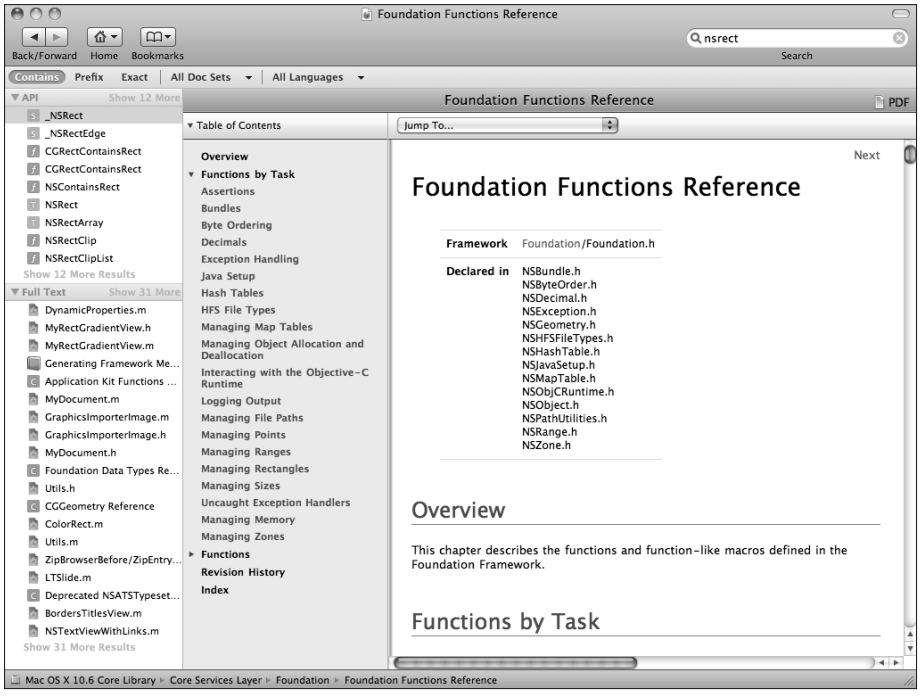


Figure 6.10

Similarly, the Foundations Functions Reference page lists a selection of other essential Cocoa functions. The task list is particularly revealing. You won't use these functions regularly, but you should know that they exist and be familiar with what they can do.



Having explored the Foundation Functions, your revised code looks like this:

```
[window setFrame: NSMakeRect(x,y,w,h) display: YES animate: YES];
```

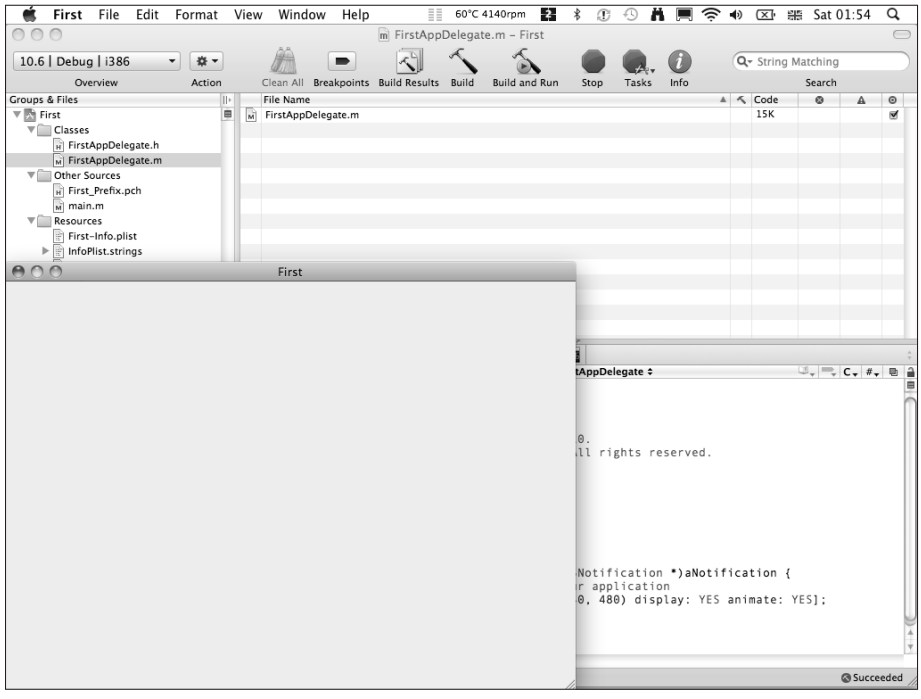
This resizes the window to an arbitrary size you can specify using static floats for `x, y, w, h`. Experimenting with this line reveals something unexpected — setting x and y to 0 anchors the window at the bottom of the screen, not at the top. This is because OS X windows use a bottom-left origin. To move a window to the top of the screen, the bottom coordinate must be calculated. For example, on a screen with a resolution of 1024 × 768, Figure 6.11 shows the result of

```
[window setFrame: NSMakeRect(0,0,640,480) display: YES animate:
   YES];
```

This simple problem is beginning to look more challenging. Not only is there no simple maximize feature, but also fixing the top of a window requires some extra coordinate transformations. If you want to anchor the window and resize it to fill the screen, you need to know the screen resolution. Is there a better solution?

Figure 6.11

A first attempt at resizing the window succeeds in changing its size, but doesn't calculate the size from the screen dimensions.



Expanding the search to related classes

For certain applications, the answer turns out to be "No": some features can't be implemented easily or elegantly. You may need to resort to hacks or other less than optimal solutions.

But in this example, if you review `NSWindow`'s Table of Contents again, you'll see a heading called Accessing Screen Information. The first method in this Task is called `screen` and returns the screen the window is in as an instance of an `NSScreen` object.

```
NSScreen *thisScreen = [window screen];
```

Clicking through to the `NSScreen` class reference reveals that `NSScreen` implements a method called `frame` that returns an `NSRect` for the screen's `frame`. So you can use the fragment

```
[thisScreen frame]
```

to return an `NSRect`. This is almost perfect, but the `NSRect` returned by `frame` covers the entire screen. Looking again reveals another method called `visibleFrame` that returns the

area that excludes the standard OS X menu at the top of the screen and the Dock at the bottom. This is exactly the area you want and solves the problem.

You can now write a new version that gets the visible frame from the current screen and passes it back to `window`:

```
NSScreen *thisScreen = [window screen];
[window setFrame: [thisScreen visibleFrame] display: YES
   animated: YES];
```

It's possible to simplify this code to a single line. There's no need to make `thisScreen` an explicit pointer; you don't access it again. So you can use a nested return:

```
[window setFrame: [[window screen] visibleFrame] display: YES
   animated: YES];
```

To recap how the nested code works:

```
[window screen]
//returns the current screen as an NSScreen
[[window screen] visibleFrame]
//runs the visibleFrame method on the screen, returning an NSRect
[window setFrame: [[window screen] visibleFrame]…];
//passes the NSRect to setFrame and sets the size and position of
   window
```

You can repeat this process to explore the other features of `NSWindow`. If you look through the class reference, you can find methods that enable or disable the drop shadow, modify the window opacity, close or minimize the window, and so on. Some methods are simple but powerful — for example, `print` sends the window contents to the OS X printing system and implements the features of a basic printing solution. Others are more complex and require a deeper knowledge of Cocoa and its messaging system.

TIP

It's an excellent idea to experiment with some of NSWindow's other methods, to familiarize yourself with the class's features.

Receiving messages from OS X with a delegate

`window` is a Cocoa object embedded in OS X. Your application controls it by sending messages to it. But how can your application handle messages that come from OS X? You've looked briefly at the application delegate and seen that it implements a handler for a single OS X message called `applicationDidFinishLaunching:`. Other messages can be handled by adding further handlers. But where can you find a list of those messages?

Working with protocols

Looking again at the code for the interface, you can see the `NSApplicationDelegate` protocol.

```
#import <Cocoa/Cocoa.h>
@interface FirstAppDelegate : NSObject <NSApplicationDelegate> {
   NSWindow *window;
}
@property (assign) IBOutlet NSWindow *window;
@end
```

The `<NSApplicationDelegate>` statement in the second line tells the compiler that the `FirstAppDelegate` class can use methods bundled inside the `NSApplicationDelegate` protocol.

You can think of a protocol declaration as a terse but powerful `#include` statement. Using an imaginary protocol called `AnImaginaryProtocol`

```
@interface AnObject: NSObject <AnImaginaryProtocol>
```

is almost equivalent to:

```
#import <Cocoa/Cocoa.h>
@interface AnObject : NSObject {
 }
//List of <AnImaginaryProtocol> delegate methods starts here
- (void)anOptionalMethod;
- (void)anotherOptionalMethod;
- (AClass *) anOptionalMethodThatReturnsAValue;
+ (void) andSoOn…;
//List of <AnImaginaryProtocol> delegate methods ends here
@end
```

In a sense, a protocol is a convenient way to skip unnecessary copying and pasting in a class interface. When you adopt `<AnImaginaryProtocol>`, the methods between the comments are pasted into the interface automatically. You can then add implementation code for them in the class implementation.

But protocol methods get a special, useful dispensation from the compiler. Unlike conventional methods, they don't have to be matched by a corresponding implementation. Instead you can *pick and choose* which protocol methods to implement. It's equally valid to add implementation code for all, some, or none of the methods adopted from a protocol. The compiler flags a warning if some of the methods aren't implemented, but you can build a project without implementing any of them.

This makes protocols and delegates immensely responsive. Without delegation or protocols, an object must be packed with empty method stubs for every possible message it can receive. With delegation, code is only required for methods that are actually implemented. Figures 6.12 and 6.13 illustrate how this works.

Figure 6.12

Without a protocol, a class has to include a list of all possible method definitions in the interface and a list of all possible method stubs in the implementation.

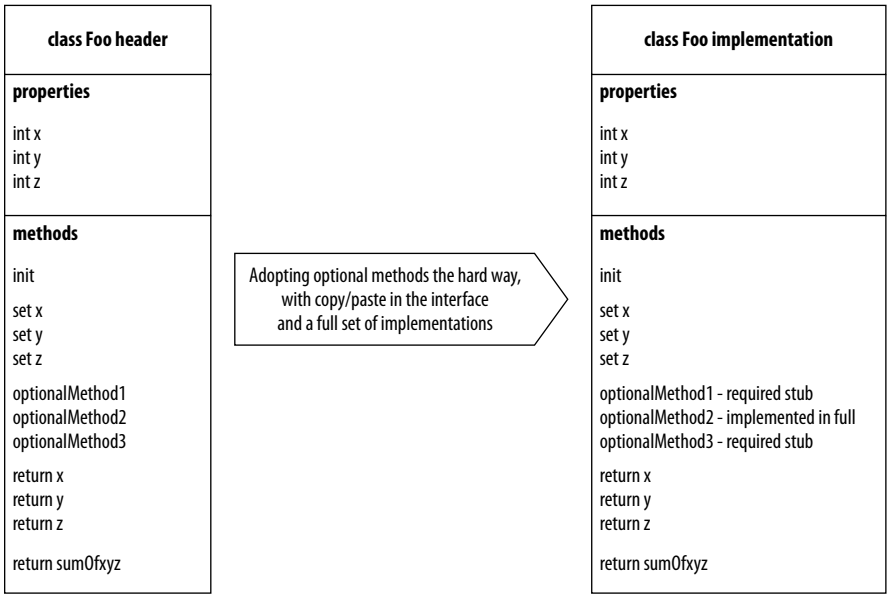
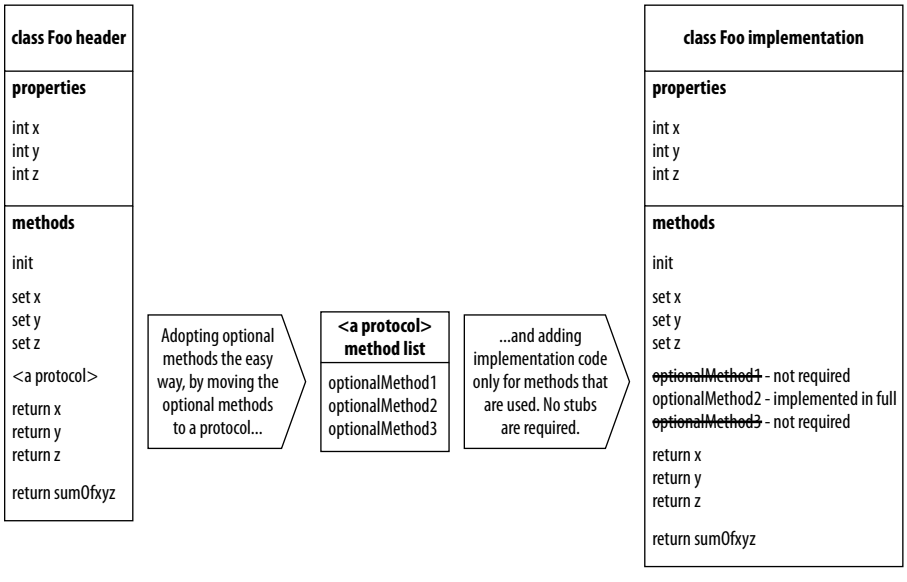


Figure 6.13

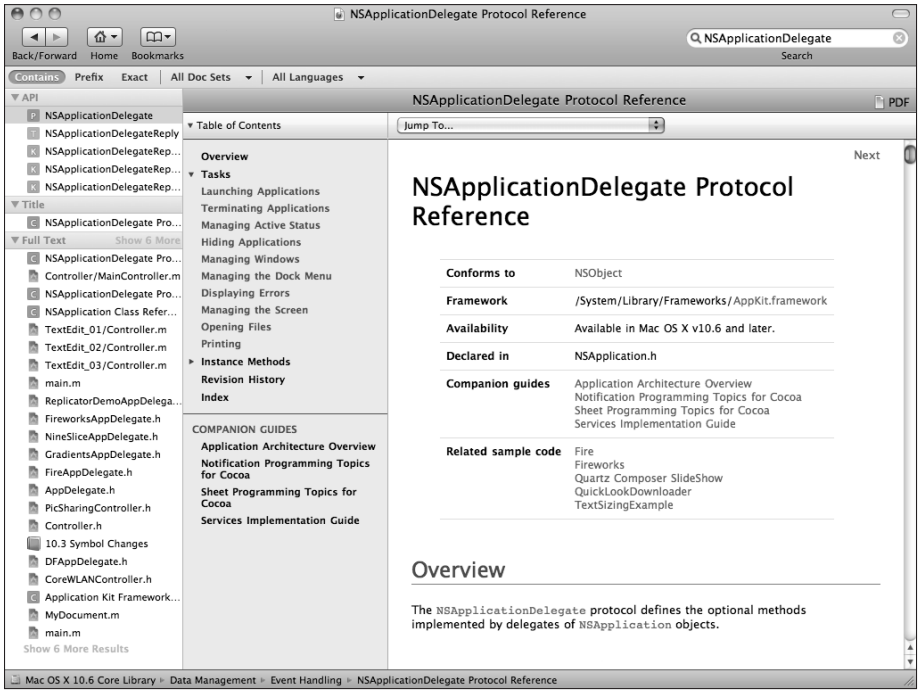
With a protocol, optional methods can be bundled into a single simple `#include` statement and only included in the implementation when needed.



Many Cocoa objects have associated delegate protocols. You can also define custom protocols for your own objects. `NSApplicationDelegate` is a standard Cocoa protocol, so you can find a list of definitions of its optional methods in the developer documentation. Protocol Reference pages are similar to Class Reference pages, and you can search them in the same way — by copying and pasting search words into the documentation search field or by using the Find Text in Documentation menu option. Figure 6.14 shows the `NSApplicationDelegate` protocol reference page.

Figure 6.14

Like a class reference, a protocol reference displays methods grouped into tasks. There's also a conventional alphabetical list of the methods in the protocol.



Reading a Protocol Reference

A Protocol Reference page is like a Class Reference page, and it is organized in a similar way. Protocols aren't objects, so they don't have properties, but they do include a list of methods grouped by tasks. You can use the Table of Contents at the top left of the page to review the tasks, and you can also access an alphabetical list lower down on the page.

The messages shown in this reference are generated automatically by OS X at various points in an application's lifecycle. We've already used the `applicationDidFinishLaunching:` method, which is triggered after the application loads. This stub is included in the template as a convenience because this method is used in so many applications. If you don't need to use this

feature, you can delete the stub. You can also add further delegate methods, as shown below. An object can adopt an unlimited number of protocols, although in practice if an object is adopting more than three protocols the application's design may need to be simplified. To adopt multiple protocols, separate them with commas:

```
@interface AnObject: NSObject <AnImaginaryProtocol,
   ACompletelyDifferentProtocol, YetAnotherProtocol>
```

If an object adopts multiple Cocoa protocols, it will receive all the corresponding messages sent by OS X. You can also define your own protocols and send messages from them to other custom objects in your application.

Understanding delegation in Cocoa

Delegates and protocols are completely general and not limited to high-level application management. In theory, a delegate object can do anything a normal object can do, and there are no restrictions on protocol methods.

In practice, most Cocoa objects use delegation in a selective way. These options were listed in Chapter 2, but they're important enough to repeat here.

Delegate messages are sent:

- When an event is about to happen
- When an event has happened
- To ask if an event or response should happen
- To request data from another object

If you review the list of methods defined in the `NSApplicationDelegate` protocol reference, you'll see they fit into one of these groups:

- `applicationDidFinishLaunching:` is received by the application delegate when the application has finished loading.
- `applicationDidChangeScreenParameters:` is received when the screen resolution changes.
- `applicationShouldTerminate:` is received when OS X wants to know if an application should quit. The application delegate controls the response by returning `YES` or `NO`. You can use this feature to keep an application running while it saves its state before quitting.
- `applicationWillUnhide:` is received when the application is unhidden.

This event-based model describes how delegates are often used in Cocoa. You won't find it explained in the documentation, but it's much easier to understand delegates and protocols if you appreciate these design goals.

Other objects that support delegation follow this model. Elsewhere in Cocoa you can find delegate messages that are sent:

- Before a menu opens
- After a window is hidden
- To return a method for a given key triggered by an animation event
- To return image data
- When an audio player objects finishes playing a file

This list is a very short selection of Cocoa's delegate features. You can find the full list of objects that support a delegate and their associated protocols by searching the documentation for "protocol reference."

Implementing delegate methods

Now that you've been introduced to the methods in `NSApplicationDelegate`, you can experiment with adding them to the application. The fastest and simplest way to implement a delegate method is to copy its signature directly from the documentation and paste it into a class implementation.

As an exercise, you'll implement `applicationWillHide:`, which is triggered when the user hides the application's window. Find the method in the task list and click it to see the full definition. Highlight the signature with the mouse as shown in Figure 6.15 and press ⌘+C to copy it.

Press ⌘+V to paste it into the implementation file, as shown in Figure 6.16. Because it's the start of a new method block, make sure it's pasted before the `@end` directive but after the curly bracket that closes `applicationDidFinishLaunching:`. The window-maximizing code has been removed because it's not used in this example.

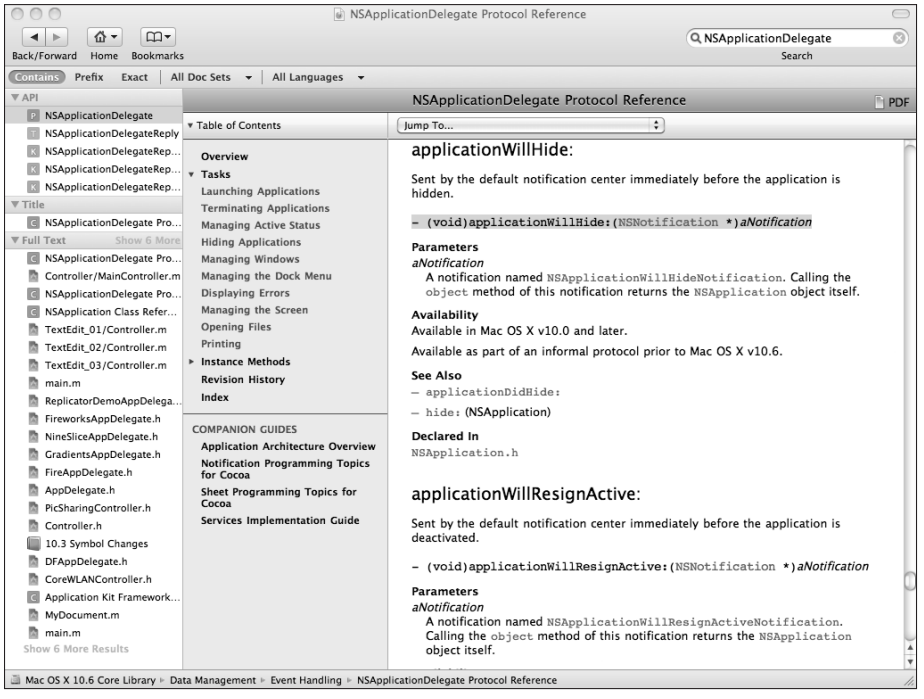
NOTE

`aNotification` is a *notification object* passed by OS X to this method. It arrives as a received parameter. In this example, you'll ignore it because it doesn't contain any useful information. Notifications are system-level events generated and managed by OS X. Don't confuse them with Objective-C messages — the two messaging systems are unrelated. For more about notifications, see Chapter 9.

There's no limit to the possible complexity of the method implementation. In a commercial application, this method might halt a running timer to save processor cycles, write the application state to disk ready for a fast restore when the application unhides, or even send an e-mail. You'll create an implementation that's very much less sophisticated — one that logs a message to the console.

Figure 6.15

To implement a method, copy and paste its signature from the documentation. You can also type it in manually, but copying and pasting is quicker and less error prone. All parameter names are local. If they clash with existing names, add an underscore.



Implement the method code with a call to `NSLog`:

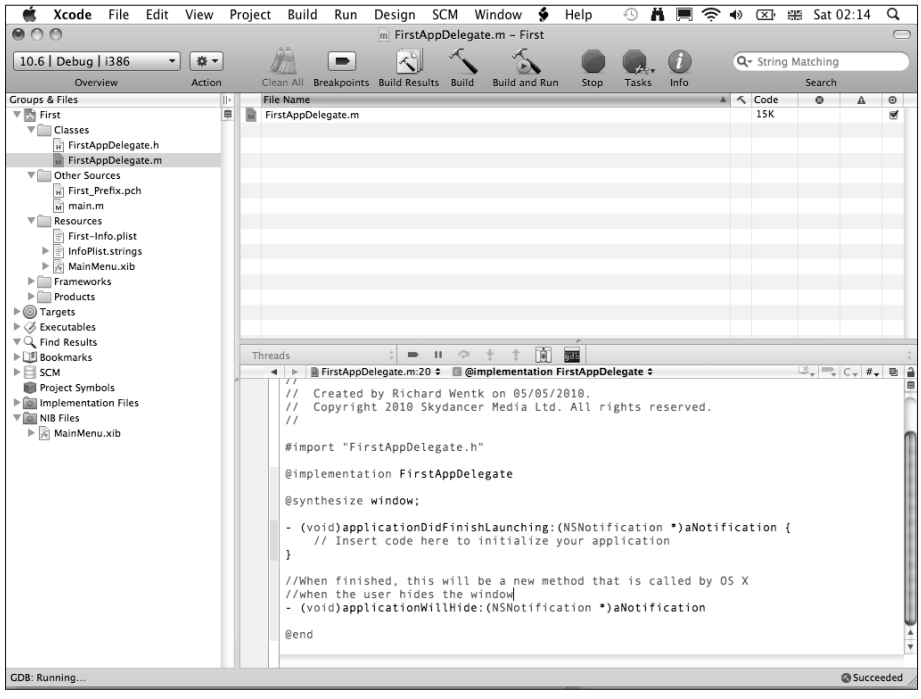
```
- (void) applicationWillHide: (NSNotification *)aNotification {
 NSLog (@"You hid the application!");
}
```

NOTE

`NSLog` is Cocoa's console output object. You can write the console with `printf`, but `NSLog` includes extra features that can convert object values into formatted strings. It's the simpler and more powerful choice for Cocoa debugging. Don't forget to end the method with a curly bracket. Also, note that Code Sense doesn't autocomplete `NSLog` itself, but does add a string reminder once it recognizes it. This is a persistent bug in Xcode.

Figure 6.16

Pasting a method signature creates the first part of the method implementation. To complete the implementation, add code between the curly brackets.



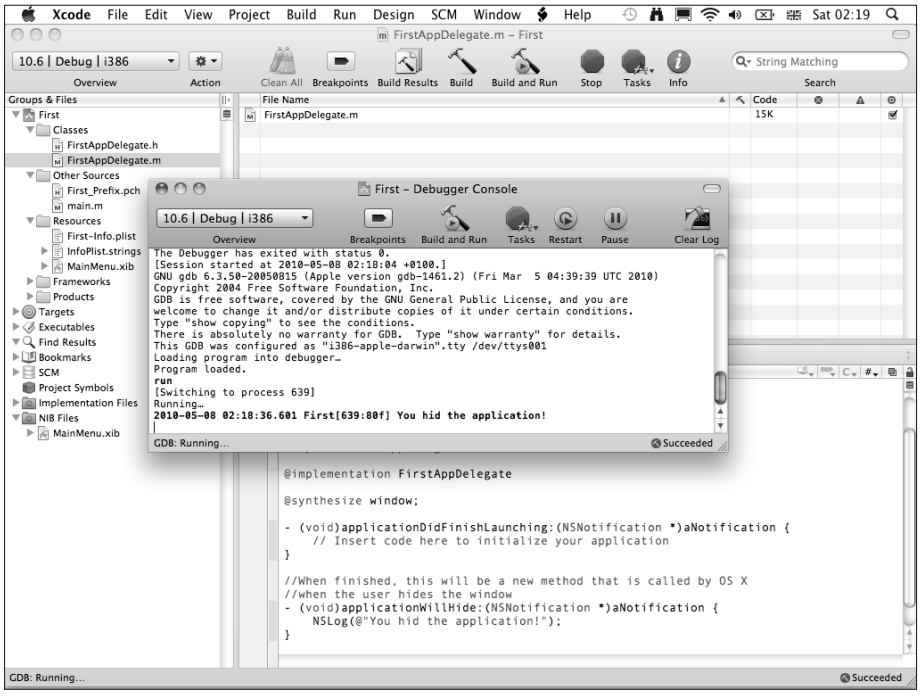
To view the output, open Xcode's console window by choosing Run ➪ Console. The console window includes a convenient copy of Xcode's main toolbar. It can't be customized, but you can use Build and Run to build and run the project without having to switch back to the main Xcode window.

Click Build and Run. When the window appears, choose First ➪ Hide First to hide the application. Figure 6.17 shows how the result.

The `applicationWillHide:` method is triggered by OS X, running your new code. The code you added posts a console message. To unhide the application, click its icon in the Dock. If you hide it again, the method is triggered again. Optionally, you can experiment with some of the other methods defined in the protocol. For example, you can use `applicationWillUnhide:` to log a message when the application unhides and at least one of its windows becomes visible.

Figure 6.17

Hiding the window makes it disappear into the Dock, but because the application is running as an Xcode subprocess, the console window continues to be visible — and it displays this message.



NOTE

`applicationWillHide:` is triggered before the window disappears. `applicationDidHide:` is triggered after the window disappears. Cocoa often gives you both options for important events. In this example they're functionally identical. In a different context, you might want to use `applicationWillHide:` to stop animations or other foreground processes before allowing the window to disappear.

Receiving messages from OS X with NSResponder

OS X uses a variety of mechanisms to send messages to an application. Application control messages don't process mouse event information — so how can you make an application respond to mouse clicks or movements? Reviewing the Table of Contents for `NSWindow` shows that there's a Handling Mouse Events task. This includes a `mouseLocationOutsideOfEvent Stream` method, which returns the position of the mouse cursor inside the window on demand.

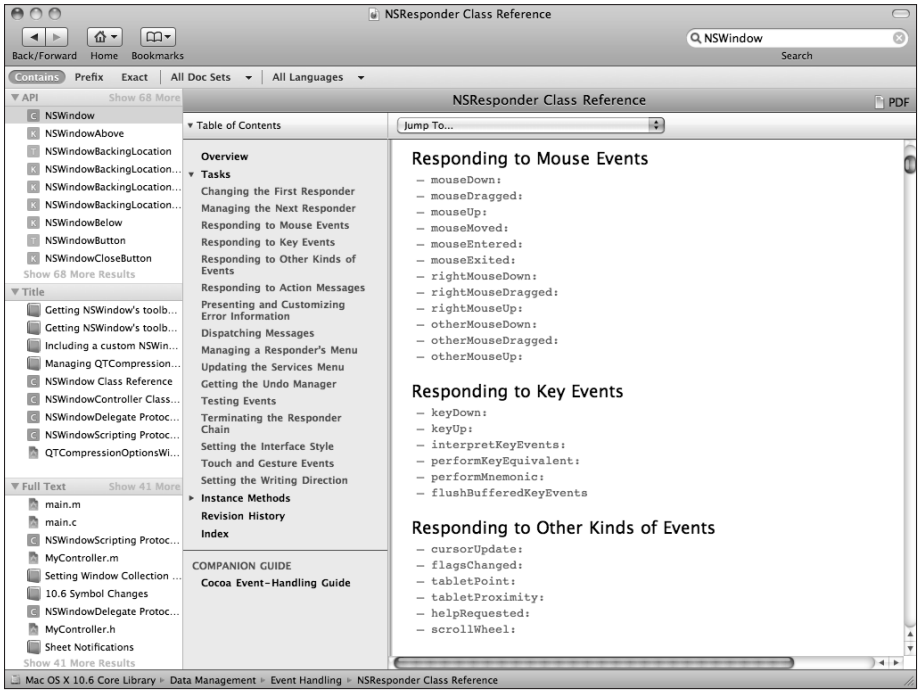
You could put this method inside a loop and poll it continuously to monitor the mouse position. But this would be inefficient, and it wouldn't handle mouse clicks. Given Objective-C's event-based programming model, it's realistic to assume that there's an event-based solution.

Interface objects — including windows — can use the `NSResponder` class to respond to user actions. Scrolling to the top of `NSWindow`'s class reference reveals the Inherits from… list. This tells you that `NSWindow` is a subclass of `NSObject`, and also of `NSResponder`. It also means that `NSWindow` includes all the properties and methods of `NSObject` and `NSResponder` "for free."

Every class that handles user messages in Cocoa is a subclass of `NSResponder`. So you can use its features to make `window` respond to mouse clicks. Figure 6.18 shows the mouse event methods listed in the `NSResponder` class reference.

Figure 6.18

By subclassing `NSResponder` — where it's not subclassed already — you can handle a rich selection of mouse and other event messages.



CAUTION

NSResponder is an *abstract class*. You can't create an instance of NSResponder, but you can create instances of its subclasses. Effectively, NSResponder is a bundle of methods that implements event handling for window, view, and application objects. You can subclass it to create your own event-driven objects. Developers occasionally debate whether NSResponder would be better implemented as a protocol. The answer may well be "yes," but it's implemented as an abstract class, so that's how it has to be used.

Like protocol methods, the `NSResponder` methods are *designed to be overridden*. You use them by copying their signature, adding them to a receiver, and then implementing them with custom code. In this example, you want to make `window` respond to mouse events, so `window` is the receiver. The code

```
[window mouseDown: theEvent];
```

doesn't work because it sends a message *to* the window. You're looking for a message handler *in* the window.

TIP

You can use this code to send simulated mouse clicks to a window. In fact, you can trigger any NSResponder method artificially by running the method from your code. Most applications don't need this, but you can use this feature to create mouse and key event automation.

But where does the code go? There are no class files for `NSWindow`. How can you modify it?

Subclassing NSWindow

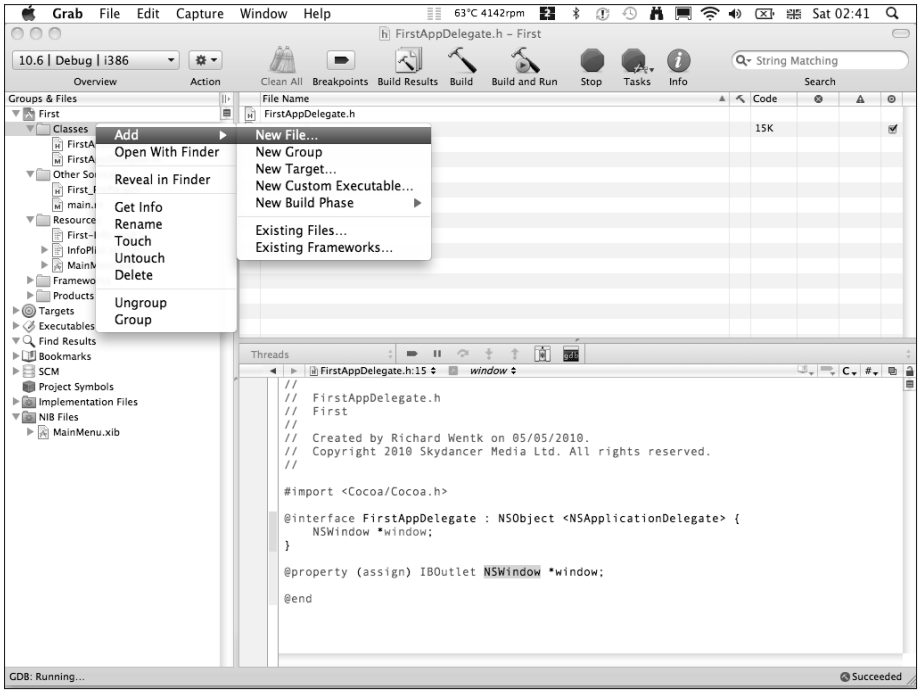
An obvious solution is to subclass `NSWindow` and extend it with a mouse-handling method. If you subclass `NSWindow`, you can add one or more custom methods to it, including some of the `NSResponder` methods.

Xcode includes features that make it easy to add a subclass to a project. Right-click the Classes group at the top of the Groups & Files pane in Xcode and choose Add ➪ New File, as shown in Figure 6.19. You can also choose File ➪ New File from the main menu.

Select Cocoa Class in the Mac OS X pane at the left, and select Objective-C class in the list of file types at the top, as shown in Figure 6.20. Locate the Subclass Of drop-down list halfway down the window, and click NSObject if it's not already selected.

Figure 6.19

Adding a new class in Xcode. If you right-click a group, the class is automatically added to the selected group. This isn't always a good thing. If the class includes a nib file, you'll have to drag it to the Resources group by hand.



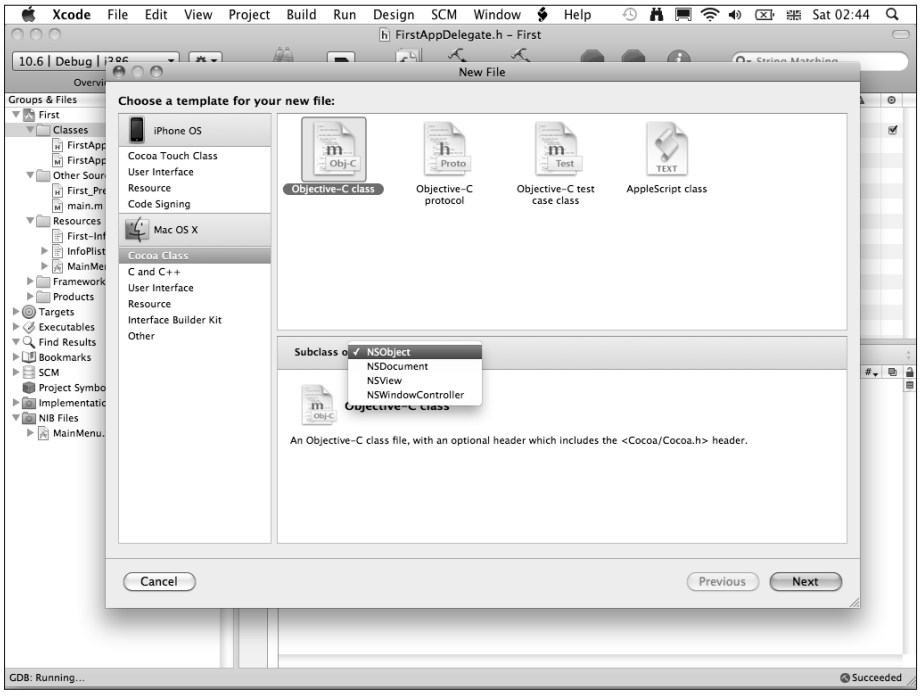
Click Next at the bottom right and enter a filename in the box at the top of the window. Because you're subclassing `NSWindow`, it's a good idea to use a name that reminds you that this is a window-like class, such as `FirstNewWindow`. Click Finish to add the new class to the project. Figure 6.21 shows the result.

TIP

Xcode doesn't enforce a naming convention or add a project prefix to new files. For clarity, it's useful to follow a standard naming convention such as <projectname><subclassname><object type>. Including the project name makes it easier to keep track of files when you begin reusing them in other projects.

Figure 6.20

Think of `NSObject` as a blank generic subclass. The other subclass options shown here include prewritten method stubs. The `NSObject` option creates a blank subclass without prewritten code.



By default, this process creates a subclass of `NSObject`. This isn't what you want here, so as a first step you need to change the interface code in `FirstNewWindow.h`. When the file is created, the first line of the interface looks like this:

```
@interface FirstNewWindow : NSObject {
```

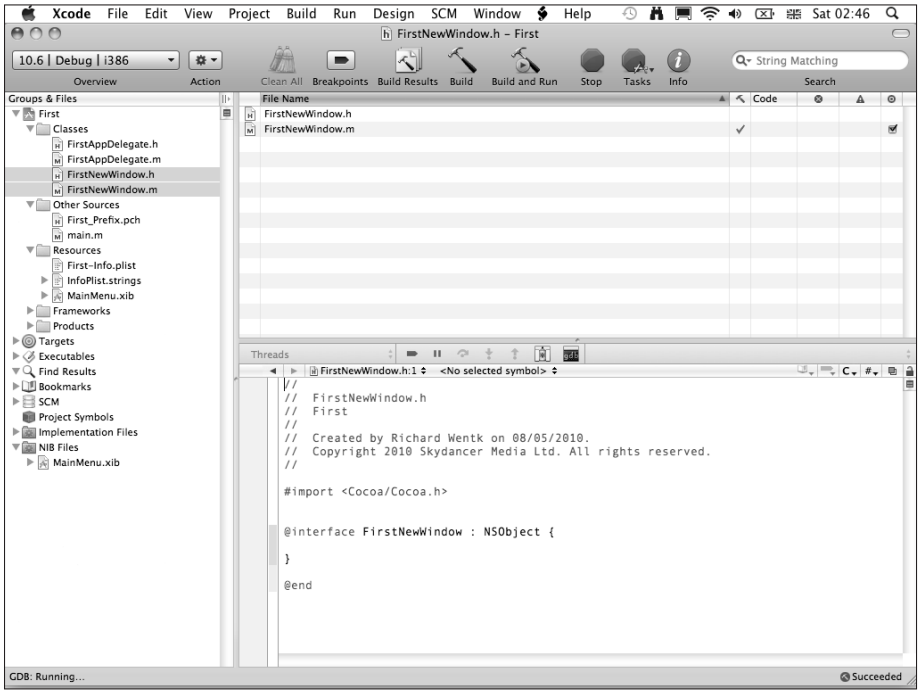
To redefine your new class as a subclass of `NSWindow` instead of `NSObject`, change the code to

```
@interface FirstNewWindow : NSWindow {
```

Save the file, and that's it — you've subclassed `NSWindow`. You can repeat these steps to subclass any other Cocoa object.

Figure 6.21

Exploring the new subclass. Currently it's still a subclass of `NSObject`.



TIP

When you create a new file in Xcode, you can use the drop-down menu to select `NSObject`, `NSDocument`, `NSView`, or `NSWindowController` as the source class. These classes are subclassed regularly, so they're built into Xcode to save you time. Choosing `NSObject` creates a blank header and implementation file. The other classes generate files that include stub definitions of some of their key methods.

Now that you have subclassed `NSWindow`, you can extend it by overriding one of the `NSResponder` mouse methods. You'll use `mouseDown:`. As before, the easy way to do this is to copy and paste the signature from the method signature from the class reference into the new class.

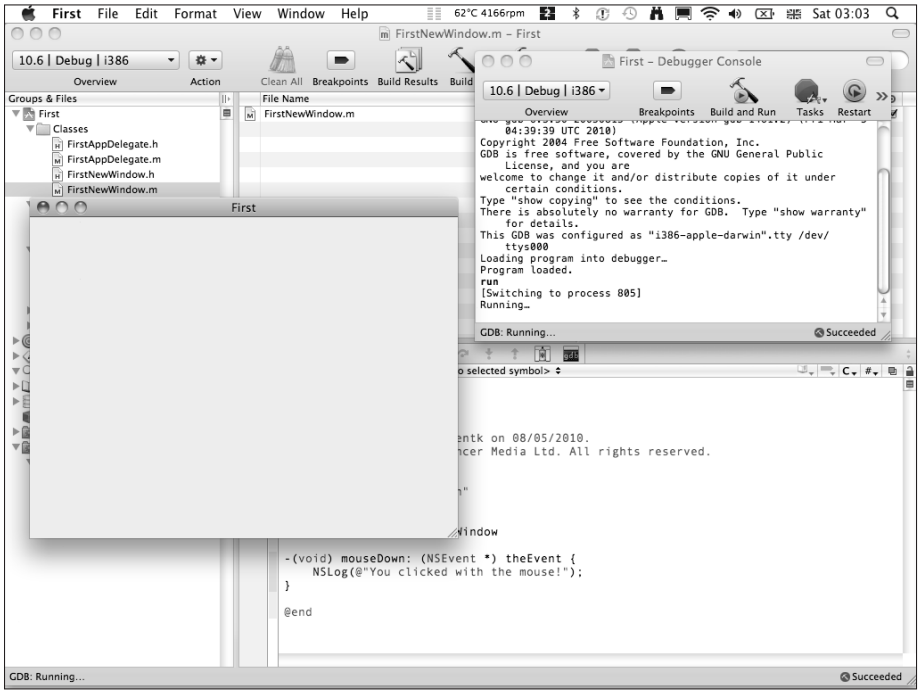
In Xcode, select `FirstNewWindow.m`. Paste the line under the `@implementation FirstNewWindow` directive so that the file looks like this:

```
-(void)
@implementation FirstNewWindow
-(void)mouseDown: (NSEvent *) theEvent
@end
```

Add curly brackets to the method and then add a line that uses `NSLog` to send a message. The finished method looks like the version shown in Figure 6.22.

Figure 6.22

Implementing the `mouseDown:` method in your new subclass of `NSWindow`. The code is correct — but it doesn't work!



Build and run the file. Open the console window by choosing Run ➪ Console in the main Xcode window. When the application window appears, click in it.

CAUTION

Whenever you click Build and Run after making an edit, Xcode displays a Save before Building? dialog. Usually you'll want to click Yes to save the edited files. If you click Cancel, the files won't be saved. Xcode always builds from the saved files, not the versions visible in the editor. You can also use Cancel if you realize you need to make more edits before a build.

Unfortunately, also as shown in Figure 6.22, nothing happens in the console window. Clicks have no effect. You've subclassed the window object, and you've added a mouse method, so why isn't the code working?

There are two ways to make the code active. One uses a feature in Interface Builder and is introduced in the next chapter. The other gives you the power to modify the features of NSWindow in place, without subclassing. It's called *creating a category* on a class.

Creating a category on NSWindow

Think of a category as an in-place subclass. Instead of copying the features of `NSWindow` to a subclass with a different name, you can create a category to add new methods directly to `NSWindow`.

TIP

Don't be confused by the name. Nothing is categorized, and there are no categories in the usual sense. A category is an in-place subclass that adds extra methods. It might as well be called a zombo or some other made-up word.

Understanding categories

Categories make it possible to modify and extend Cocoa objects without changing their class. This can be useful when Cocoa objects enforce strong property typing and refuse to accept a subclass.

For example, `NSWindow` has a `setContentView:` method that takes an instance of `NSView`. The compiler generates a warning if you try to pass a subclass of `NSView`. In this example you can ignore the warning and your code will work — probably. But other Cocoa classes are less forgiving of type mismatches.

When you create a category, you change the properties of the class as a whole. If you create a category on `NSWindow`, every window in the application is modified.

Creating a category

To create a category, follow these steps:

1. Create a new file, using the steps required to create a subclass.
2. Save the file as <classname>+<category name>. For example, a category on NSWindow should be saved as NSWindow+<aName>. The category name is arbitrary.
3. In the interface file, add a list of the methods you are adding to the class.
4. In the implementation, add the code that implements each method.

For example, to extend NSWindow with a `maximize` method, the interface looks like this:

```
#import <Cocoa/Cocoa.h>
@interface NSWindow (maximize)
    -(void) maximize;
@end
```

The `(maximize)` after `NSWindow` is arbitrary. The name doesn't have to match the method or methods in the interface.

The implementation looks like this:

```
#import "NSWindow+Maximize.h"
@implementation NSWindow (maximize)
-(void) maximize {
   [self setFrame:[[self screen] visibleFrame]
          display: YES
          animate: YES];
}
@end
```

After you add this category to your project, you can maximize any window in your application with

```
[aWindow maximize];
```

You can repeat steps 3 and 4 to add your custom `mouseDown` method to `NSWindow`.

NOTE
Sample code for this chapter is available on the Web site at `www.wiley.com/go/cocoadevref`.

Summary

You've learned a lot in this chapter, so take some time to review your new skills. You looked at the structure of a Cocoa application and learned about the role of the delegate object. You were introduced to protocols, discovered some of their possible applications, and explored how to use delegate objects and protocols in practice.

You encountered `NSResponder` for the first time, and learned how to implement its methods in a subclass of `NSWindow`. You were taken through a step-by-step demonstration of subclassing, were introduced to categories, and discovered that important parts of every Cocoa application are defined in its nib files, which are explored in the next chapter.

Introducing Interface Builder

As emphasized in previous chapters, Interface Builder — also known as *IB* — could be called Application Builder. Many developers use IB exclusively as an interface design tool. But you can use it in a completely general way, creating arbitrary collections of objects for your applications and defining arbitrary links between them.

TIP

Even when used exclusively for UI design, you can use IB to create loadable elements within a UI, such as individual entries in a table or scrolling subwindows. You don't have to use IB to design complete views. Potentially you can combine IB elements with code to create customized animated interfaces that have more in common with Flash design than with the standard Aqua/OS X look and feel.

Introducing Nib Files

A nib file is a collection of objects. The contents of a nib file are read-only: you can't use a nib file as a persistent store for mutable data. But IB includes initialization options for certain objects, so you can preset some of their properties to useful values; for example, you can set the initial position and size of an application window. Table 7.1 lists some of the key features of the nib system.

Nib file loading is semiautomated. An application's initial nib file is defined in its `info.plist` file, as shown in Figure 7.1. This nib is always loaded automatically.

In This Chapter

Introducing nib files

Getting started with Interface Builder

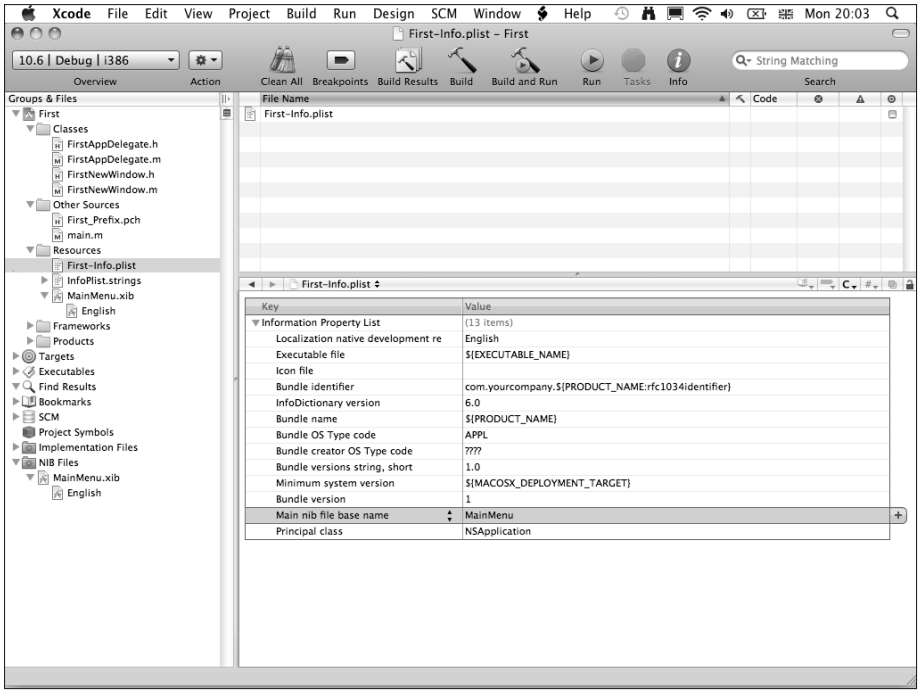
Setting classes and subclasses

Table 7.1 Key Features of the Nib System

Feature	*Explanation*
General object support	Nib files can include any Cocoa object or customized subclass.
Implicit object instantiation	Object instances loaded from a nib file are allocated and initialized in memory without `alloc/init` code.
Visual UI design	Objects with visible elements can be positioned with IB's graphic view editor.
UI object defaults	Visible objects can have a subset of their properties initialized to default values.
Object graphs	Objects can be grouped together in trees or hierarchies, which are accessed via a common root object.
Semiautomated loading	Nib files can be loaded automatically or loaded manually using code.
Lazy loading	Object groups can be loaded and released as needed to maximize memory.
Nib swapping	Nib swapping is a practical application of lazy loading. Nib files can be swapped as needed — for example, to load different views into a single window.
Outlets and actions	Objects in the nib file can be linked to objects in code via *IBOutlets*. Methods in code can be triggered by events generated by nib objects via *IBActions*.
Bindings	Objects can trigger code events automatically without explicit outlets or actions.

Figure 7.1

To locate the default nib file, open the Resources group, click `First-info.plist`, and scan down the list of items to find the Main nib file base name. You can edit this name to load a different nib file, but more typically you'll use the default nib when the application loads.



The classes shown in Table 7.2 can load a nib file on demand. `NSNib` supports two-stage loading, where the nib is loaded in one operation and expanded into a set of accessible objects in a second operation. This guarantees that all objects are available. For practical examples of on-demand nib loading, see Chapter 15.

Table 7.2 Cocoa Nib Loading Options

Option	*Explanation*
`info.plist`	The application `info.plist` file includes a default nib that is loaded automatically.
`NSNib`	Use the `initWithNibNamed:` method to load a named nib, and use either `instantiateNibWithOwner:` or `instantiateNibWithExternalNameTable:` to unpack the nib objects into memory. Use this deferred unpacking option to control when objects are ready to be accessed.
`NSBundle`	Use the `loadNibNamed:` method to create a new object from the nib contents.
`NSDocument`	Use the `windowNibName:` method to specify a nib that creates an empty document.
`NSWindowController`	Use `initWithWindowNibName:` to specify a nib that defines the window contents.
`NSViewController`	The iPhone equivalent of `NSWindowController` — use `initWithNibName:` to select and load a nib that contains a view.

Loading objects from nib files

The nib files generated by IB contain blank class instances, arranged in groups called *graphs*. Certain Cocoa objects can be nested in a folder-like way to create a tree or hierarchy. For example, a view controller object can contain a number of views, each of which may contain subviews. IB supports structures like these automatically and displays them visually. Not all objects can be graphed, and IB automatically limits nesting to objects that support it.

Comparing Xcode 3 and Xcode 4

The examples and illustrations in this book were prepared with Xcode 3. Preview versions of Xcode 4 were announced but not yet shipping just before the book went to press. Although the features and changes to Xcode 4 haven't been finalized, a key difference is that IB no longer runs as a separate application and is now built into Xcode. The linking process outlined below has also been enhanced and extended. You can now drag links directly to the code that features outlets and actions. If you drag a link to the code window from an object that doesn't have an assigned action method, Xcode/IB creates a stub for you automatically.

Generally, IB is much better integrated than it was in Xcode 3. You no longer need to be so careful about saving changes in IB when you modify the interface, or reloading classes when you make changes to the code. Otherwise, the steps used to design an interface and link it to code remain recognizably similar to those used in Xcode 3.x.

Cocoa's nib loader runs its own implicit `alloc/init` code as objects are loaded into memory. But by default, the objects aren't accessible from your code. To make them accessible, you must create links to them — a complex process that combines code directives with manual editing in IB, introduced in detail in the next chapter.

Editing nib files

Editing a nib is a six-step process. Typically you repeat the steps as the project evolves, adding more objects to a nib, or more nibs to the project.

1. **Define outlets and actions in your code.**
2. **Launch IB, using one of the methods listed later in this chapter.** IB scans your code and generates a list of outlets and actions. To avoid restarting IB after every code edit, you can also choose File ➪ Reload All Class Files file option.
3. **Add objects to the nib file.** If the objects are visible — buttons, windows, and so on — specify their size, position, color, and other default properties.
4. **If your code uses subclasses, reclass any modified objects to force IB to load the subclass.**
5. **Use IB's linking tools to connect objects and messages to outlets and actions.**
6. **Save the nib file.** This step is critical because Interface Builder doesn't save changes automatically before a build.

When you build and run the project, Xcode includes the modified nib file. If you follow the steps and create links successfully, your application can receive messages from objects in the nib file, send messages to them, and read or write their properties.

Getting Started with Interface Builder

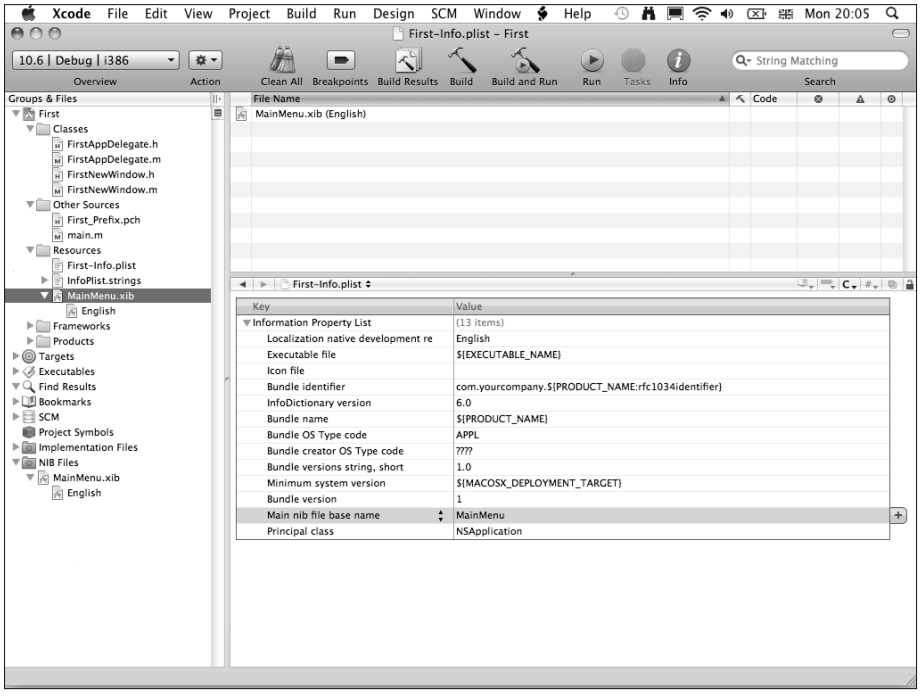
You can launch IB by double-clicking any nib file within Xcode. You can also run IB directly by double-clicking `Interface Builder.app` in the Applications directory created by Xcode, but double-clicking a nib file is more convenient.

With the First project open, click the Resources group in Xcode's Groups & Files pane to open it. `MainMenu.xib` is generated automatically when you create a new project, so you'll see it when you open the Resources group, as shown in Figure 7.2.

Double-click `MainMenu.xib` and wait. Interface Builder is a complex application, and it can take up to a minute to load. When it appears, choose Window ➪ Bring All To Front. You should see the windows shown in Figure 7.3.

Figure 7.2

The `MainMenu.xib` file is in the Resources group. The English item indicates the default English localization for the –; you can support other languages by creating nibs for them with different labels.



NOTE

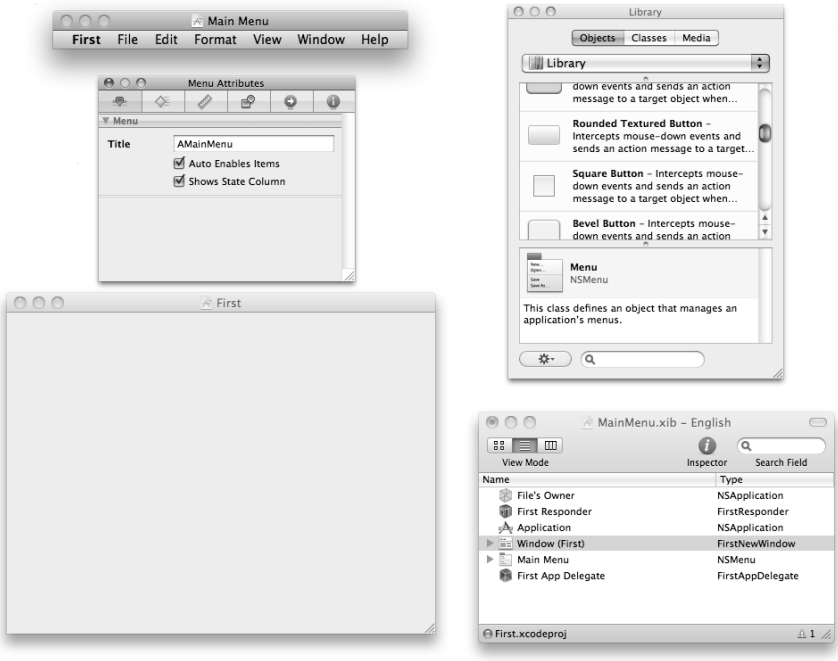
Although nib files have a `.xib` extension, they're not called xib files. The original extension was `.nib`, an acronym for *NeXT Interface Builder*. In Interface Builder 3.0, nib files were reimplemented using XML, and the extension was changed to `.xib`. The original name remained the same — possibly because no one is entirely sure how to pronounce xib.

Introducing IB's windows

IB has four window types. The *Document*, or *Doc*, window lists the objects in the nib file. The Doc window is associated with one or more *Edit* windows, which preview the visual design of the nib file. A nib often contains multiple elements that can be previewed in separate edit windows; for example, the standard application template includes a blank window and a separate menu element.

Figure 7.3

Interface Builder's windows, which are described next. The Main Menu window is independent of the others and simulates the application menu.



The *Library* window displays a library of Cocoa and other OS X items that can be added to the file. The *Inspector* window displays information about whichever item is currently selected in the Doc window. The Inspector and Library windows include tabs that can display subwindows.

TIP

For reasons that remain mysterious, windows in Interface Builder like to hide behind other windows. It's helpful to get into the habit of selecting the Bring All To Front menu option to force them into the foreground. You can open multiple projects in Interface Builder. When you select Bring All To Front, they all appear, so it's a good idea to work on a single project at a time.

Introducing the Document window

Selecting the Doc window shown in Figure 7.4 reveals the full list of objects included in the blank template. Objects are arranged in a finder-like way, with reveal triangles. Select the Window (First) object and click its reveal triangle. You'll see another object, called Content View, which appears indented under Window. Indentation indicates that an object is "inside" another object, within a tree structure with the top-most object as the root.

Figure 7.4

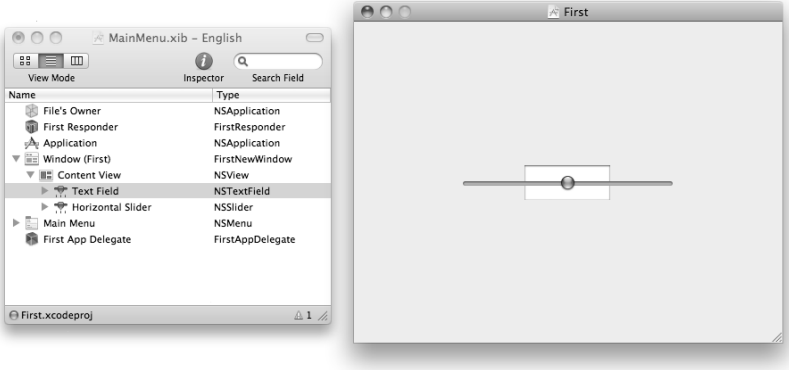
Expanding the items in the Doc window reveals how items are shown "inside" other items or placed on a lower branch of the object tree.

You can change the graph by dragging and dropping items. To remove items from the nib, highlight them and choose Edit ➪ Delete. You can also use the backspace-delete key. To add items to the nib, drag them across from the Objects list in the Library window.

The hierarchy is interpreted by different objects in different ways. To see an example, click the reveal triangle next to the Main Menu object. For a menu, the hierarchy defines menu headings, followed by lists of menu items. For a window, the hierarchy defines the objects that appear inside the window, and it also sets the order in which they're drawn. It's possible to hide objects behind other objects by changing their position in the graph. Objects with the same indentation are drawn top down, so objects lower down on the list cover objects above them, as shown in Figures 7.5 and 7.6.

Figure 7.5

Objects are drawn from the top of the Doc window down. Lower objects cover higher objects. When the slider is under the text field — the small white rectangle — it appears above it in the preview.

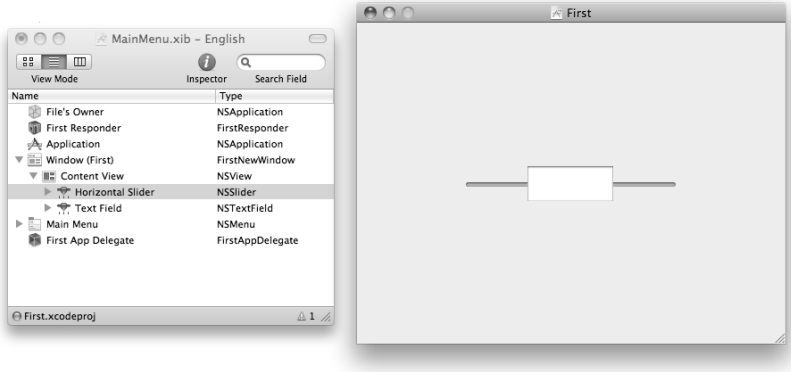


NOTE

When you drag an object from the Library to the Doc window, the release location is important: it defines the object's position in the graph. Only certain object relationships are permitted, and it's impossible to drop an object in a location that creates an invalid graph.

Figure 7.6

Swapping the items changes the draw order. When the text field is drawn last, it covers the slider.



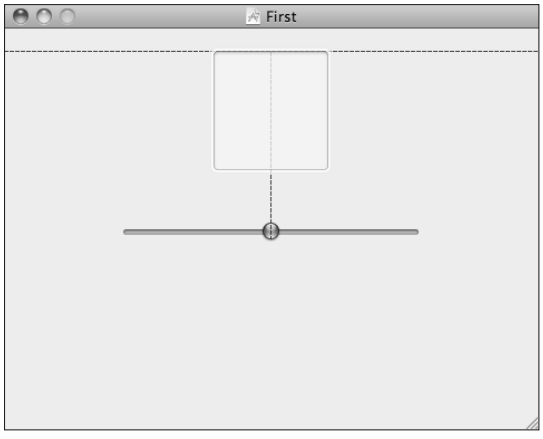
Introducing the Edit window

The Edit window, shown in Figure 7.7, is visual representation of the visible elements in the Doc window. Double-click the Window object to open an Edit window for it. The window in the blank template is empty, but you can add items to it by dragging them from the library window.

Use the Edit window for visual editing by dragging elements to set their sizes and positions. As you move items, guidelines appear to indicate useful visual relationships that help you align elements with each other. Note that you can use the Inspector window, described later, to set an object's size and position numerically.

Figure 7.7

The Edit window's guidelines appear automatically to indicate alignment. In this example, the text field is center aligned with the slider, and it is also aligned with the "safe" edge of the window, which defines the useful drawing area. On the iPhone — but not on the Mac — a guideline appears to indicate the window's center line.



CAUTION

Interface Builder displays independent nib elements in separate preview windows, so an application's menu always appears separately. It's considered an independent feature that isn't associated with a window, and it has a special preview window that enables you to test the menu design dynamically. However the preview doesn't support editing — you can only add, remove, and rearrange menu items in the Doc window.

It's important to understand that only some of the items in the Doc window can be edited in an Edit window. For example, if you double-click the `NSApplication` object in the Doc window, nothing happens. It only exists in code, and code is edited in Xcode, not in IB. Typically only menus, windows, and preference panes can be edited visually.

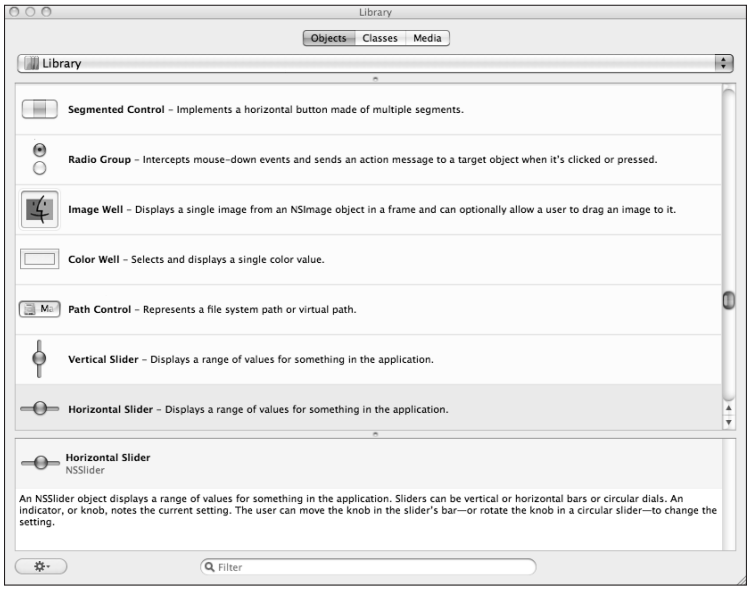
Introducing the Library window

The Library window is a list of resources that can be included in a nib file. It has three tabs: *Objects*, *Classes*, and *Media*. When you design a nib file, you'll use the Objects tab regularly and the Classes and Media tabs much less often.

The Objects tab, shown in Figure 7.8, displays a scrollable list of objects that you can add to a window to create a Cocoa interface. To add an object to a nib, drag it from this window to the Doc window and release it. The Doc window automatically enforces valid object trees. You can drop any object at the root level of a nib file, but only selected objects can be dropped inside other objects. For example, you can't drop an instance of `NSApplication` inside a window, because — not surprisingly — windows can't contain applications.

Figure 7.8

The Objects tab of the Library shows *visible* objects — Cocoa elements that can be used to build an interface. Not all Cocoa objects are visible.



You can also add objects with a visual element by dragging and dropping them onto an open Edit window. This immediately adds the object to the nib design so you can see it and work with its location and size. It also places it into a valid position in the graph.

TIP

You can create a very simple Hello World application by dragging a Label object — an instance of `NSTextField` — from the Library onto an empty window, double-clicking it to enable editing, and changing the text to "Hello, World." If you save the nib file and then build and run the application, the window appears showing the text. After an empty window, this is the simplest of all possible Cocoa applications.

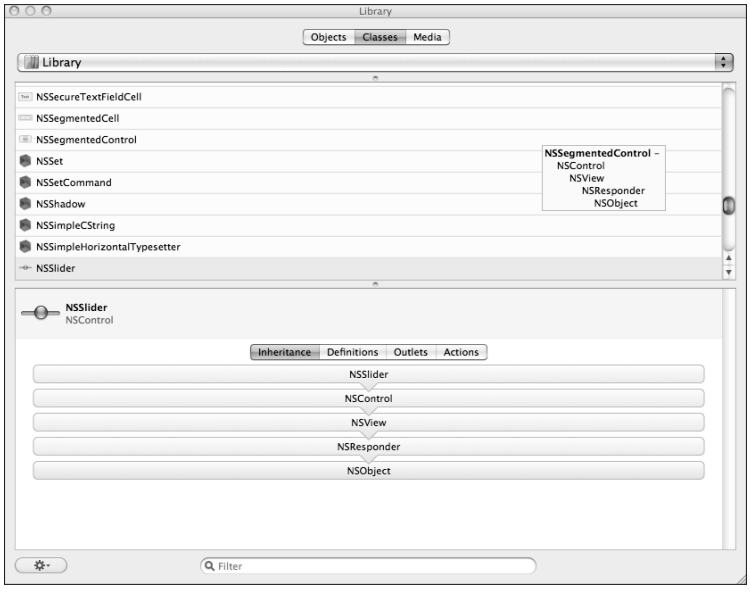
The Classes tab, shown in Figure 7.9, displays an alphabetical list of Cocoa classes. Selecting an object reveals information about it in four subtabs:

- **Inheritance** displays the object's superclasses.
- **Definitions** displays the framework that defines the object. Some objects are defined in *plug-ins*. Ignore the plug-in feature for now; it's not used in basic nib design.
- **Outlets** displays object properties that have been prelisted in IB as a convenience. You can create your own custom outlets for any objects, but a handful of objects feature predefined outlets. For example, `NSDrawer` includes outlets to a parent window, a content view, and a delegate.
- Similarly, **Actions** displays a list of predefined actions. For example, `NSDocument` includes links to predefined print, save, save as, and other methods.

Most objects don't include predefined outlets and actions, so these elements remain blank. The contents of this tab are occasionally useful as a memory jogger, but it's possible to design a nib file without using this tab at all.

Figure 7.9

The Classes tab of the Library window shows *all* Cocoa objects — visible and invisible. You can use this tab to add data objects such as `NSSet` to a nib. This tab also displays class inheritance relationships and predefined connections.



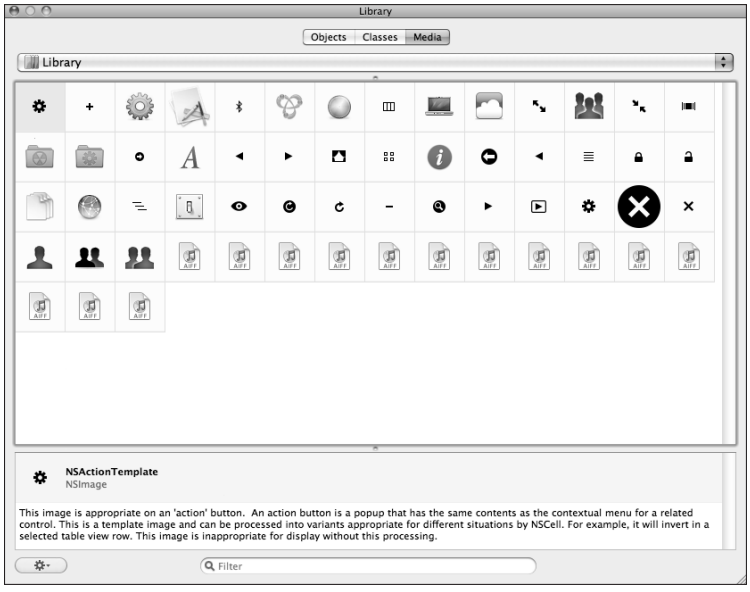
The Media tab, shown in Figure 7.10, displays predefined graphics and other media features. Most of the graphics are designed to be used within an instance of `NSCell` — Cocoa's button and table cell manager class. Use this tab to add standard Mac graphics to your application.

NOTE

You can design your own graphics and import them manually using other Xcode features, but they won't appear in the Media tab. For details, see Chapter 16.

Figure 7.10

The Media tab displays standard Mac graphics that you can use in your applications to create a familiar look and feel. You can also load graphics independently and assign them in code, but this tab is a quick way to create a UI with standard Aqua features.



Introducing the Inspector window

The Inspector window is the most critical window in IB. It has multiple functions, and is used for the following:

- Presetting object properties and behaviors
- Defining object special effects, including animations
- Setting object position and size and defining object behaviors when a window is resized
- Creating *bindings*, implicit links to code that are triggered when the object is modified
- Creating explicit links to object and outlets in code
- Assigning an object to a subclass
- Setting an object tooltip

When you select an object in the Doc window, its corresponding properties appear in the Inspector window, selected by the icons in the tab bar at the top of the window. The icons are slightly cryptic. Here's a list of their formal names, from left to right.

The *Attributes* tab, shown in Figure 7.11, is a context-dependent property editor. It's equivalent to setting object properties in code, but is easier to work with because it displays many properties simultaneously. When the selected object is visible in an Edit window, changing its properties in the Inspector immediately updates its appearance. Each object has a different selection of editable properties: some objects have no editable properties at all; others display tens of options. Properties define how the object appears and how it behaves. For example, a slider object can appear with or without check marks, and the slider's movement can be free or constrained to the check marks; a text field can be center-, left-, or right-justified, with various possible line break options; and so on.

Figure 7.11

The Attributes tab is one of the most critical elements of IB. Use this tab to preset object properties. Every object displays a different selection of properties; for example, you can set the title and define the controls that are visible for the application's window.

The *Effects* tab, shown in Figure 7.12, selects special effects, including drop shadow and transparency effects, and various animation effects implemented with both predefined and editable *filters*. For example, you can specify Photoshop-like compositing modes to combine the object with objects that are drawn beneath it. You can also set animated transitions — such as page curls and dissolves — that are triggered in response to user actions.

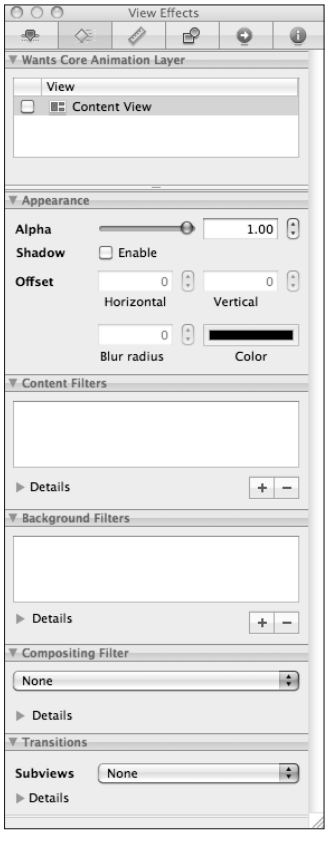
CAUTION

IB's animated effects can be unreliable. It's better to create animations using the techniques introduced in Chapter 17.

Effects are optional and usually cosmetic, so you can ignore these options, as many developers do. But this tab provides creative possibilities that you can use to create unique, striking, and distinctive effects. Effects are supported by almost all visible objects.

Figure 7.12

The Effects tab defines optional animations and special effects. Most of these features are specialized, and many are only used rarely, although the drop-shadow effect shown here is a simple but useful effect that adds depth to a window or object. This tab is not available on the iPhone.



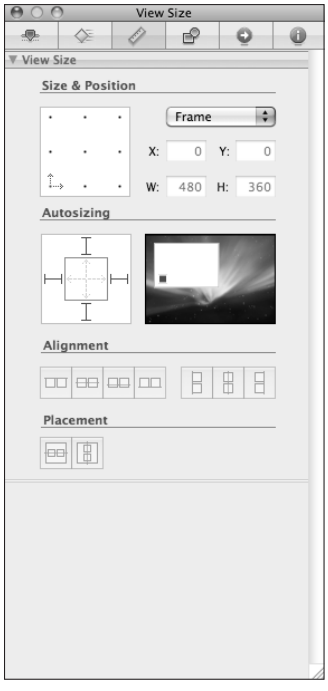
The *Size* tab, shown in Figure 7.13, defines object size and alignment. It not only sets the object dimensions, it also defines how the object is anchored to the window around it, and it includes alignment and placement buttons that can center an object horizontally or vertically or align it with other elements.

TIP

You can select multiple objects in either the Editor or the Doc window, and use the placement buttons to align and center them as a group. For an example of working with the Inspector, see the next chapter.

Figure 7.13

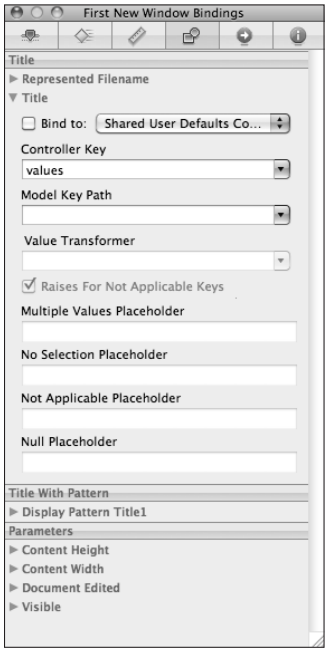
Use the Size tab to set the size, position, and alignment of objects with fine numerical control. The Placement and Alignment buttons auto-align and auto-center selected items.



The *Bindings* tab, shown in Figure 7.14, links a selection of object properties to various data processing features in the application. Bindings are an advanced Cocoa feature that simplifies data management. For example, you can use bindings to manage application preferences without having to create outlets and actions for every possible preference. For more information, see Chapter 13.

Figure 7.14

The Bindings tab links items to data structures in the application, bypassing standard accessors and allowing for automated refreshes and updates. Bindings are not available on the iPhone.

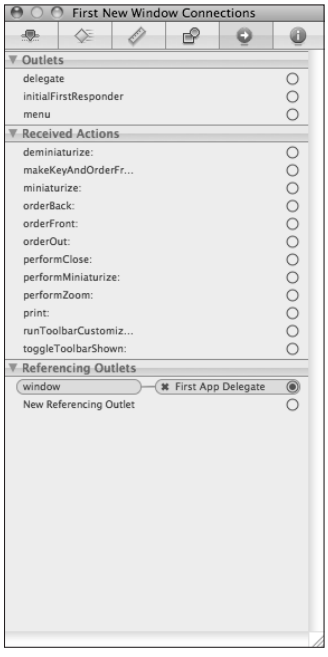


The *Connections* tab, shown in Figure 7.15, is used to link an object to outlets and actions defined in code. When you select the tab, the Inspector window displays a list of available outlets and actions. The linking process is detailed in the next chapter. After linking, you can click this tab to review the connections between an object and its supporting code.

The *Identity* tab, shown in Figure 7.16, defines an object's class, sets an optional tool tip, and defines optional runtime attributes. You can also highlight objects with a colored tag for convenience. This is a cosmetic feature that doesn't affect the object's appearance in the application, but makes it easier to view related objects in IB. The Class Identity field at the top of the Identity tab is the most useful element. Use it to set the class of an object using a drop-down list that displays a list of available classes. This list is automatically preset to show compatible classes; for example, you're not allowed to reclass an application object as a slider object, but you can change a text field into a different type of text field with slightly different features.

Figure 7.15

The Connections tab lists an object's outlets and actions. Received actions are events received by the object from other objects or processed internally, while sent actions (not shown here) are sent from the object to other objects. Outlets are links to predefined properties, while Referencing Outlets are links to custom properties in the code.

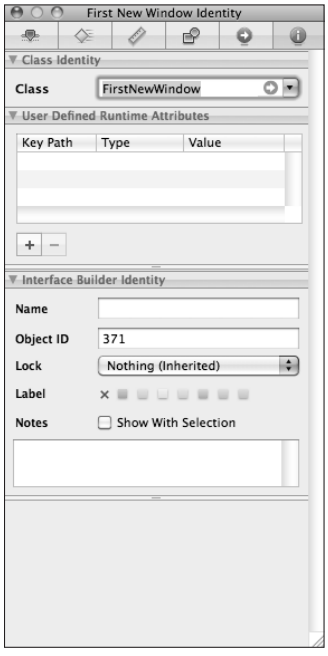


Introducing First Responder and File's Owner

The Doc window includes two items called *File's Owner* and *First Responder*. Unlike the other items in a nib file, they aren't true objects — they're placeholders, and they are used as link destinations. As Figure 7.17 illustrates, they're a halfway house between messages and outlets generated within the nib and the corresponding outlets and actions defined in code. File's Owner is a placeholder for the object that loaded the nib and stands in for the object that "owns" the nib. First Responder is a placeholder for the Cocoa responder chain.

Figure 7.16

Use the Identity tab to select an object's class from a list of nominally compatible classes – including custom subclasses. You can also define tool tip text here.

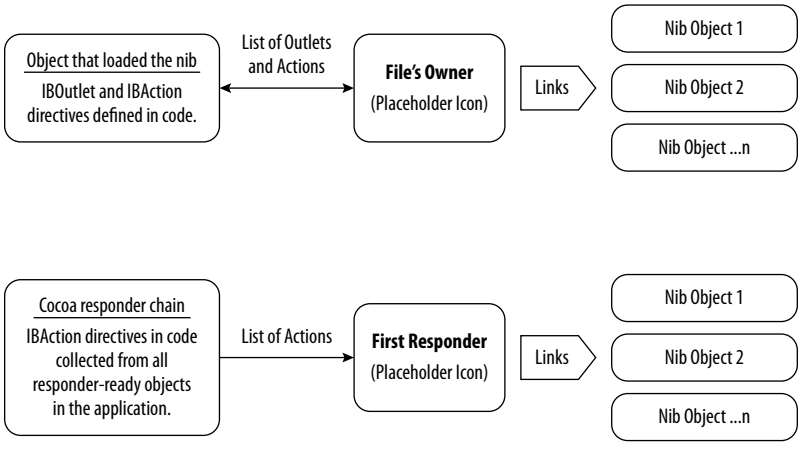


NOTE

The blank application template includes an instance of `Font Manager`. If your application doesn't need to support multiple fonts, you can delete this object. It's essential for full-featured text editing, but is redundant in simple applications. The template also includes an instance of `NSApplication`, which is the main application object. This object must appear in the main nib file. It isn't needed in any other nibs.

Figure 7.17

File's Owner is a virtual placeholder object, standing in for the object that loaded the nib. It's included as a link source and destination — it's not practical to link from IB to the corresponding code in Xcode, so File's Owner is used as a halfway house. First Responder performs an equivalent function for the Cocoa Responder Chain.



Setting Classes and Subclasses

Now that we've had a tour of Interface Builder, we can return to the problem at the end of the previous chapter where we successfully subclassed `NSWindow`, but discovered it wasn't responding to messages.

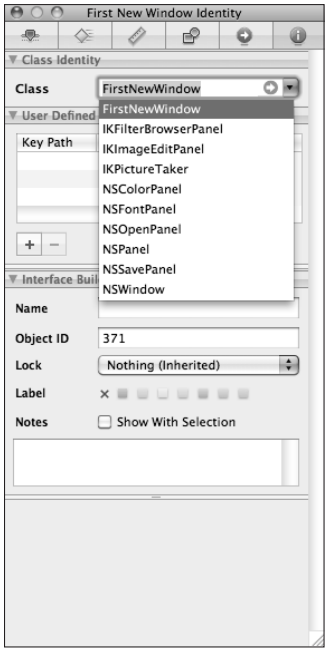
Under the menu tree is an object called Window (First). Window (First) is an instance of `NSWindow` — this is why the subclass of `NSWindow` we created in the previous chapter does nothing. The nib file is still loading an instance of `NSWindow`, and our new subclass is being ignored.

Interface Builder includes an option for changing the class of an object. Click on it in the Doc Window to highlight it, and click the right-most tab in the Inspector window — it shows the letter "i" inside a circle. The top-most pane is called the Class Identity pane. It includes a drop-down menu with a list of all the classes used in the project. Click on the drop-down menu to see a list of compatible classes, and select FirstNewWindow, as shown in Figure 7.18. This reclasses the window as an instance of our new subclass, adding the newly defined features to the object.

When you create a subclass in code, you *must* subclass the corresponding object in any nib file that references it. If you don't, the application loads the original unmodified class, and your code is ignored. I illustrated this feature early on because it can be one of the more puzzling and confusing features of Cocoa application design.

Figure 7.18

To replace an object with a custom subclass, select it in IB, click the Identity tab, and select the subclass name from the list. The object now supports all the properties and methods of your subclass.

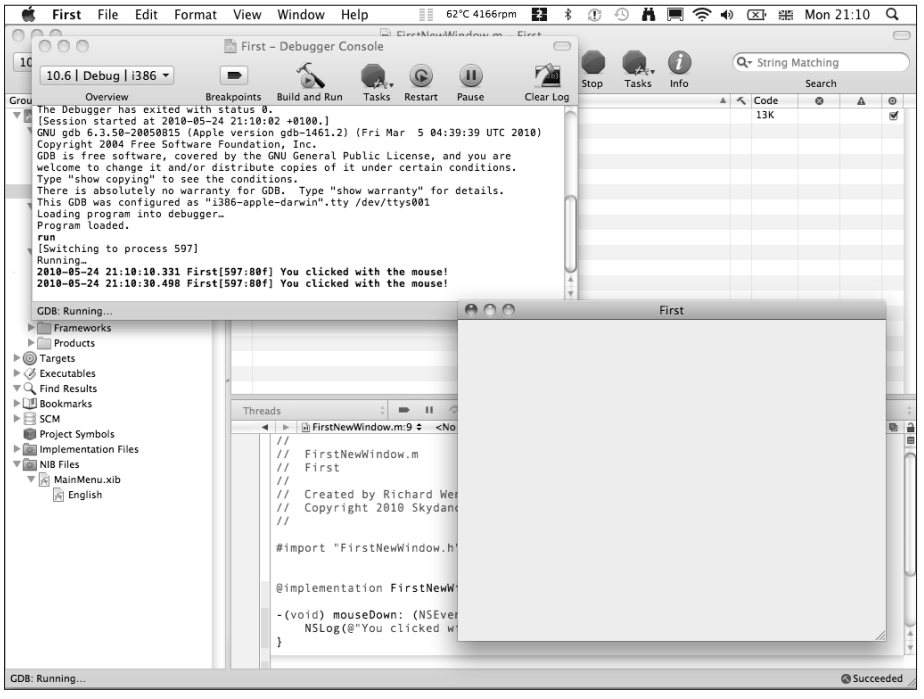


Save the file, and Build and Run the project. You should get the result shown in Figure 7.19. The default window has been replaced with the subclass, which now responds to mouse events. Success!

Now that you have created a very simple project, you can move on to a more complex application.

Figure 7.19

After reclassing the application window as your new `FirstNewWindow` class, the window responds to mouse click events and runs your click handler.



Summary

In this chapter, you were introduced to nib files and looked briefly at the Cocoa classes you can use to load them into your application. You were given a detailed tour of the different windows in Interface Builder, with their subtabs. Finally, you learned how to use the Identity tab in the Inspector window to subclass a nib object to force it to run custom code.

8 Building an Application with Interface Builder

Now that you have had an introduction to Interface Builder (IB), you are ready to use it to build a simple application. This application will use a simple UI created in IB that links nib objects with code. It runs a simple timer, controlled by two buttons. The button text changes according to the application state, and the count appears in a large font.

Designing a Project in Interface Builder

First, in Xcode, choose File ➪ New Project, click the Application item under the Mac OS X section, select the Cocoa Application template, and click the Choose button. Save the project as Counter. Open the Resources group, and double-click `MainMenu.xib` to launch IB and load the nib file for editing. After IB loads, choose Window ➪ Bring All To Front.

If the Library window isn't visible, choose Tools ➪ Library. If the Edit window isn't visible, double-click Window (Counter) in the Doc window. You should see a collection of windows similar to the one shown in Figure 8.1.

The position, size, and layout of the windows aren't standardized in IB, so the layout you see may vary. You'll be working with the Doc, Inspector, Edit, and Library windows. You can ignore the menu. If you have a larger monitor, you should be able to view all of these windows simultaneously. On a smaller display, you may need to minimize certain windows to allow space for the others.

TIP

The sample project in this chapter uses the objects in the standard IB library. You can extend the library with third-party objects. For example, the BWToolkit at `http://brandonwalkin.com/bwtoolkit/` is a useful free collection of IB objects.

8 In This Chapter

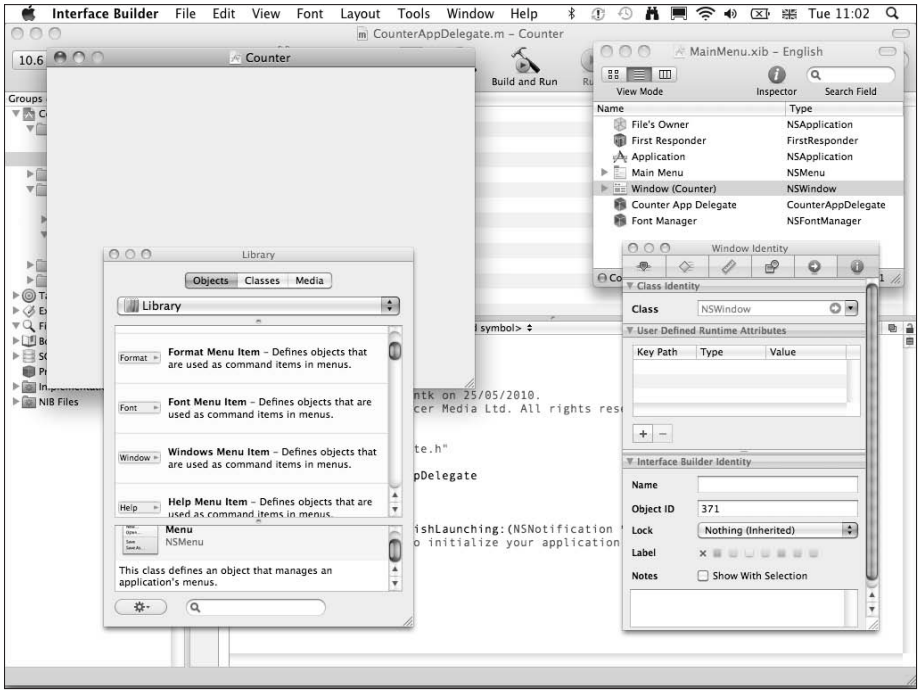
Designing a project in Interface Builder

Understanding links, outlets, and actions

Using advanced UI techniques

Figure 8.1

Creating a working layout ready for editing. On a dual monitor system, you can keep the code visible in one window and IB's windows visible in another. A single monitor system is less convenient.



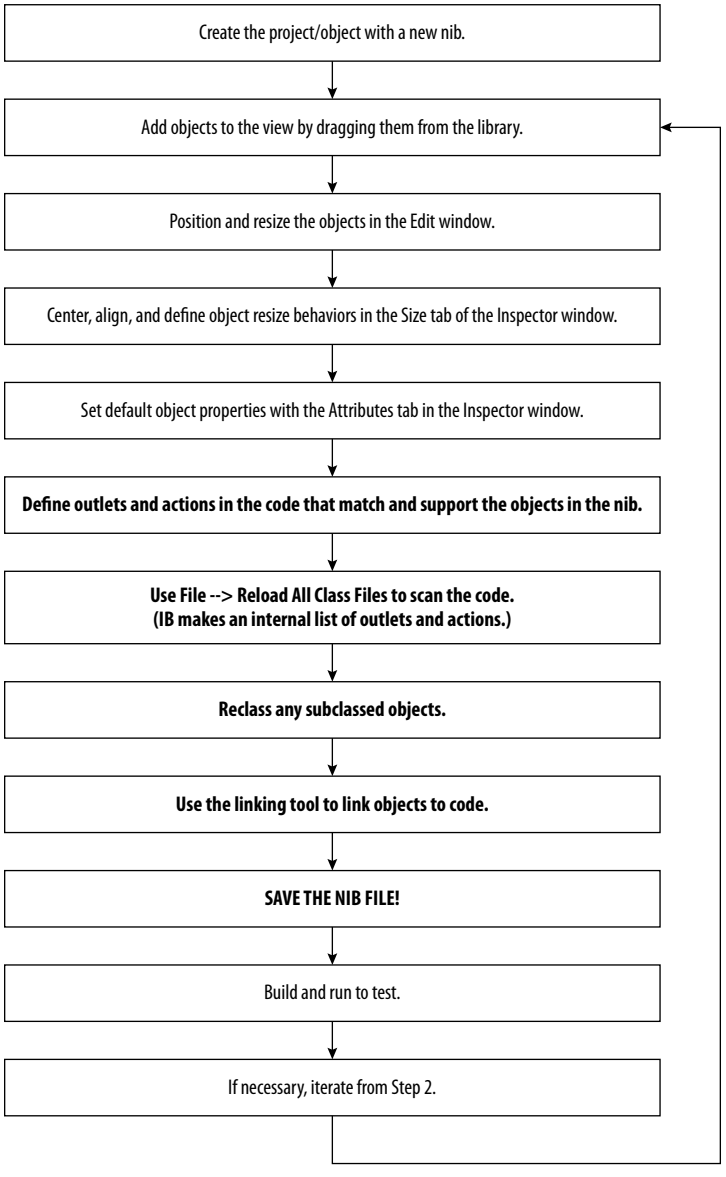
Introducing the Interface Builder workflow

As outlined in the previous chapter, Interface Builder and Xcode do not create automatic links between objects in a nib file and objects in code. You must add links manually, and until you do, the nib objects and the code remain disconnected. If you load a nib without creating links, it appears in memory, but your code can't access its features. When the user interacts with it, it doesn't send messages to your application.

Linking is a complex multistage process. It's error prone, and many basic build errors are created by missing links and other nib-related problems. These errors are usually trivial and easy to repair, but the most efficient way to avoid them is by following an explicit workflow, shown in Figure 8.2. The items in bold are critical steps. You must work through these steps for every object in the nib that is accessed from your code.

Figure 8.2

A typical IB workflow. For a simple project, you can start with code definitions and then add matching objects to the nib. In most projects, you'll iterate through the steps as shown.



NOTE

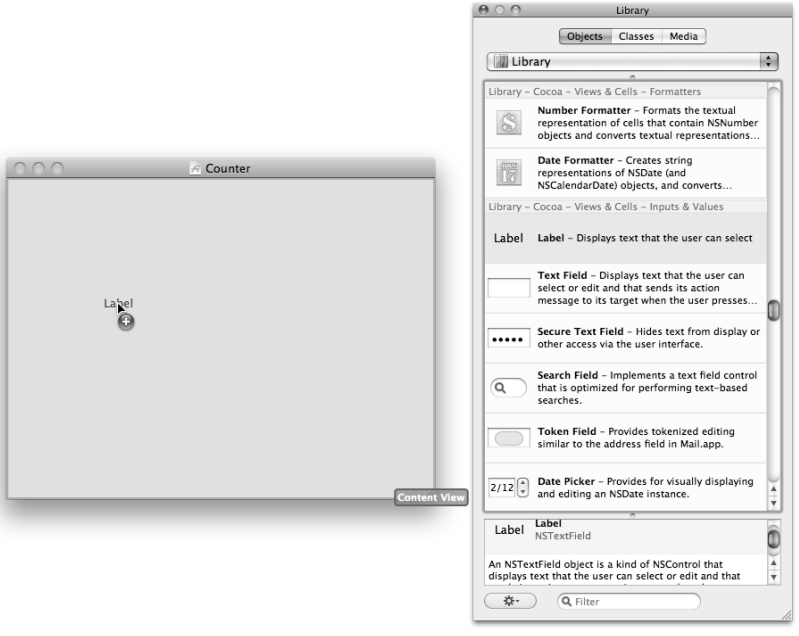
You can add "dumb" objects to a nib for decoration. Dumb objects don't need links: they appear in a view when the nib loads, but they don't support user interaction, and their appearance never changes. Examples include decorative or indicative graphics, such as background wallpaper in a view.

Adding objects to a nib

You'll begin by adding a text field to the nib. If it isn't already selected, select the Objects tab in the Library window. Scroll through the window until you find the Label object — it's slightly more than halfway down the list. Drag it from the Library window and drop it on the Edit window, as shown in Figure 8.3. When you release the mouse button, the label is added to the view.

Figure 8.3

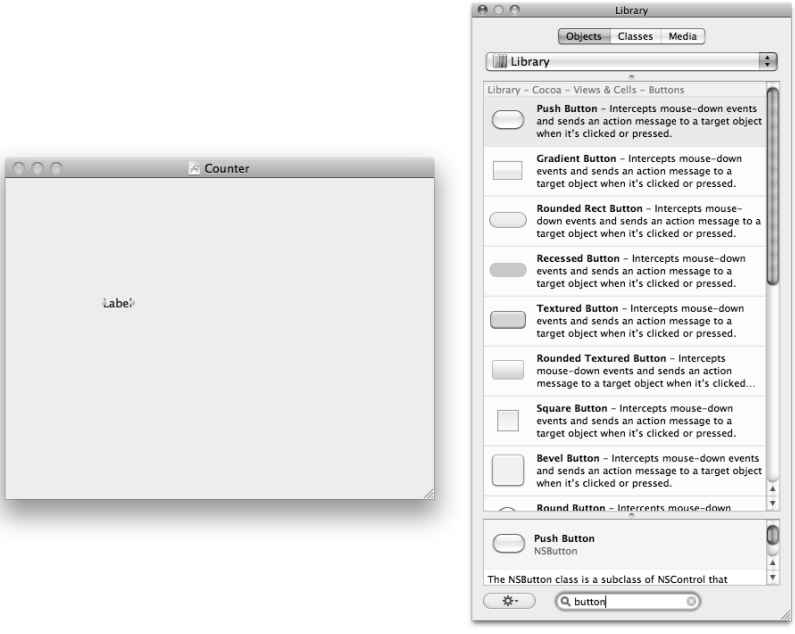
Adding an object to a view in the Edit window. You can also drop objects into the Doc window, but adding them to the Edit window lets you define the object's position.



The full list of objects in the Library window is long, so it can be difficult to find items. The window includes a search feature that can preselect objects for you by name. At the bottom of the window, type **button** into the search box, as shown in Figure 8.4.

Figure 8.4

Using the Library search to preselect items by name



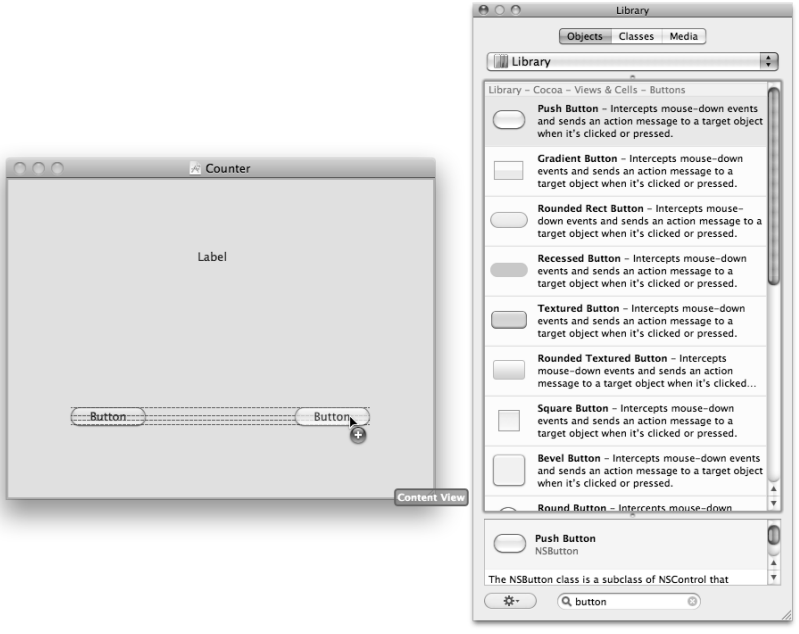
TIP

The search is text-based, so you can use any named feature; for example, you might search for "textured" instead of "button."

Drag and release a Push Button from the Library into the view, and then drag and release another Push Button. Move it around the view without releasing it. You'll see various guidelines appear as it aligns with the label and the other button; for example, when the two buttons are horizontally aligned, you'll see the lines shown in Figure 8.5.

Figure 8.5

Adding or moving any object in the view displays guidelines that make it easier to align objects horizontally, vertically, or along a mutual center line.



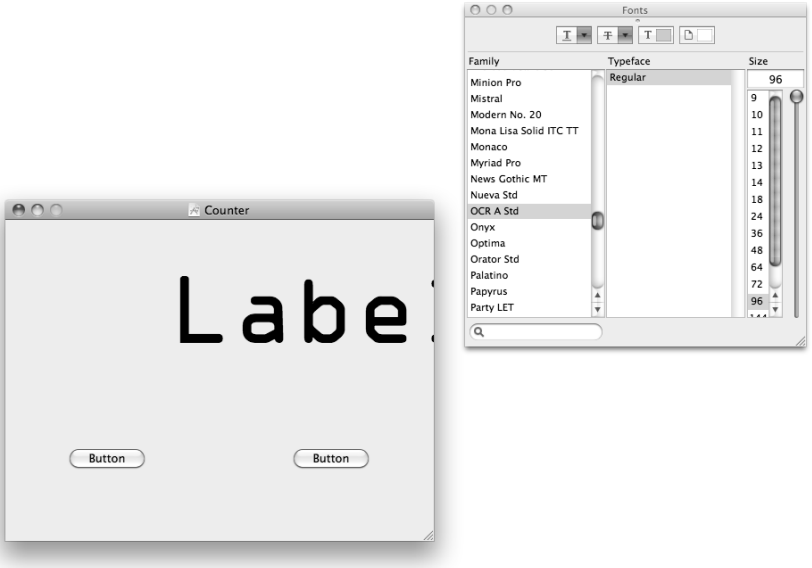
Setting fonts and font sizes

Click the label to select it; two marker dots appear at each side. Choose Fonts ➪ Show Fonts to display a font selector and text size dialog. The label should be big and readable, so you'll set the size to 96 points. At the right-hand side of the font selector, select 96 from the list.

You'll also change the font to OCR A Std to create a digital look. The choice of font doesn't affect the application, so you can select a different font here if you prefer an alternative styling. The result is shown in Figure 8.6. For completeness, you can also change the font used in the buttons. Click each button in turn and select a new font from the list.

Figure 8.6

Using the Fonts dialog to select the font and size for the text label. When you change the font size, the label resizes itself automatically.



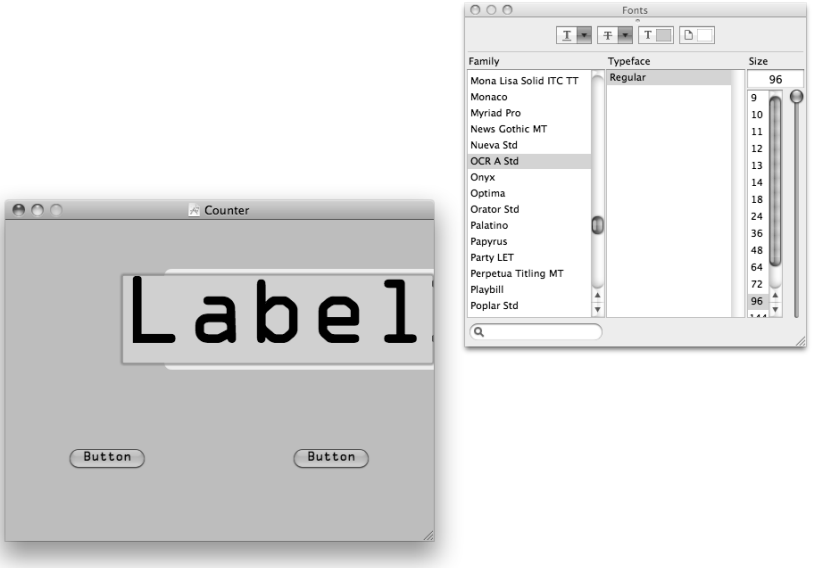
Aligning objects and setting attributes

Because you don't want the word "label" to appear, you'll begin by changing the label text. Double-click the label. The view changes to the one shown in Figure 8.7

Because this is a counter, the default count is 0, so type "0" and press Return to replace the label text.

Figure 8.7

Double-clicking an item with text enables text editing. You can type in any replacement text. Press Return to confirm an edit, or press Escape to cancel it.



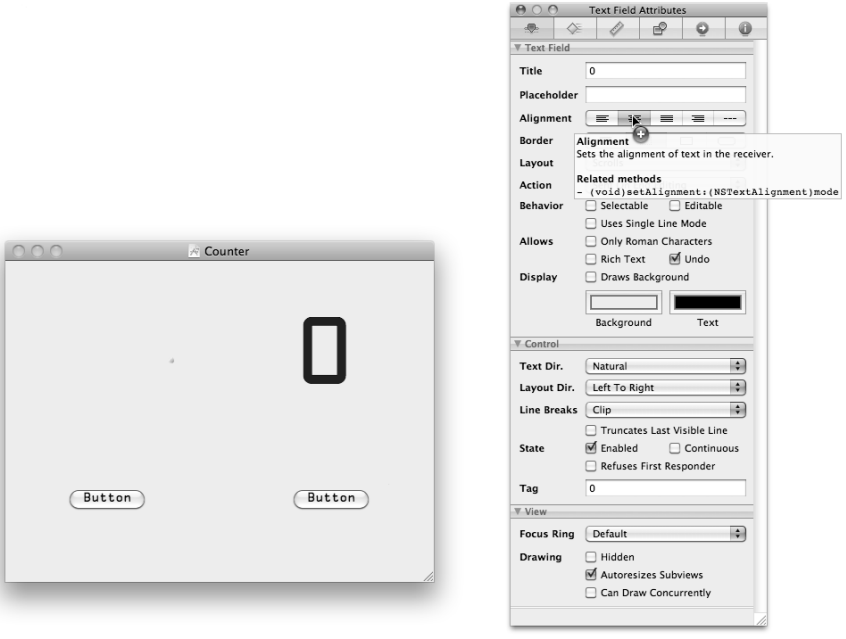
You'll notice that the text is left justified. It's often useful to center-justify text labels, so you'll modify the text field accordingly. If the Inspector window isn't visible, choose Tools ➪ Inspector to display it. Select the leftmost Attributes tab. If the text field isn't already selected, click it. You'll see a list of properties — called *attributes* — that can be set for the text field. You can set most of these properties in your start-up code, but it's often more convenient to preset them in IB. This generates less code, makes the code easier to maintain, and also gives you instant visual feedback as you design a view. To select center-justification, click the center-alignment tab, shown in Figure 8.8.

TIP

If you hover the mouse over an attribute, a tool tip appears with a reminder of the code used to access the equivalent property.

Figure 8.8

Setting default text field attributes. Most of the attributes are self-explanatory. Optionally, you can set the font color here. Clicking the black area in the Text box, which appears halfway down, displays a color picker.

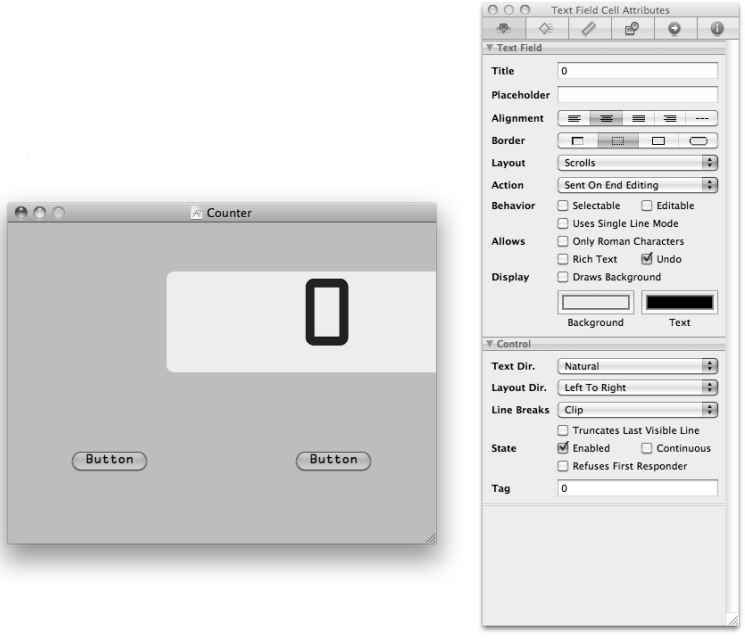


Selecting cell objects

On the Mac, many UI items are split into a container object and an underlying cell object. This isn't true on the iPhone, where UI items tend to be self-contained single objects. If you click an item twice in the Edit window — slowly, so that you don't generate a double-click — you can select the underlying cell object, as shown in Figure 8.9. Note that the title of the Inspector window changes to Text Field Cell and a highlight appears around the cell object. The cell object defines the object's contents and most of its appearance. Any edits you make to the cell are automatically applied to the object as a whole.

Figure 8.9

Clicking an object twice slowly, rather than double-clicking, selects an object's cell, if it has one. Most cell objects share most of their attributes with their container object. Editing one object automatically changes the properties of the other.



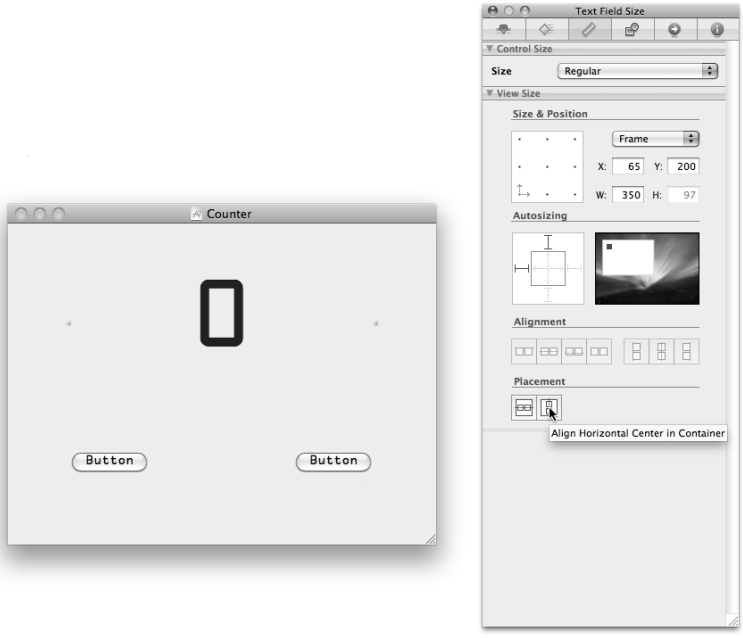
Centering and aligning objects

The label isn't centered in the view. You can align objects by eye, but IB includes a centering tool for precise alignments. Select the Size tab at the top of the Inspector window (the third tab from the left) and click the Align Horizontal Center button, shown in Figure 8.10. The label is centered automatically. The adjacent placement button centers objects vertically. You can use the numbers at the top left of this window to set the size and position of objects numerically. You can drag a rubber-band box around groups of objects in the Edit window to align them together.

The Autosizing features control how objects respond when the surrounding window is resized. The outer lines in the square anchor objects to the surrounding frame; for example, you can force an object to keep a fixed distance from the edges of a window. The inner lines control whether or not the object resizes itself with respect to the anchor points.

Figure 8.10

Using the Placement buttons in the Size panel to center an object in the window. The Alignment button row above the Placement buttons is active when multiple objects are selected.



TIP

You won't use resizing in this example, but you'll find it useful to experiment with this feature. It's particularly important for iPhone applications that implement autorotation in code; turning off the anchors and autosizing options forces objects in a view to rotate and reposition themselves automatically.

You're nearly done editing the view. Edit the left button text so it says Start and the right button text so it says Stop. Save the file. You're now ready to start linking the objects in the nib to new supporting code.

TIP

You can build and run the application at this point. When it loads, you'll see that the window contains the new objects. You can click the buttons. They darken momentarily in the usual Aqua way, but because they're not connected to the code, nothing happens.

Understanding links, outlets, and actions

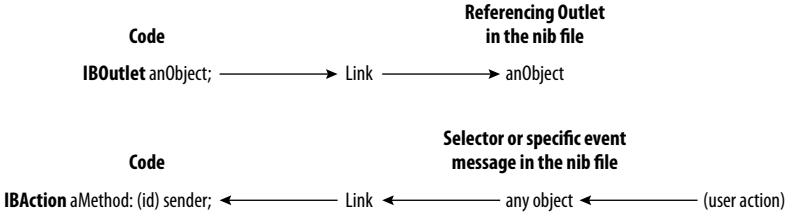
When you link nib objects to code, every active object in the nib must have a counterpart in code. In this example nib, there are two buttons and a text field, so you must declare two button objects and a text field object in the code. As mentioned earlier, "dumb" non-interactive objects don't require code, but active objects *must* have a code counterpart, and you must link the counterpart to its associated object in IB.

Links have two ends and two types. *Outlets* are used to read and set object properties. *Actions* are used to respond to messages. At the code end, outlets are defined using the `IBOutlet` directive. Actions are defined with `IBAction`, but this isn't always obligatory. At the nib file end, links are defined using a visual linking tool.

Figure 8.11 illustrates how outlets and actions are used. Outlets define a unique link between an object referenced in code and an object in the nib file. Outlets have exactly one object at each end of the link. Actions have a single destination — an event handling method — but can be triggered from multiple sources.

Figure 8.11

Typical design patterns for outlets and actions. This figure shows the simplest and most common design pattern. More complex patterns are also supported; for example, received actions are processed inside an object, rather than being sent to another object.



TIP

Technically, `IBOutlet` tells the compiler to leave a pointer undefined at compile time. Cocoa's nib loader returns the value at runtime when the nib loads. `IBAction` is a method placeholder. Objective-C calculates all method addresses at runtime, so the directive is largely for clarity.

Using IBOutlet and IBAction

To create an outlet, insert the `IBOutlet` directive into an object's property declaration. Replace

```
@property (assign) AClass *anInstance;
```

with

```
@property (assign) IBOutlet AClass *anInstance;
```

That's all that's required. The rest of the process is completed in IB.

TIP

Although it's trivially easy to create an outlet, it can be harder to remember that you need to do it. If you follow the checklist in Figure 8.2, you won't forget.

`IBAction` is more complex. In certain circumstances, the `IBAction` directive is optional, and a conventional `void` method is an acceptable substitute. This feature is implemented differently on the iPhone and on OS X. On the Mac, Interface Builder recognizes a method as an `IBAction` as long as it has the following signature:

```
-(void) methodName: (id) sender;
```

You don't need to replace `void` with `IBAction`, but actions *must* have this signature to be recognized. No variation is allowed. The `sender` parameter holds a pointer to the object that triggered the message. Applications are described next.

This auto-recognition feature is implemented on the iPhone. You must include the IBAction directive. However, you can specify actions that don't take a parameter. For example:

```
- (IBAction) methodName;
```

is valid on the iPhone only. The `sender` parameter is discarded.

NOTE

More information about `sender` appears later in this chapter.

Defining outlets and actions in the interface

To keep the application as simple as possible, you'll implement the counter within the application delegate. In a typical application, the delegate would be reserved for system messages, and the counter features would be implemented in a subclassed window or a custom view. In practice, this means subclassing at least one extra object, and doesn't add anything to the functionality or efficiency of the application. For this example only, you'll keep the supporting code in the delegate.

Click CounterAppDelegate.h. An outlet to the `window` object is already defined, so you can see an example of the correct syntax. Add declarations for an `NSTextField`, and two `NSButtonCell` objects, as follows:

```
#import <Cocoa/Cocoa.h>
@interface TimerAppDelegate : NSObject <NSApplicationDelegate> {
 NSWindow *window;
 NSButtonCell *startButton;
 NSButtonCell *stopButton;
 NSTextField *countText;
}
```

```
@property (assign) IBOutlet NSWindow *window;
@property (assign) IBOutlet NSButtonCell *startButton;
@property (assign) IBOutlet NSButtonCell *stopButton;
@property (assign) IBOutlet NSTextField *countText;
@end
```

NOTE

You're creating outlets to button cell objects and not to button objects because cell objects are the active and editable element in many UI objects, including buttons. You can only set the text of a button by modifying the cell object inside it. However, this rule of thumb isn't completely consistent. The text field includes a cell, but it's easier to ignore it and set the text with a direct outlet to the field itself.

How do you know the names of the objects in the nib? The names are listed in the right-hand column in the Doc window, as shown in Figure 8.12. Click the reveal triangle for Window (Counter) and for all of its subobjects. The Type column lists the object names. When you add objects to a nib, you can use this feature to read their names, if you don't know them already.

NOTE

While you must use standard Cocoa object names, you have a free choice of pointer names. This example uses self-explanatory names for clarity, but these names are arbitrary.

Figure 8.12

To find object names, look in the Type column. You must use the same object names when you create outlets in your code.

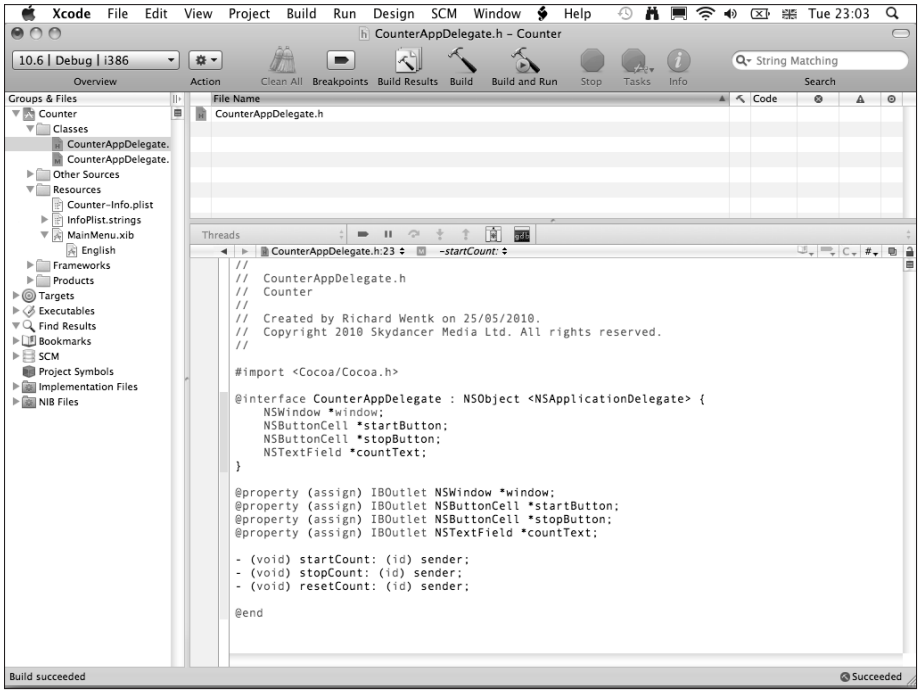
You'll also define three action methods — one that will be triggered from a button to start the count, one to stop the count, and one to reset the count. Add these three signatures after the property declarations, but before the `@end` directive:

```
- (void) startCount: (id) sender;
- (void) stopCount: (id) sender;
- (void) resetCount: (id) sender;
```

This completes the interface, which is shown in Figure 8.13. Choose File ➪ Save to save the file.

Figure 8.13

The completed interface for the timer project. Outlets and actions are defined, but not yet implemented.



NOTE

Although there are three methods, there are only two buttons. In this example, I'll demonstrate how to switch methods dynamically, reprogramming a button to trigger two different methods depending on the state of the timer. It's just as practical to create a separate reset button, but the dynamic option is more powerful.

Defining outlets and actions in the implementation

To complete the first stage of the code, you'll add the new objects to the implementation and create method stubs for the new methods. At this stage in the development workflow you have two choices: You can create a stub-filled prototype implementation, add links to it in Interface Builder, and then return to the code to complete the implementation; or you can write a first draft of the code, add links in Interface Builder, and run a test build.

Both approaches are valid, but the first is simpler and easier to manage because you can add code features in stages, implementing and testing each method and each event separately. The second approach is better suited to implementing handler methods with working code copied from an existing application. You'll use the first approach here.

Select CounterAppDelegate.m in Xcode. Start by adding the new pointers to the `@synthesize` directive in the file. Next, add a stub for each method. The final implementation should look like this:

```
#import "CounterAppDelegate.h"
@implementation CounterAppDelegate
@synthesize window, startButton, stopButton, countText;
- (void)applicationDidFinishLaunching:(NSNotification *)
   aNotification {
 // Insert code here to initialize your application
}
- (void) startCount: (id) sender{
}
- (void) stopCount: (id) sender{
}
- (void) resetCount: (id) sender{
}
@end
```

Save the file. Now you're ready to link the code to the objects in Interface Builder. Following is a recap of what you've done so far:

1. Added an object for each "live" object in the nib
2. Created pointers for those objects
3. Added an `IBOutlet` directive for each object
4. Defined some action methods in the interface
5. Created a stub implementation for each method

This checklist is the bare minimum needed to create code-side objects ready for linking. As you can see, this isn't an elegant or simple process — and there are more stages to come when you finish the nib-side links. Unfortunately, there is no other way to work with nib files. You must work through these steps carefully whenever you design a nib and create supporting code for it. With practice, the process becomes easier.

Creating links in Interface Builder

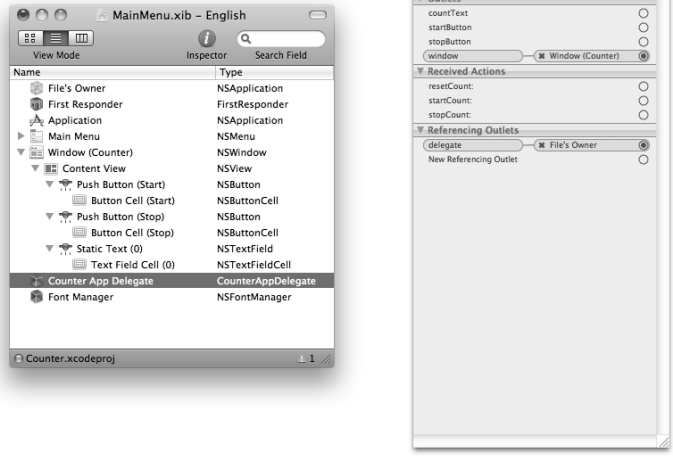
The nib-side linking process begins when IB loads the current class headers. Behind the scenes, it scans them and creates a list of possible link destinations. The destinations are listed in various pop-up menus in IB. To create a link, you drag a line from a link destination to a target object. When linking actions, you select the action from a separate floating pop-up menu that appears over the target object.

Take a look at how this works in practice. Begin by choosing File ➪ Reload All Class Files in IB. This runs the scan and creates the link list. This process is invisible — IB provides no feedback while it's happening, although for larger projects, the spinning beach ball icon may appear while you wait for the process to complete. The scan is also run automatically whenever IB loads, and when you double-click a nib file in Xcode to open it for editing.

In the Doc window, select the Counter App Delegate object. In the Inspector window, select the Connections tab (the one with the rightward-pointing arrow, second from the right). Figure 8.14 shows the result. The link destinations in the Application Delegate are listed in the window.

Figure 8.14

A list of link destinations. The outlets and actions are listed in separate sections. Some links already exist; they're built into the template to save you time, so you don't need to add them by hand.



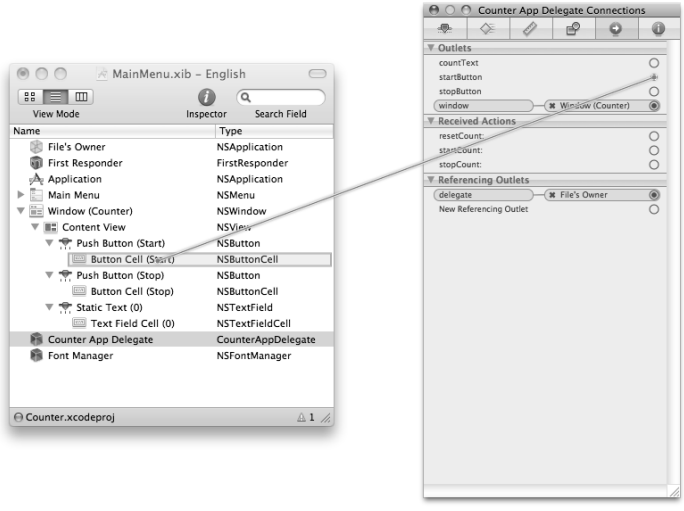
Two links are already present. The `delegate` property is linked to File's Owner, and the `window` object is linked to the window object in the nib. These premade links are built into the template.

Creating a link to an outlet

You'll start by linking one of the outlets. Click and hold the mouse on the circle to the right of the startButton outlet. Drag the cursor toward the Button Cell (Start) object in the Doc window. Release the mouse when the blue rectangle appears around the object, as shown in Figure 8.15.

Figure 8.15

Creating a link to an outlet. Drag a line from the outlet to a target object, then release the mouse. The new link (not shown here) appears in the Connections window.

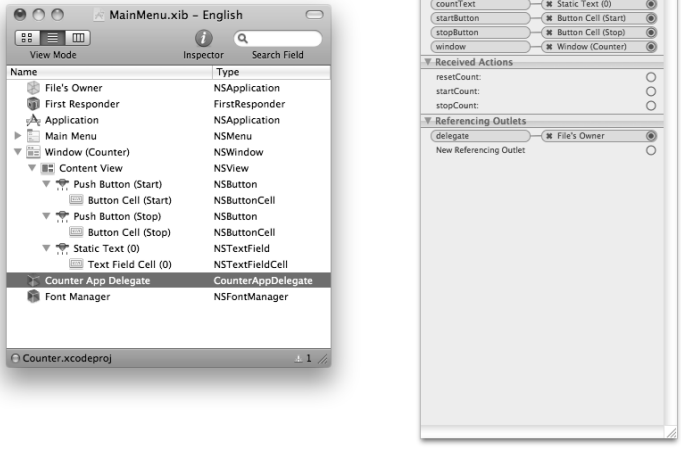


When you release the mouse, you'll see that a new link has been created. Repeat the process for the stopButton outlet, releasing the mouse on the Button Cell (Stop) object. Repeat it again for the countText outlet, but release the mouse on the Static Text (0) object.

Note that it's not possible to release the mouse on either of the Push Button objects, or on the Text Field Cell objects. This isn't accidental — IB doesn't allow you to create links to invalid destinations. Figure 8.16 shows the result of linking all three outlets.

Figure 8.16

After linking all three outlets, you should see this when you select the Counter App Delegate object. All three outlets have been connected to their counterpart objects in the nib.



TIP

You can drag links to a Doc window or to an Edit window. Dragging a link to an Edit window links an outlet to valid objects automatically. If you hold the mouse without releasing it when the blue rectangle flashes, the Edit window highlights the current target object for you.

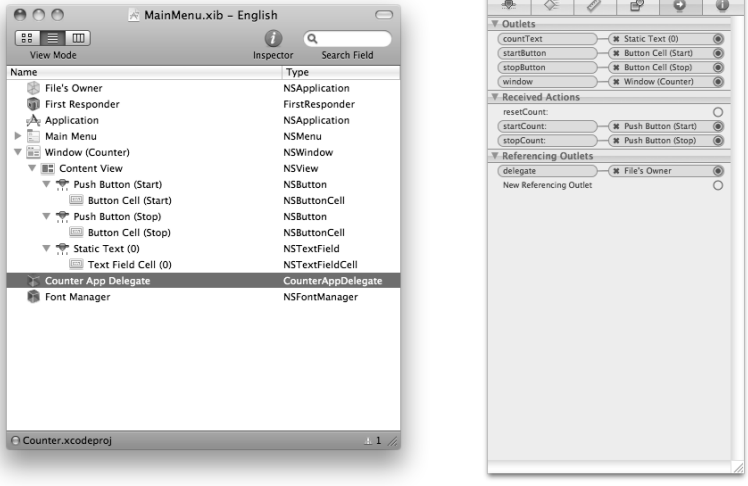
Creating a direct link to an action

Click the Counter App Delegate object to select it. Drag and release a line from the `startCount:` method to the Push Button (Start) object. Repeat for the `stopCount:` method, linking it to the Push Button (Stop) object. Leave the `resetCount:` method unlinked.

Linking to an action is slightly different from linking to an outlet. Cell containers and cell objects are interchangeable — either works as a link destination. For this first example, the linking is complete. Figure 8.17 shows the result. We're now ready to implement the core of the application. To recap, if you build and run the application, clicking on the buttons will trigger the stub methods in the code, and you can add code to set the properties of all of the objects. This code doesn't exist yet — and before we add it, we'll take a look at an alternative way to create links that illustrates one of Objective-C's most powerful features.

Figure 8.17

The start and stop buttons have been linked to their corresponding methods. Clicking the button triggers the method.



Creating a link using a selector

An action is a target, triggered by an event. In IB, event triggers are listed as Sent Actions. Although you've just dragged a link from the App Delegate to a button, in reality the message passes in the opposite direction, from the button to the delegate.

This distinction isn't critical in this application. But sometimes it's more convenient to drag a link from a button or other UI object to a link destination object, and then select one of its available methods.

Click the Counter App Delegate again, and click the diagonal cross to the left of the startCount: link. This deletes the link; it's the standard IB link removal option. You'll re-create the link using an alternative approach.

Click Push Button (Start). Under the Sent Actions tab, you'll see a single circle, labeled selector. A selector is a programmable method variable. It works like a pointer, but it points to a method and not to an object.

Think of a selector as a generic placeholder for methods. It can be filled with any method you choose. When you link to a selector, you fill the placeholder with a specific method. The method is triggered when the user interacts with the object.

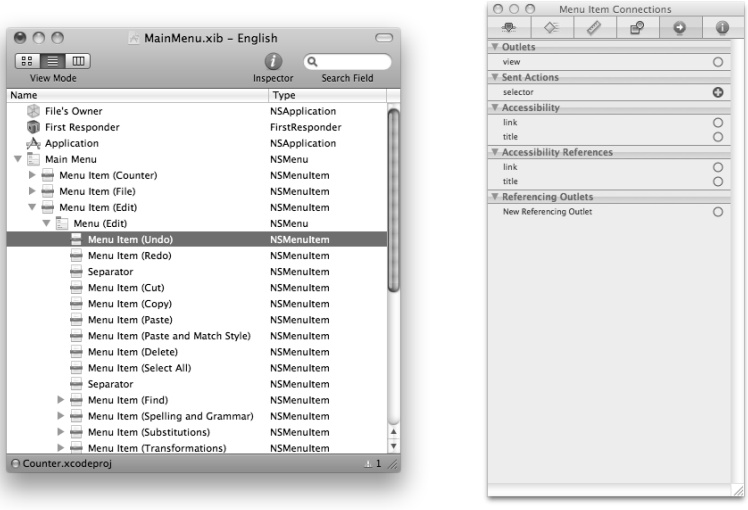
NOTE

For more information about selectors, see Chapter 9.

With a button, the method is triggered when the user clicks the button. Selectors are used throughout Cocoa. For example, click the Main Menu item, and use the reveal triangles to drill down to the Menu Item (Undo) object in the menu. You'll see that a sent action to a method called `undo:` is already defined. Click the delete cross. The method is deleted, and you can see that the menu item uses a selector to specify its action, as shown in Figure 8.18.

Figure 8.18

Many objects use a selector to define the method that is triggered when the user interacts with them. Selectors are the core feature of the Cocoa menu system.

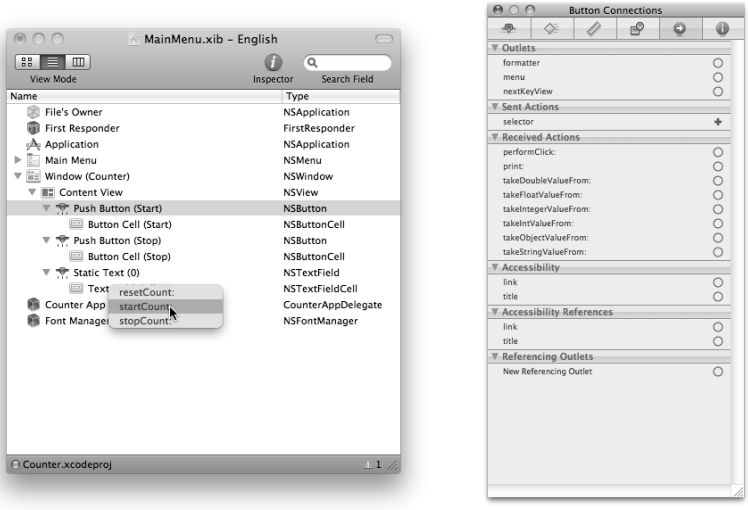


When you design an application menu, you specify a selector for each item. Each selector calls a different method in your code. The default menu includes a list of predefined methods. To implement each feature, you implement a corresponding method. You can customize the menu by changing the text labels for one or more items and by making their selectors point to new custom methods.

Creating a link to a selector requires an extra step that isn't needed when linking an outlet. Re-select Push Button (Start), and drag a link from its selector property to the Counter App Delegate object. When you release the mouse, you'll see a pop-up menu appear, as shown in Figure 8.19. The menu shows a list of available methods in the target object. Click a method to select it and complete the link; select `startCount:` in this example. When the linking is complete, you have re-created Figure 8.17. As a final step, *save the nib file* before you move back to Xcode to implement the rest of the counter code. Xcode doesn't remind you to save it before you build a project.

Figure 8.19

Using a selector or method pop-up. To create a link, select a method from the list. If the list is very long, you'll see up/down scroll arrows at the top and bottom.



TIP

Save the nib file. Did I mention that you should remember to save the nib file? *Save the nib file.*

Using NSTimer to create a simple seconds counter

There are many ways to implement a timed counter in Cocoa. The simplest is to use `NSTimer` — a lightweight but powerful timer object. `NSTimer` can be run as a one-shot counter or as a repeating counter. To create a repeating counter, add the following boilerplate code to `applicationDidFinishLaunching:`:

```
NSTimer *myTimer =
[NSTimer scheduledTimerWithTimeInterval: 1
                                 target: self
                               selector: @selector (timerMethod)
                               userInfo: nil
                                repeats: YES];
```

NOTE

`NSTimer` uses a very long nested signature. For convenience and clarity, it's useful to split the parameters across multiple lines. This is purely cosmetic. It doesn't affect the code, but it does make it easier to read and modify.

This creates a timer that calls `timerMethod` once a second. Setting `repeats` to `NO` would call `timerMethod` once. Note how `timerMethod` is defined in code with a selector. The principle is almost exactly the same as for a button — the method in the selector slot is triggered automatically by a timer event instead of a user action. The name of `timerMethod` is arbitrary. It must be unique and it must exist, but there are no other restrictions on the method name.

There's an extra detail to note. The `target` property specifies the object in which the selector method exists. You can use this feature to trigger a method in a different object; or, as is done here, you can specify `self` to trigger a method in the same object.

Finally, to stop a timer, use the `invalidate` method.

```
[myTimer invalidate];
```

This stops the timer and releases the timer object from memory. Optionally, you can set the timer pointer to `nil` after invalidating it. You can then test for `nil` to avoid invalidating the timer again, which is likely to cause a crash.

Implementing a timer method

A simple outline of the timer method looks like this:

```
-(void) timerMethod {
doThingsHere...
}
```

Add this code beneath `applicationDidFinishLaunching:`. It's possible to pass the timer object to a method. This feature isn't used in this application, but it's often useful when working with timers. Replace the selector with

```
selector: @selector (timerMethod:)
```

The extra colon is enough to tell Cocoa to pass a pointer to the timer object as a parameter, which you can then copy from the signature.

```
-(void) timerMethod: (NSTimer *) timer  {
doThingsHere…
[timer invalidate]; //Stop the timer
}
```

In this example, the timer method needs to increment a counter. It also needs to report the timer value to the view; specifically, it needs to update the text in the text field. It's easy to implement a counter with an `int`, and increment it at every timer tick.

```
-(void) updateTimer {
timerCount += 1;
}
```

Reporting the timer value is slightly more difficult. Many Cocoa objects include a `text` or `title` property that holds a text label. Uniquely, `NSTextField` uses a property called `stringValue` instead. To set the text of the label called `countText`, use

```
countText.stringValue = <any string>;
```

To convert a number into a string, use the `NSString stringWithFormat:` method. This implements standard C text formatting identical to that used in `printf` and `sprintf`; for example, `%i` interprets a number as an `int` and converts it to a text string. The method is implemented as an `NSString` class method. The format string must be "objectified" with `@` in the usual way. The final code for the timer method is

```
-(void) updateTimer {
   timerCount += 1;
   countText.stringValue =
   [NSString stringWithFormat: @"%i", timerCount];
}
```

This increments the count at each tick, and then writes the value to the label in the nib. Because you've linked the `countText` object to the text field in the view, updating the `stringValue` parameter automatically updates the count that appears in the view. In short, it just works. No more code is needed to display the value.

Implementing start and stop button methods

The button methods need to start and stop the timer and reset the counter variable. In this example they also reprogram the buttons dynamically. Initially the buttons are labeled Start and Stop. Only the Start button is enabled. Clicking Start renames the buttons to Reset and Continue. It modifies their selectors and disables the Start button so that the user can't create a second timer if one is already running. The complete code is as follows:

```
#import "CounterAppDelegate.h"
@implementation CounterAppDelegate
@synthesize window, startButton, stopButton, countText;
NSTimer *thisTimer; // Generic timer object pointer
int timerCount;
- (void)applicationDidFinishLaunching:
 (NSNotification *)aNotification {
    timerCount = 0; //Initialize the count on start-up
}
- (void) startCount: (id) sender {
    stopButton.action = (SEL)@selector(stopCount:);
      //Stop button does stop
   stopButton.title = @"Stop";
  [stopButton setEnabled: YES];
  [startButton setEnabled: NO]; //Prevent double timing
    thisTimer = [NSTimer scheduledTimerWithTimeInterval: 1
   target:self
 selector:@selector(updateTimer)
 userInfo:nil
  repeats:YES]; //Boilerplate timer initialization
}
- (void) stopCount: (id) sender {
  if (thisTimer != nil) {
    [thisTimer invalidate];
    thisTimer = nil; //Kill the timer
}
  [startButton setEnabled: YES];
  stopButton.action = (SEL)@selector(resetCount:);
     //Stop button does reset
  stopButton.title = @"Reset";
  startButton.title = @"Continue";
     //Identical to Start, but without clearing the timer
     //variable
}
- (void) resetCount: (id) sender {
  //Reinitialize everything on reset
  timerCount = 0;
  countText.stringValue = @"0";
  stopButton.action = nil;  //Stop button does nothing
```

```
    stopButton.title = @"Stop";
    startButton.title = @"Start";
    [stopButton setEnabled: NO];
    [startButton setEnabled: YES];
}
- (void) updateTimer {
    timerCount +=1;
    countText.stringValue =
      [NSString stringWithFormat:@"%i", timerCount];
}
@end
```

The methods cycle through different possible timer states, changing the labels on the buttons and the selectors they trigger. For example, in `startCount:`

```
stopButton.action = (SEL)@selector(stopCount:);
```

This line fills `stopButton`'s selector slot with the `stopCount:` method. When the user clicks the button, `stopCount:` is triggered.

A key feature of selectors is that they can be modified at any time. The `stopCount:` method updates the selector to trigger the `resetCount:` method, which in turn modifies the selector so that it does nothing. Selector switching is a simple but very powerful technique. You can use it to reconfigure a UI dynamically, so that the user triggers different events depending on the application state. You can also modify selectors internally, changing the events that are triggered as the application runs; for example, you might change the timer's selector so that it triggers a different method on every tenth count.

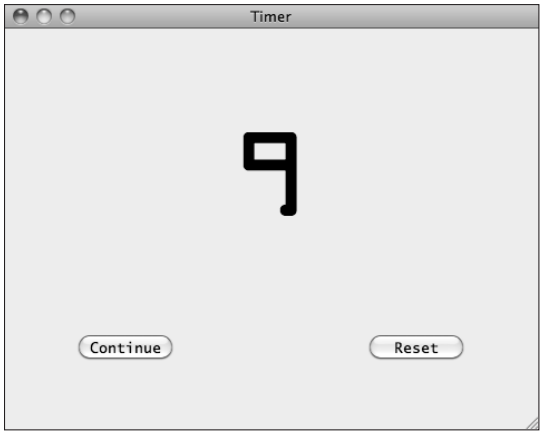
The remaining code updates the title property of both buttons to reflect their new functions. The `setEnabled:` property is used to enable and disable buttons dynamically, making it impossible for the user to click the Start button again when the timer is already running. When this property is set to `NO`, the button appears grayed out and the user can't click it.

You can now add the code, or view the sample on the Web site. The finished application is shown in Figure 8.20.

NOTE
The sample code is available at `www.wiley.com/go/cocoadevref`.

Figure 8.20

The finished application. The button labels are updated dynamically to indicate different application states. Buttons are also enabled and disabled dynamically.



Using Advanced UI Techniques

To create a basic UI in OS X and on iPhone OS, you need to know how to use the following:

- Selectors
- Cell objects
- Delegate objects, delegate messages, and `self`
- Outlets
- Actions
- Visual nib layout
- Attribute settings
- Linking

With these core skills, you can add almost any interface object to a Cocoa UI and create supporting code for it, using the documentation as a reference. For example, you can now create customized menus by changing the menu labels, adding and removing menu items, and defining selectors for active menu items.

But other Cocoa object interfaces aren't implemented consistently, and some objects require special tricks or knowledge. With these core skills, you can understand what needs to be done in outline, even when it's not obvious how it needs to be done in practice.

TIP

The easiest way to find worked solutions for difficult or inconsistent objects is to look for them online. Developer discussions often mention these gotchas. Finding a worked solution can save you hours of frustration.

With more advanced techniques, you can use UI objects in more flexible and elegant ways, creating simple and powerful interfaces with minimal code.

Using loose typing and (id) sender

Objective-C supports an open, generic placeholder pointer type named `id`. Use `id` in parameter lists and method returns when the type isn't defined until runtime or when the same method must handle different object types. For example:

```
- (AClass *) thisMethodDoesSomething: (id) aParameter;
```

`aParameter` can be of *any* data type. `id` is a catch-all pointer type, and it tells the compiler that the pointer points to . . . something. This is often used in UI event handlers.

```
-(void) handleAnEvent: (id) sender;
```

`sender` is the object that triggered the event. It can be any type of object: a button, switch, slider, and so on. Many Cocoa objects are designed with `id` in their signature because it would be unwieldy to create different subclasses and methods for every possible supported object in the signature.

`id` is typically used in four ways:

- **Case 1:** As a placeholder pointer when the type doesn't matter; for example, when a method saves a pointer to an object but doesn't need to access its features.
- **Case 2:** Where a Cocoa object uses `id` in a parameter or return field, but the application is deliberately limited to ensure that only certain types are used.
- **Case 3:** When the type must be discovered explicitly at runtime.
- **Case 4:** For direct object identification.

Using id with data collections

In Case 1, Cocoa's data collection objects — `NSArray`, `NSSet`, `NSDictionary` — implicitly save pointers of the `id` type. You can load any slot in a data collection with any kind of object, mixing them as needed. This makes collections very flexible and powerful. The same array can store numbers, strings, standard Cocoa objects, and custom objects, and move them between entries as needed without restriction.

NOTE

You can use the id data type in your own methods. It's a generic data type. Although it's often used with a sender parameter, it has other possible applications.

Using (id) sender with casts

In Case 2, objects must be "re-typed" with a cast before their properties can be accessed. For example:

```
- (void) anActionMethod: (id) sender {
aButton = (NSButton *) sender; //Assume sender is an NSButton
something = aButton.aProperty; //Access a property
…
}
```

This is a standard idiom in event handler methods. Use it when you are sure that, for example, `aParameter` is an `NSButton`. You can guarantee this by linking the interface in IB so that only `NSButton` objects trigger `aMethod:`. Do not use it when sender's type is not fixed.

This idiom is necessary because the `id` type doesn't support property accessors. The following isn't valid because the compiler doesn't know which properties are valid for an id type:

```
something = sender.aProperty;
```

The cast gives the compiler this information.

Using (id) sender with class checking

In Case 3, Objective-C supports a Class data type for classes, and you can use it to read a class name at runtime. Classes are managed with a predefined `Class` type. You can discover a class with

```
Class thisIsAClass = [sender class];
```

You can also convert the class name to a string for listing or testing with

```
NSString *aString = NSStringFromClass([sender class]);
```

This is sometimes useful for debugging. But you don't need to check class names as strings because you can compare them directly. To run code according to the class of an object, use the following:

```
//Check for an exact match
if ([anObject isMemberOfClass: [className class]])
 {doSomething…
//Check for a match while also checking inheritance
if ([anObject isKindOfClass: [className class]])
 {doSomethingElse…
```

The first conditional returns `TRUE` if and only if `anObject` is an instance of `className`. The match must be exact. The second returns `TRUE` if `anObject` is a subclass of `classname`. For example, all Cocoa objects are subclasses of `NSObject`, so setting `classname` to `NSObject` always returns `TRUE`.

You can use these conditionals to select code according to an object's class at runtime. This is a more complex solution than Case 2, shown previously, but it can support a much wider range of objects.

Potentially, you can create a single method to handle every possible event from an interface. This solution won't be simple or elegant, but it is possible to do this. More typically, you can use class checking in a `switch` to select handler code for a small number of different `sender` types.

Using (id) sender to identify objects

In Case 4, you can use the value of `sender` directly, in a direct comparison to identify the object that triggered an event. For example, your application might have three objects in its UI:

```
@property (assign) IBOutlet NSButtonCell *firstButton;
@property (assign) IBOutlet NSButtonCell *secondButton;
@property (assign) IBOutlet NSTextField *firstTextField;
```

You can link all of these objects to a single handler method, and use a pointer comparison to discover which object triggered the event.

```
-(void) eventHandler: (id) sender {
if (sender == firstButton) {
 //First button was clicked
}
if (sender == secondButton) {
 //Second button was clicked
}

if (sender == firstTextField) {
 //The text field did something
}

//Etc… }
```

This code is inherently loosely typed, and it is valid for all objects. It's particularly useful for UIs with multiple text fields.

Placing outlets and actions

A standard problem in UI design is deciding which object to link to. The simple example in this chapter is built around a single subclassed object — the app delegate. This minimal design is perfect for applications with a single view. You can implement menu and other event handlers in the delegate without creating further objects. This is somewhat nonstandard, but it's a simple solution and works.

In a more complex application, there may be many nib files, each with many objects. When designing multi-view UIs, you must do the following:

1. Define how objects send messages to each other.
2. Create links for those messages.

A key limitation of the nib editing system is that you cannot drag links between nib files. IB's interface doesn't support this. But the First Responder object is designed to offer a workaround for action linking.

NOTE

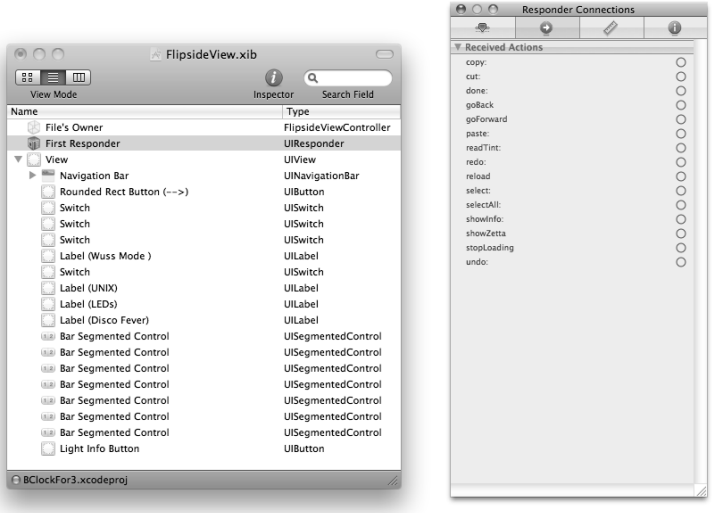
There's no way to link outlets across multiple nib files because the requirement doesn't make sense. Outlets are associated with a single object, which is in turn associated with a single nib file. Typically, all outlets in a nib are linked to File's Owner or, occasionally, to a delegate object.

When you create an action method in any object in the application, it's added to the responder chain. When you click First Responder, you'll see a list of received actions. This list includes all the action methods in the application, collected from every object. You can link to these actions in the usual way — and this solves the cross-linking problem.

Figure 8.21 illustrates this with an example from an iPhone application. The `showZetta` and `showInfo:` methods are in different objects, but both appear as linking destinations inside First Responder. When an application has multiple nibs, the same list of First Responder methods appears in all of them.

Figure 8.21

Using First Responder to view all of the action methods in an application. You can use this feature to send messages to objects that aren't in the current nib.



CAUTION

First Responder doesn't indicate if a method is already linked directly as a selector. You can easily create multiple links to the same methods. This may not be a problem in your application, but it can be useful to double-check whether or not a method is already linked within an object before trying to link it again in First Responder.

Placing NSResponder events in OS X

First Responder also supports a very long list of standard methods that can be used in every application and which are supported by many objects. These are defined in the documentation for `NSResponder`, which was introduced in Chapter 6.

To implement these methods, follow this two-step process:

1. **Link a suitable object to them, so that they're triggered by a user action.** The menu items in the default menu are prelinked to some of the methods for convenience.
2. **Implement a handler method in a suitable object in the application.** For example, to implement undo events, create a method called `undo:`. You can place the method in any object that is a subclass of `NSResponder`.

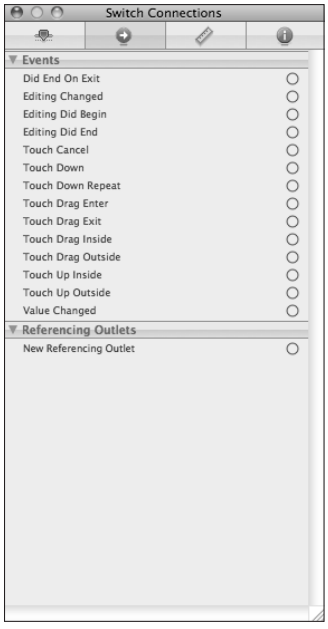
Using standard event messages on the iPhone

The iPhone supports a simplified version of `NSResponder`, with a smaller selection of standard messages. It also supports a modified selector system, which generates yet another list of messages. Instead of a single blank selector slot triggered by a click, iPhone objects respond to different user actions with different event messages. Figure 8.22 shows the standard event list. This list is available for most objects, but is implemented differently in different objects. For example, a button doesn't send the Value Changed message, but a slider does; a text field doesn't send Touch Down, but it does send Editing Did Begin; and so on.

This is a refinement of the selector system; you can define different handlers for different event types and for objects within them, but in practice it works in a similar way. Link each event to the method in your code that it handles. To ignore an event type, leave it unlinked.

Figure 8.22

Events on the iPhone. This list is standardized and available in many UI objects, but most objects can only send a subset of these messages.



Summary

This chapter has covered a lot of ground. You may need to work through it a couple of times to take it all in, but it's worth the time because it explains some of Cocoa's key practical concepts.

At the start of the chapter, you were introduced to a typical Interface Builder workflow. Next, you found out how to add objects to a nib, how to center and align them, and how to modify their default properties.

In the next section, you were introduced to outlets and actions, and how to define them in code. You also learned about the Interface Builder linking process and how to link objects in code to objects in a nib file. You then explored a simple application that used selectors to swap handler methods dynamically, creating a UI that reconfigured itself.

Finally, you were introduced to some more advanced techniques used in event handlers, including `(id) sender` idioms and First Responder events.

Going Deeper

In This Part

Chapter 9
Using Cocoa Design Patterns and Advanced Messaging

Chapter 10
Working with Files, URLs, and Web Data

Chapter 11
Using Timers, Threads, and Blocks

Chapter 12
Managing Data and Memory in Cocoa

Chapter 13
Using Preferences and Bindings

Chapter 14
Using Core Data

Chapter 15
Working with Text and Documents

9 Using Cocoa Design Patterns and Advanced Messaging

It's traditional in a Cocoa book to mention design patterns and features such as Model-View-Controller (MVC), target-action, Key-Value Coding (KVC), and Key-Value Observing (KVO).

Apple's documentation mystifies these design patterns and makes them seem more difficult and complex than they really are. They're an essential part of Cocoa, and you can use them to implement impressively efficient code. But the practice can be much simpler than the theory.

Understanding Model-View-Controller

Model-View-Controller, often shortened to MVC, was introduced in Chapter 2. The outline of MVC is shown in Figure 9.1. An application is split into three elements — a store of data called a model, an interface called a view, and a controller object that passes data between them.

MVC implies, but doesn't enforce, the suggestion that model data should be kept in separate objects and perhaps in separate custom classes.

In a simple application, this seems unnecessarily complex, because it is. If your data is in a single array or dictionary, there's nothing to be gained by taking a data object out of the App Delegate and putting it into a separate wrapper object. It can be useful in a larger application to keep data in a separate data-handling class. But generally, the key feature of MVC isn't how the data is stored but how it's processed and prepared for display by the controller.

9 In This Chapter

Understanding Model-View-Controller in Cocoa

Understanding and using target-action

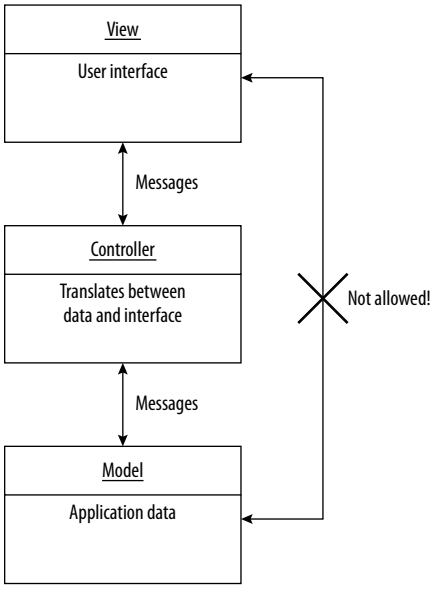
Using Key-Value Coding

Using Key-Value Observing

Working with notifications and errors

Figure 9.1

MVC in outline form. The model and the view don't communicate directly. The key to MVC is implementing an efficient and, optionally, creative controller design.



A controller can do more than link the view and the model. When MVC is done properly, the controller layer adds useful benefits, typically some of the following:

- **Increased efficiency.** The controller can select some of the data for display, as needed, rather than attempting to access all of it at every refresh. Typically a table displays a small selection of values from a data source. A well-designed controller accesses only the values it needs to create a useful display or implement an editing feature.
- **Abstraction and reuse.** The same controller code and view objects can be reused and can work with different data sources.
- **Format translation.** A simple controller can support multiple types, interconverting between strings, number values, and Booleans. A more complex controller can take a list of numbers and convert them into a table, a scatter plot, a graph, and so on, suitable for display by an appropriate view object.
- **User control.** A controller can manage selection events, handle multiple selections, and implement editing operations. It can also sort values for display, and it can optionally produce other summary displays, such as averages.

To use MVC effectively, concentrate on the controller code and consider how it can add value to your application. Figures 9.2 to 9.4 outline some practical examples.

Figure 9.2

Using MVC in a game. The controller translates the game data into a format that can be processed by the game's rendering engine, and it maintains the viewport properties that define the view of the game world seen by the user.

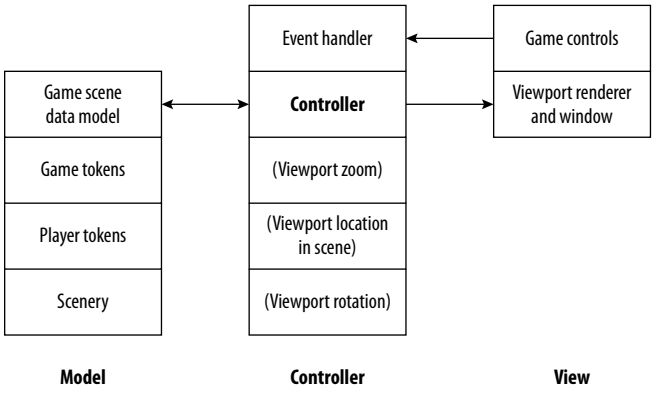


Figure 9.3

Using MVC to manage a database. MVC is a natural design pattern for databases. The controller manages edit events and can reinterpret the data to display it in various ways, for example, as graphics, tables, sorted summaries, and so on.

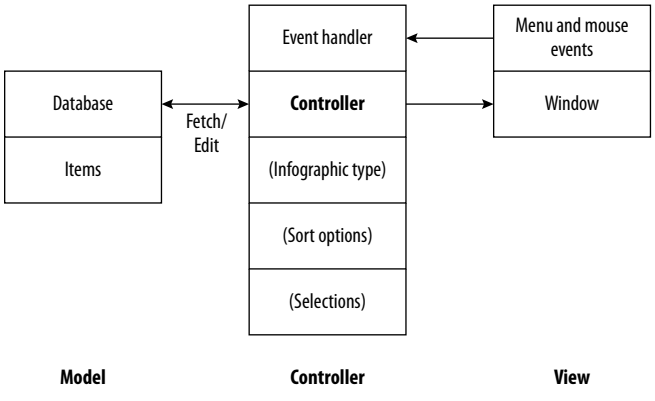
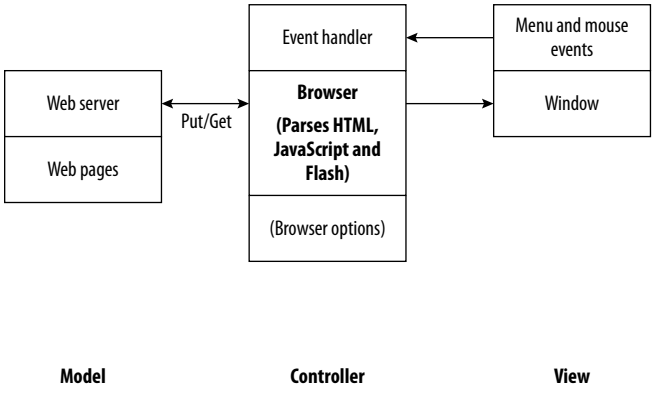


Figure 9.4

Using MVC in a browser. Browser applications support MVC almost automatically. It would be very inefficient to download the entire contents of a Web server to display a single page. Instead, the MVC pattern makes it possible to interact with each page one at a time. The controller translates the HTML and other scripts on each page into a visible view, and it manages user action with the data model.



CAUTION

Model-View-Controller isn't related to *modal* views. In Cocoa and Cocoa Touch, a modal view takes control of the interface and locks out other interaction. For example, when a save/open sheet appears, the user can only select a file or click the Cancel button. The rest of the interface stops responding until the sheet is dismissed.

Using MVC with Cocoa controller objects

Some of Cocoa's classes, such as `NSTableView`, require explicit translation code in a controller. For example, to display a table of values from a data source array in a table view, implement two delegate methods:

```
-(int) numberOfRowsInTableView: (NSTableView *) tableView {
return [dataSourceArray count];
}
- (id)tableView:(NSTableView *)tableView
      objectValueForTableColumn:(NSTableColumn *)tableColumn
      row:(int)row
{
      return [dataSourceArray objectAtIndex:row];
}
```

These two methods perform the auto-enumeration introduced in Chapter 2. Cocoa uses them to ask your application for a count of items to display in the table column, and then to ask for an object to display in each row.

Similar code is needed whenever your application displays the contents of an array in a column. To save reinventing the wheel, it's useful to have a generic controller class that can solve this problem.

You could create your own solution, but Cocoa includes classes that implement a solution for you. *Controller objects*, such as `NSArrayController` and `NSDictionaryController`, abstract the relationship between a data source and a view object and provide translation, sorting, and selection features that make it easy to design an interface with tables and other UI elements that interact with the contents of a data collection object, such as `NSArray` and `NSDictionary`.

To the controller, the data source array is a generic array, and the UI object is a generic multi-object view. The controller doesn't care what the data objects represent. This is a useful feature because it makes the controller universal; it can manage any data.

In Cocoa, MVC has been applied in a specific way, and controller objects are the most obvious practical implementation of the pattern. Each controller works slightly differently, so to use them effectively you must learn how they capture, translate, and output data. The concepts and techniques are introduced in detail in Chapter 13. For now it's enough to know that MVC isn't theoretical — it's implemented in Cocoa in specific objects that are designed to make MVC easy to use. These objects can add extra features to your application that you might not have been able to include otherwise.

Creating custom controllers

At its most basic, a controller must implement data source access and UI update methods. These methods are often plain and unexciting glue code. This code must be included but isn't fun to write.

You can add benefits to your application by treating controller design as a creative opportunity rather than a chore. Good applications present data in transparent, intuitive ways. You can use the controller layer to add features that summarize or preprocess data and to design a UI that implements the new possibilities.

There are many opportunities for innovation in controller and UI design. For example, most applications use a three-step (select-initialize-implement) design pattern for their UIs. First the user selects a feature, then he or she initializes its settings, then the feature is applied to modify or display the data.

One of the goals of iPhone and iPad app design is to streamline this process to eliminate unnecessary steps. For example, in some contexts it's possible to create apps that save and load settings and data in the background, without an explicit "Save settings" step. User data persists automatically.

You can apply the same principle to other application features, and also to desktop application design. Applying MVC creatively makes it possible to make applications that are more appealing, intuitive, and valuable.

Defining the data model

Cocoa's data collection objects — `NSArray`, `NSDictionary`, `NSSet`, and others — are designed to simplify data management. Key-Value Coding (KVC), described in detail later in this chapter, is one of the fundamental data access patterns. You *must* understand KVC to use Cocoa effectively, because it's used by a significant selection of Cocoa objects. Custom data models often use KVC by default. Other design patterns are possible, but KVC is built into Cocoa and is a natural fit for data management.

The data model should also support *archiving* — file save and load. Data archiving features are built into Cocoa objects, and it's possible to save and load an instance of most classes, with their current property values. Some classes require extra code to implement this; others work as is. For more information, see Chapter 10.

Cocoa also includes a more complex technology, called *Core Data*, that makes it possible to build a model that defines *entities* — object-like data containers with properties — and relationships that link entities to each other. Core Data is optional, but it offers the benefit of a persistent managed store of entities that can be saved and reloaded as a single combined object.

Taken together, a good Cocoa data model should be the following:

- **Archivable.** This is true by default for most objects.
- **KVC compliant.** This gives Cocoa the best possible flexibility for data access.
- **Built from standard data collection objects such as** NSArray**.** This simplifies archiving and also supports KVC compliance.

While you can build a data model from standard C data types, it's unlikely to meet these requirements. The code interface may be simpler, but it won't support the features that Cocoa needs to access it efficiently. Without those features, MVC becomes difficult to implement.

Understanding Target-Action

Now that you've looked at some outline notes about MVC, it's time to take a closer look at some related design patterns. MVC is a top-level design pattern that sketches the features of the application as a whole. Target-action is a lower-level pattern that defines how objects message each other. You were introduced to some examples of target-action in the previous chapter. Now, you'll look at it more closely.

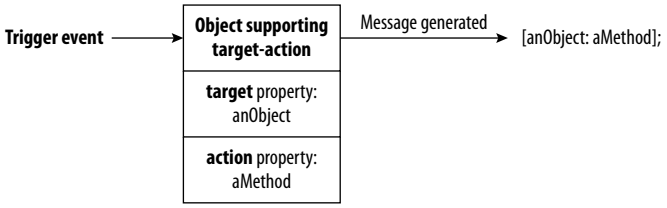
Target-action is uncomplicated. A typical objective-C message looks like this:

```
[anObject aMessage];
```

When objects support target-action, the object and the message are stored as properties, and an external event triggers the message, as shown in Figure 9.5. The trigger event is often a user action, but in some applications it can be a timer event or a generic event scheduled by an object.

Figure 9.5

Using target-action when the target and the action are specified as properties.



Some methods also support target-action, specifically timer methods and a method called `performSelector:`. The target and the action are defined in the method body.

```
[anObject doSomethingThatSupportsTargetAction
target: anObject action: @selector(aMethod)…];
```

Both variations create the same result: `aMethod` is triggered in `anObject`.

CAUTION

If `aMethod` isn't found in `anObject`, an error is generated and the application may crash.

The key benefit of target-action is that both the target and the action can be changed under program control. You can dynamically reconfigure your application to redirect messages from one object and method to another, as needed.

In practice, this happens less frequently than it could. Typically a target-action is defined at launch, and it isn't modified again. But sometimes it's useful to modify targets and actions dynamically. Target-action makes it possible to do that.

Defining selectors

To define an action method, you must wrap it in a *selector* — an "objectified" method. You can use selectors as objects, copying them, archiving them, comparing them, and performing other tricks. More typically, you use the selector syntax as a bit of boilerplate syntax that wraps up a method so that it can be plugged into target-action code.

The syntax is slightly unusual and can be difficult to remember. It looks like this:

```
@selector(aMethod:)
```

Key points to remember:

- There are no quotation marks anywhere in the statement.
- The brackets are round.
- If the method takes a parameter, a colon is included. Parameters aren't listed.

SEL is the associated data type. You can define a SEL variable and use it in place of a literal selector.

```
SEL theSelector;
if (aCondition)
theSelector = @selector(methodOne:);
else
theSelector = @selector(methodTwo:);
anObject.action = theSelector;
```

CAUTION

SEL doesn't define a pointer. Don't put an asterisk in front of a SEL variable.

Using selectors in code

Target-action is often, but not exclusively, used with subclasses of NSControl. This includes most UI objects such as buttons, menus, and sliders. This code is valid for all of its subclasses:

```
[aControl setTarget: anObject];
[aControl setAction: @selector(aMethod:)];
```

Where aControl is @synthesized, dot syntax is also valid.

```
aControl.target = anObject;
aControl.action = @selector(aMethod:);
```

You can read an action and compare it with a selector directly with ==:

```
If (aControl.action == @selector(aMethod:)) {…}
```

You can use this to test whether you need to modify the selector or to process an event according to the current selector state.

A property called continuous is an optional feature implemented in NSControl. When continuous is TRUE, a control fires its selector while the user holds the mouse down. Otherwise, it generates one single message for each click. Use this feature to implement auto-increment and auto-decrement.

Understanding the limitations of selectors

The action system has an obvious limitation: you can't use it to pass parameters to the target method. Some variants of target-action support an extended syntax with an extra parameter that can pass an object to the selector:

```
[anObject setAction:
@selector withObject: anObjectUsedAsAParameter];
```

This is only available in limited circumstances. If you need to pass parameters to your selector method, pass them externally through some associated properties or other data objects.

The other limitation is that a selector *isn't* a pointer. It doesn't point to a specific entry point in memory. Instead, it triggers a runtime lookup of a method name, which returns a pointer. This can be an ambiguous process, because more than one object may implement the same method name.

Generally, selectors are local. If there's no explicit `target` object, the lookup assumes that you're trying to find a matching method in `self`. This is intuitive behavior, and rarely a problem. But keep in mind that selectors don't usually range over the whole of your application, except when there's a supporting `target` property.

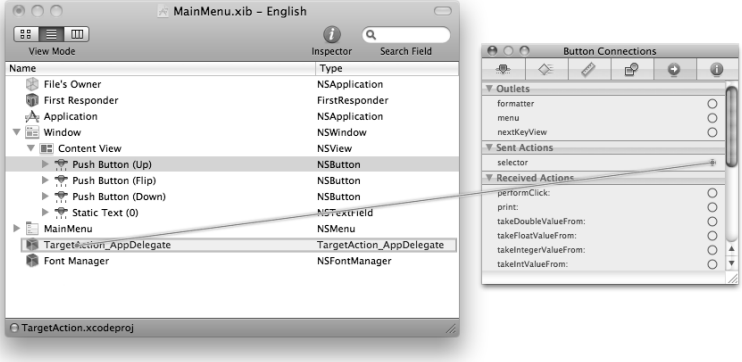
This short summary is almost all you need to know about target-action. It's an unusual feature that isn't available in most other languages, but in practice it's very unmysterious.

Defining selectors in Interface Builder

When you create an `IBAction` link in Interface Builder (IB), you're really setting an object's target and the action properties. This is why setting an action is a two-step process. Assume you have a nib file with a button and you're going to link the button to an action. The first step selects the target object graphically, as shown in Figure 9.6.

Figure 9.6

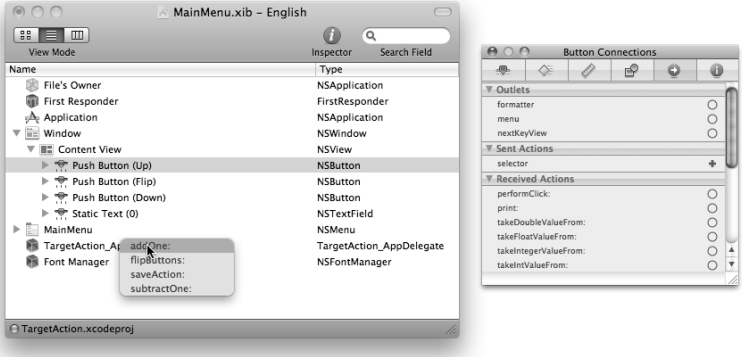
When creating a link, the first step sets the target object for the link.



In this example, this step defines the button's `target` object as the App Delegate. The second step sets the action method, as shown in Figure 9.7.

Figure 9.7

Setting the action method by selecting from a pop-up list



In code, you can select any public method as a valid action. In IB, only methods that are defined as an `IBAction` appear in the pop-up menu. It would be convenient if IB listed every compatible method in the target object — but it doesn't. You can work around this by defining the actions in code when the application loads. Use this to choose any valid selector in any object, including objects that your code creates dynamically.

CROSS-REF

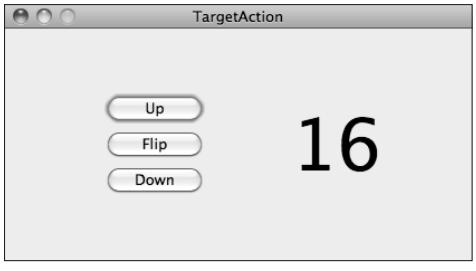
For an example of dynamically created buttons with an assigned action, see the random button example in Chapter 16.

Creating an example application

Figure 9.8 shows a simple application that illustrates target-action. It's a slightly extended and simplified variation on the counter application from the previous example. Two buttons create a counter that counts up and down. The timer is replaced by the `continuous` property. When this is enabled, trigger events are generated while the mouse button is held down.

Figure 9.8

A simple target-action application. Selecting the Flip button swaps the selectors in the Up and Down buttons.



The buttons are linked to `addOne:` and `subtractOne:` methods that change the value of a timer variable. The value is displayed in a text field. Figure 9.9 shows the elements and links in the nib file.

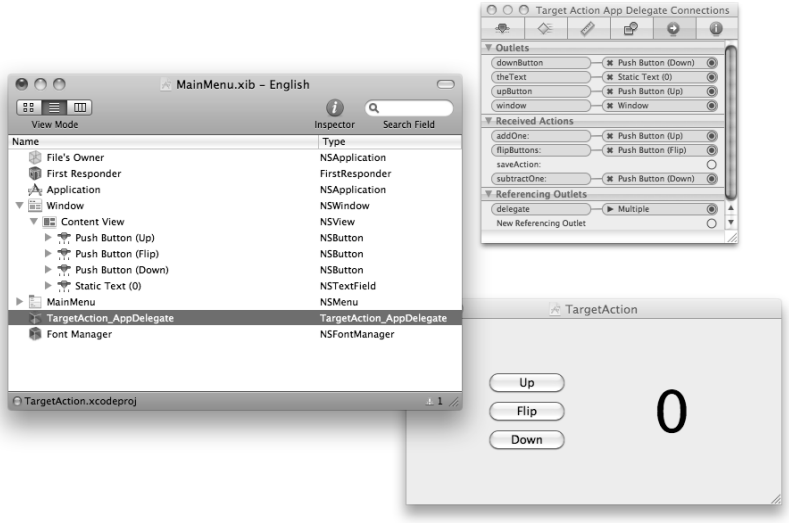
`upButton`, `downButton`, and `theText` outlets link to the properties of the two main buttons and the text field. The `addOne:` and `subtractOne:` methods control a counter. The `flipButton` action triggers the button flip method.

The header file for the project defines the objects, outlets, and actions that are used.

```
#import <Cocoa/Cocoa.h>
@interface TargetAction_AppDelegate : NSObject
{
    NSWindow *window;
NSTextField *theText;
NSButton *upButton;
NSButton *downButton;
}
@property (nonatomic, retain) IBOutlet NSWindow *window;
@property (nonatomic, retain) IBOutlet NSTextField *theText;
@property (nonatomic, retain) IBOutlet NSButton *upButton;
@property (nonatomic, retain) IBOutlet NSButton *downButton;
- (IBAction) addOne: (id) sender;
- (IBAction) subtractOne: (id) sender;
- (IBAction) flipButtons: (id) sender;
@end
```

Figure 9.9

The project nib file: it's straightforward and has no unusual features.



The implementation file implements the button action methods. The code for the `addOne:` and `subtractOne:` methods updates the counter and writes its new value to the text field.

The `flipButtons:` method swaps the title and action of both buttons, and it also toggles their continuous property. When the buttons are flipped, single clicks trigger a single update of the counter. Otherwise, the count changes while the mouse button is held down. The code also includes a selector comparison that checks the current flip state.

```
#import "TargetAction_AppDelegate.h"
@implementation TargetAction_AppDelegate
@synthesize window, theText, upButton, downButton;
int theCount;
-(IBAction) addOne: (id) sender{
   theCount +=1;
   theText.stringValue =
   [NSString stringWithFormat:@"%i", theCount];
}
-(IBAction) subtractOne: (id) sender{
   theCount -=1;
   theText.stringValue =
```

```
    [NSString stringWithFormat:@"%i", theCount];
}
- (IBAction) flipButtons: (id) sender {
    if (upButton.action == @selector(addOne:)) {
        upButton.action = @selector(subtractOne:);
        [upButton setContinuous: NO];
        upButton.title = @"Down";
        downButton.action = @selector(addOne:);
        [downButton setContinuous: NO];
        downButton.title = @"Up";
    } else {
        upButton.action = @selector(addOne:);
        [upButton setContinuous: YES];
        upButton.title = @"Up";
        downButton.action = @selector(subtractOne:);
        [downButton setContinuous: YES];
        downButton.title = @"Down";
    }
}
@end
```

Other applications of selectors

Table 9.1 lists some other ways that selectors and target-action are used in Cocoa.

Table 9.1 Selected Advanced applications of target-action and selectors

Application	*Description*
`NSTimer`	Supports both target and action. Selects a method that is called on each timer tick or after a delay.
`performSelector:`	A method built into `NSObject` and supported by most classes. Supports both target and action. Selects a method that can be run immediately or after a delay. Supports explicit multithreading.
`performSelectorInBackground:`	A very simple way to run a method in a separate background thread.
Data collection objects: `NSArray` and so on	Include a `makeObjectsPerformSelector:` method that automatically enumerates through every object in the collection and runs the selected method on it.
Core Animation	Certain Core Animation classes can trigger a selector when an animation completes. Use this for composite multiple animations or to control user input while the animation runs.
`NSInvocation`	Wraps a selector and a target into a combined object with parameters to create a powerful composite. Sometimes used to manage Cocoa's undo system.
`canPerformSelector:`	A method that can be run on any object to check whether it implements a selector. Use it to avoid crashes and error messages.

Using Key-Value Coding

Like selectors, Key-Value Coding (KVC) introduces indirection into Objective-C. But KVC is a very different and unrelated technology, with very different applications.

KVC makes it possible to access properties indirectly, using their name strings.

```
aReturn = anObject.aProperty;                    //Direct access
aReturn = [anObject valueForKey: @"aProperty"]; //KVC access
```

KVC looks simple in theory, but it can be tricky in practice. Some of the issues include:

- **Values must be "objectified."** The technology could more accurately be called Key-Object Coding, because, with a few exceptions, values must be wrapped inside objects.
- **KVC isn't usually used for standard property access.** It's easier to use standard setters, getters, and `@synthesize`.
- **KVC is widely used to initialize and return groups of values and to access the values in data collection objects.** You must master it to use Cocoa effectively.
- **Cocoa's keypath management can seem erratic.** You can use `object.property.subproperty` syntax to define a keypath. But not all classes support keypath access, and classes that do support it don't always implement it consistently. This particularly applies to mutable data collections.
- **KVC makes it harder to trap errors.** If you use string literals as keys, the compiler can't check them. You can solve this problem by defining the key strings as constants in a header file and using them throughout your code.

"Objectifying" values

Consider this simple example of KVC, which reads and sets some of the values of `NSWindow`. This code looks like a reasonable attempt to set the `hasShadow` property.

```
[window setValue: NO forKey: @"hasShadow"];
```

But it doesn't work, because `setValue:` expects an object, not a `BOOL`. The correct code is:

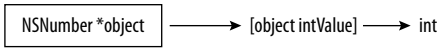
```
 [window setValue: [NSNumber numberWithBool: NO]
forKey: @"hasShadow"];
```

This demonstrates how values must be objectified, adding an extra layer of complexity to KVC. Figure 9.10 illustrates this.

Figure 9.10

Converting numerical types into `NSNumber` objects, and back again. You can't use KVC on numerical types without "objectifying" them.

Converting a numerical type to an NSNumber object



Converting an NSNumber object to a numerical type

CAUTION

If there's a `set...` method, convert the first character after `set` to lowercase. For example, `NSWindow` has a `setHasShadow` method. The corresponding key is `hasShadow`.

Using NSNumber

Strings are already objectified with the `@` character. To objectify numerical values, wrap them in an instance of `NSNumber`. All numerical types, including `int`, `float`, `BOOL`, and others, must be wrapped in a similar way. For the full list of supported types, see the `NSNumber` class reference.

For example, use this code to set a `float`:

```
 [window setValue:
[NSNumber numberWithFloat: 0.75]
forKey: @"alphaValue"];
```

`NSNumber` supports named `<type>value` methods that translate an `NSNumber` back into its equivalent simple type. Equivalent code to read a value looks like this:

```
BOOL returnFlag =
[[window valueForKey: @"hasShadow"] boolValue];
float aFloat =
[[window valueForKey: @"alphaValue"] floatValue];
etc…
```

TIP

When creating new keys, it's useful to include a type definition in the key name as a reminder or "code tool tip."

```
@"something"     //Bad
@"somethingBOOL" //Good
```

Using nil

Because `NSNumber` is an object type, it's possible for its value to be `nil`. If there's a danger that your code may access a value that hasn't been set, it's important to test for this:

```
if ([window valueForKey: @"hasShadow"] ! = nil)
 BOOL returnFlag = [[window valueForKey: @"hasShadow"]
   boolValue];
```

Cocoa keys are likely to be initialized. Custom objects may or may not include initialization code, so it's good practice to make sure that all possible values are initialized correctly or that returned values are tested for `nil` before being used.

Using NSNull

Confusingly, collection objects — `NSDictionary`, `NSArray`, `NSSet`, and so on — use a different but equivalent value called `null`.

`nil` is an empty pointer, while `null` is an object used as a placeholder for a missing and undefined value. `null` is defined as the output of the `NSNull` class, and it is an instance of `NSNull`.

```
NSNull *aNullValue = [NSNull null];
```

This bizarre construct is the only way to access the `null` value.

Avoiding errors

If the KVC system can't find a key, an error is raised. You can control KVC errors by implementing the `valueForUndefinedKey:` method in the object you're accessing. Add this boilerplate method signature, implementing the method to create a useful return, such as an error value that your code can trap:

```
-(id) valueForUndefinedKey: (NSString *) key
{
//return a default value to avoid generating an exception
}
```

An equivalent `setValue: forUndefinedKey:` method is triggered when you try to set the value of an unrecognized key. Overriding this method with an empty implementation is the easiest way to avoid generating an on-write exception.

Using Key-Value Observing

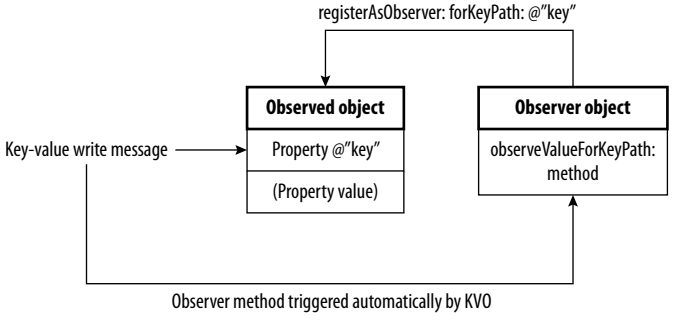
Key-Value Observing (KVO) is a related technology that links keys to an observer method. Whenever the value is updated, the observer method is triggered. You can use this to monitor any property in any object that is KVO compliant.

KVO literally means that when a value is modified, the observer method is triggered — a form of remote viewing. KVO works even if there's no other connection between the object that's being watched and the object that is doing the watching.

The observer method receives a dictionary containing details of the changes. The dictionary can be configured to report any combination of the old, new, original, and prior values. The observer method can be in a different object. Figure 9.11 illustrates how this works.

Figure 9.11

Understanding KVO. The `registerAsObserver:` method is run once to enable KVO. The `observeValueForKey:` method is run each time the value of the keypath is modified.



Like KVC, KVO is built into `NSObject`. To implement it, add a method to register an object:

```
- (void)registerAsObserver
{
[objectToWatch addObserver: watchingObject
     forKeyPath:@"observedProperty"
        options:(NSKeyValueObservingOptionNew |
   NSKeyValueObservingOptionOld)
        context:NULL];
}
```

This initializes `watchingObject` as an observer for the `observedProperty` key in `objectToWatch`. Objects can observe themselves, so both objects can be `self`.

The `options` field ORs together two selection flags and sets up the dictionary to return both the old and new values. `context` is a free field that can be used to pass any pointer to the observer.

To process updates, add the following method to the observer. The method definition is a boilerplate list of passed parameters. The same method body can handle updates for multiple properties, so it's useful to include a list of conditionals that runs separate code for each.

```
- (void)observeValueForKeyPath:(NSString *)keyPath
        ofObject:(id)object
          change:(NSDictionary *)change
         context:(void *)context {
 if ([keyPath isEqual:@"observedProperty"]) {
    //Do something
    }
}
```

TIP

Although the method is called `observeValue...`, it's more accurate to think of it as `changedValueWasObserved:` because it's triggered when a changed value is observed. Technically, it's meant to imply that the method should observe the value and do something with it. But the name doesn't make this obvious.

This example reads the most recently updated value for the key from the dictionary and passes it to a `doSomethingWith:` method (not defined here) for further processing. There are no restrictions on the complexity of the observer method. It's possible — but not useful – to build most of an application's features into the observer code.

If the object is observing more than one other object, you can also compare `(id) object` to a list of possible targets, with a direct pointer comparison.

The `observeValueForKeyPath:` method is triggered whenever an observed value is changed. To remove an observer, use

```
- (void)unregisterForChangeNotification
{
 [objectToWatch removeObserver: watchingObject
                    forKeyPath:@"observedProperty"];
}
```

These methods can be run at any time. You can enable and disable observation dynamically.

It's important to understand that KVO is *object-focused,* not *key-focused*. Any object or combination of objects can observe a given keypath. Each observer is independent, and their observer methods can respond to a change in independent ways. Adding or removing an observer has no effect on other observers. It's sometimes useful to turn off observations in one object while allowing them to continue in other objects.

Making assignments KVO compliant

When you `@synthesize` object properties, they automatically become *partially* KVO compliant. When another object updates a property, KVO is triggered. But when code inside an object updates a property, KVO does nothing.

KVO plugs itself into an object's accessor methods. Direct assignments don't use accessors, so KVO can't observe them. This can be a problem, because some assignments are internal.

As an example, assume you have an object called `myObject`, an instance of `MyClass`, which has a property called `thisInt`. The following code will trigger a KVO update *as long as you run it inside another object:*

```
myObject.thisInt = 15;
```

The accessor method is triggered by an external property write. But if you include internal update code within `myObject`, the assignment doesn't use an accessor, and so the assignment is invisible to KVO.

```
@implementation MyClass
…
thisInt +=10;
```

One solution is to force updates with explicit code:

```
[self willChangeValueForKey: @"thisInt"];
thisInt = thisInt + 10;
[self didChangeValueForKey: @"thisInt"];
```

This is ugly and is a lot of extra trouble when you want to add one number to another. Other solutions are possible, but the most elegant is to use `self` and dot syntax for critical assignments:

```
self.thisInt = thisInt+10;
```

Specifying `self` forces the assignment to use an accessor, which solves the problem in a simple and readable way. Use it for *critical* assignments only. To improve performance, don't use it for intermediate steps in a calculation.

You must also check that accessors are used when working with bindings (described in Chapter 13). Bindings use KVO, and if you don't set values with accessors, they don't track updates correctly.

CAUTION

This solution only works with dot syntax. If you don't approve of dot syntax, you'll have to use ugly, old-fashioned constructions like `[self setThisInt: [self thisInt] +10];`.

Using KVO

KVO can be immensely powerful. Your application can monitor events anywhere in your code and respond automatically to given values or application states. Setting a value in one place in the code can trigger a chain of dependent events automatically. Many Cocoa objects are KVO compliant; your application can watch for changes in Cocoa object properties and respond as needed. Some applications include the following:

- **Responding when a background process writes a value to a property after completing a calculation.** Instead of using a delegate method or explicit method call, your application can monitor the property and capture the update automatically.
- **Monitoring changes to application preferences.** The application can track updates as a user makes them on a separate preferences pane, without linking (see Chapter 13 for an example).
- **Copying values from one location to another.** KVO makes it much easier to implement spreadsheet-like mutual dependencies.

Understanding KVO limitations

KVO isn't supported by every object, and sometimes it's only partially supported. This is particularly true of mutable data collection objects. For example, KVO is triggered when you create, assign, or copy an array, but not when the contents of the array are modified. This means KVO is blind to array edit operations. You can fix this with a dummy assignment after you modify the array:

```
self.theArray = theArray;
```

This is an ugly workaround, but you can use it to notify a KVO observer that the array has changed or to trigger a KVO observer method on demand.

KVO and nil/null values

In the same way that KVC can return `nil`, KVO can return `null` values. For robustness, it's useful to test for this by testing for `[NSNull null]`, which is Cocoa's way of specifying a null value.

```
if((newValue[change valueForKey: @"new"]) != [NSNull null]) {…
```

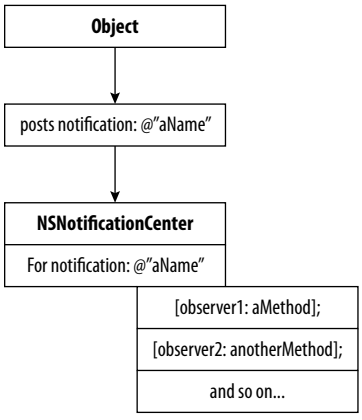
When using Cocoa objects, the key names are given. It's not always easy to remember the data type reference by a key, so it's good practice to note the type in a comment.

Using Notifications

Cocoa's notifications are an application-wide messaging service. Many classes generate notifications as they work. You can set up objects to receive all notifications posted by another object or to respond only to named notifications. Figure 9.12 is a block diagram of the notification system.

Figure 9.12

Understanding notifications. The notification is passed to the common message center, which checks its records for a matching name and triggers the registered selectors for that notification.



Posting a notification triggers the following events:

1. **An object posts a named notification to the default notification center.**
2. **The center checks its records for the name.**
3. **If it finds a matching name, it enumerates each stored entry for the name.**
4. **Each entry includes a target object and a selector.** The notification center triggers the selector in the target object, passing it an `NSNotification` object with a copy of the name and an optional object pointer, which is usually the original sender.

To register an object for notifications, use

```
[[NSNotificationCenter defaultCenter]
addObserver: theObserverObject
   selector: @"TheMethodTriggeredWhenTheNotificationArrives"
       name: @"AUniqueName"
     object: anOptionalObject];
```

If `name` is left blank, the observer responds to all notifications posted by an object.

Some names are predefined. For example, `NSWindow` posts `NSWindowDidBecomeKeyNotification` and `NSWindowWillCloseNotification`, among others. If they exist, you can find the notification names listed at the bottom of a Class Reference.

If a message triggers a notification automatically, you'll also find the notification listed in the method description; for example, sending `update` to a window updates it and posts an `NSWindowDidUpdateNotification` object. This "just happens" — it's built into the method.

Posting notifications

You can create custom notifications with custom names. To post an identification, use

```
[[NSNotificationCenter defaultCenter]
postNotification: @"AName"
     object: anOptionalObject];
```

Often `anOptionalObject` is `self`, giving the notification an explicit return address. You can also use the object field to pass useful information in a dictionary, array, or string, or in a pointer to another object. It's up to the sender and receiver to agree to the contents of the notification and how they'll be used — or more accurately, it's up to you to define the format for both.

An optional extended method supports both an optional object and a dictionary you can pack with other objects.

```
[[NSNotificationCenter defaultCenter]
postNotification: @"AName"
          object: anOptionalObject
     userInfo: aDictionary];
```

To define a method that receives and processes notifications, use

```
-(void) handlerMethod: (NSNotification *) theNotification {
NSString *theName = theNotificaton.name;
id *theObject = theNotification.object;
NSDictionary *theDictionary = theNotification.userInfo;
}
```

Plug this method into the handler selector slot when you register a notification. All data fields are optional. You may not need them, because sometimes it's enough to know that a certain event happened.

Using notifications and delegates

Notifications include a useful shortcut. If an object has a delegate, the delegate automatically observes notifications and can trigger a corresponding delegate method. There's no need to define an observer.

This is a hidden feature. You won't find the corresponding delegate methods listed in a protocol, so you have to reconstruct them using guesswork and insider knowledge. This isn't difficult, because the format is consistent. To find the delegate method for the `NSWindowDidMoveNotification` message, remove the class name from the start and the "Notification" from the end.

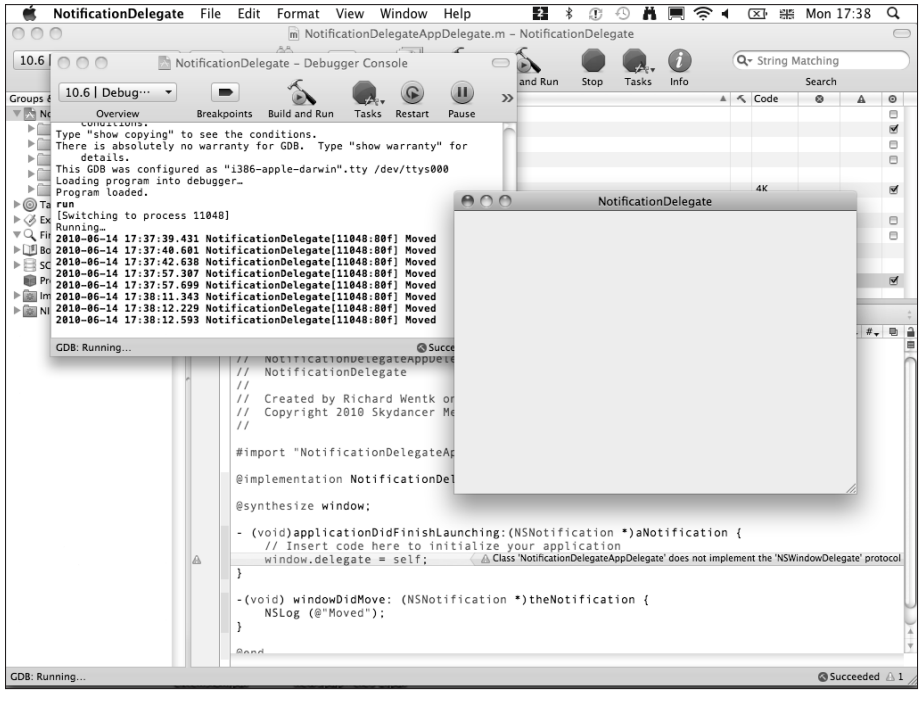
Implement the method so that it receives a notification as a parameter and returns void, like this:

```
-(void) windowDidMove: (NSNotification *) theNotification {
//Code goes here
}
```

Figure 9.13 shows the result when the code is added to the App Delegate in an empty template. The delegate method logs a message when the window is moved. The compiler complains that it can't find a delegate protocol for `NSWindow`, but the code works anyway.

Figure 9.13

Converting a predefined notification into a delegate method. The method is triggered automatically in the delegate, even though there's no formal protocol and no explicit registered observer.



Handling Errors and Exceptions

Many Cocoa objects support an error parameter, which is an instance of `NSError`. `NSError` stores a pointer to an error object, so the syntax is unusual: the error pointer is prefixed with `&`. For example, to write a string to disk with error checking, use the following:

```
NSError *error;
[aString writeToFile: aFilePath
          atomically: YES
            encoding: anEncoding:
               error: &error];
```

Ignore the other parameters for now; they're discussed in Chapter 10 and Chapter 12. In this section, you'll look at `&error`.

The previous code tells the `writeToFile:` method to fill the `&error` pointer with `error` information if the method fails. You can ignore this error trapping and leave `error` set to `nil`. This is occasionally valid. For example, when you read an item from disk, you can check if it's `nil` instead of trapping an explicit error, but including an error check creates more robust code.

Using NSError

The error object itself isn't very informative. You can use

```
[error localizedDescription];  //Returns a description string
[error localizedFailureReason];//An alternative description
```

to return text descriptions of the error suitable for logging. Sometimes these include a cryptic error code, which can only be converted to a useful description of the error by reading through various header files. For egregious errors, the description may be straightforward and useful.

NOTE

Errors are grouped by *domain* — one of Mach, POSIX, or OSStatus — which defines the OS layer responsible for defining the error.

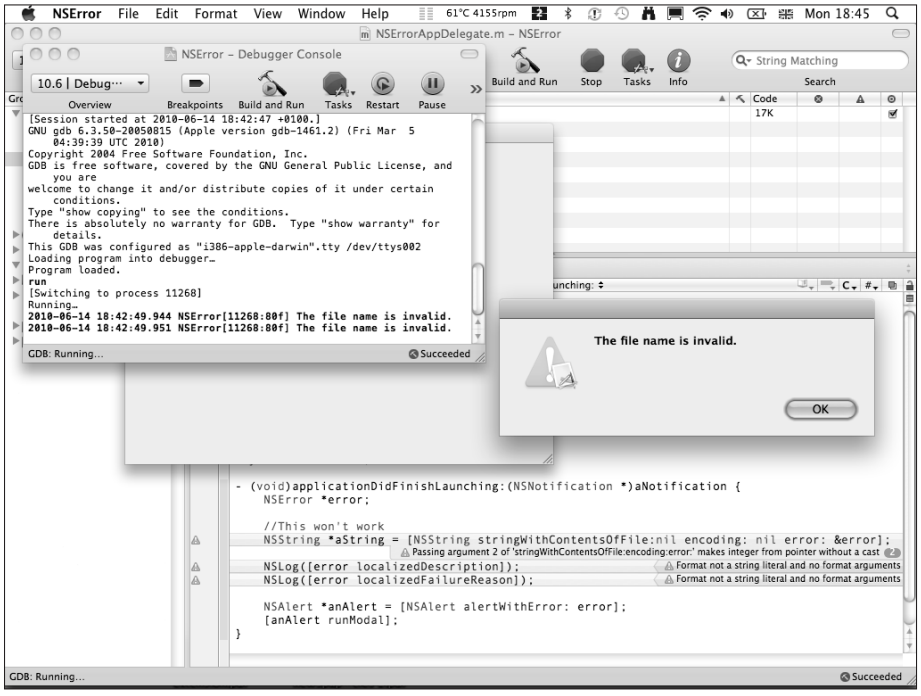
You can convert an error into an `NSAlert` with the following:

```
NSAlert *alert = [NSAlert alertWithError: error];
[alert runModal];
```

This displays the error with the description strings built into a pop-up alert panel, as shown in the example in Figure 9.14, which tries to load a string from a file with a null file path.

Figure 9.14

Converting an error object into an error alert panel.



Handling errors with NSException

If a block of code is error-prone, you can use `@try`, `@catch`, and `@finally` compiler directives to handle errors. Use `@throw` to create an `NSException` that diverts the code to an error handler. For example:

```
@try {
    //{Do useful things}
    if (thereWasAnError) {
    NSException *thisException = [NSException
      exceptionWithName: @"Something bad happened"
                 reason: @"For a reason"
               userInfo: nil];
    @throw thisException;
    }
}
@catch (NSException *exception) {
    if ([exception name] isEqual: @"Something bad happened"]) {
       //Something bad handler
```

```
}
@finally {
   //Clean up, if necessary
}
```

The `@try` block contains code that may create an error, and the `@catch` block handles the error if it occurs. The `@finally` block runs regardless.

You can add multiple conditionals in the `@catch` section to deal with multiple exceptions by name. You can also subclass `NSException` to create custom exception objects, with optional extra features. Replace `NSException` with the name of your subclass in the `@catch` handler.

Optionally, you can add a `@catch` section to catch system-generated exception events. This is a plausible but potentially risky way to create a crash-free application. Even if an application is fatally wounded by an exception, it may still be possible to give the user a chance to save his or her data before quitting.

Summary

In this chapter, you learned more about various Cocoa messaging features and design patterns. You were introduced to a detailed explanation of Model-View-Controller (MVC), with information about how it's implemented in various controller objects in Cocoa.

Next you learned about the target-action design pattern, and you investigated a simple application that used target-action to swap the methods triggered by a button click. You also learned about selectors, and you discovered how to use selectors to trigger a message after a delay.

In the next section, you explored key-value coding, learning about its strengths and some of its limitations. You were introduced to practical key-value observing, and you discovered how to use it effectively.

You learned about notifications, discovering how to post and observe them, and you also learned how to implement notification-based delegate methods without a formal protocol.

Finally, you explored Cocoa's error-handling features, and you discovered how to create error alert panels and how to split your application into sections that can trap and manage exceptions without crashes.

10 Working with Files, URLs, and Web Data

Cocoa's file handling is unusual. In most operating systems, files are managed by calling a file manager object. In Cocoa, data objects can read and write data directly. A separate file manager is only required when you need to delete, copy, rename, or move files under program control.

NOTE

Cocoa's NSFileManager class can be used to access `stdio` — the default system i/o channel. It also supports nonlinear file reads, making it possible to extract bytes from a while without loading it into memory.

For example, `NSString` has methods for reading and writing string data to files; `NSDictionary` can read and write dictionaries; and so on. Typically an `initWithContentsOfFile:` method initializes a data object with the contents of a file specified by a path string that points to a unique location in the file system. A corresponding `writeToFile:` method writes data.

For completely general file access, the `NSCoder` class described in the Chapter 12 read and write complex arbitrary data collections. You can use `NSCoder` to create application-specific files that mirror your application's internal object or document structure, combining data from multiple objects of different classes.

CROSS-REF

For more information about Cocoa's data objects, see Chapter 12.

Perhaps the most unique feature of Cocoa file handling is the close relationship between local and remote data. In most operating systems, local and remote data is handled by different classes.

In Cocoa, URL objects and local file paths are almost synonymous. All data objects have an `initWithContentsOfURL:` method that loads data from a specified URL. The URL can represent a local file path or an online data source. The same class handles both, and it doesn't distinguish between them.

In fact, string file paths are becoming a legacy feature of both OS X and iPhone OS. Strings are not yet deprecated, but there is a trend toward using URL objects as the default path specifier for all file operations.

10 In This Chapter

Creating and using file paths

Creating and using URLs

Using Web APIs

Using WebView

Creating and Using File Paths

In theory, a path string is a standard instance of `NSString`, with the usual Unix format. For example:

```
/Users/Main/Desktop/aFileName
```

In practice, path strings require specific formatting, and their contents must exclude invalid characters. It's possible to create path strings with the general concatenation methods built into `NSString`. But `NSString` also includes a selection of path-specific methods that create valid path strings automatically. Other methods can decompose them into their components — elements that define subdirectories.

Creating paths with NSString

The `NSString` path methods are listed in the Class Reference, and a task is dedicated to them. You can use them to do the following:

- Create absolute and relative paths from a source array of components
- Break an existing path into an array of components
- Append a new path component to an existing path
- Add or remove an extension
- Convert a "tilde path" such as `/~Me` into a full system path. (Tilde paths are used as shortcuts, often to user directories.)

Most path methods are self-explanatory and easy to use:

```
NSString *myDirectory = @"~me";
NSString *myPath = [myDirectory stringByExpandingTildeInPath];
//myPath = @"/Users/me"
```

To find a user's home directory, use the `NSHomeDirectory()` functions:

```
NSString *userHomeDirectory = NSHomeDirectory();
NSString *myHomeDirectory = NSHomeDirectoryForUser(@"me");
```

Getting the application bundle path

When you distribute an application, it's often useful to include images, sounds, and other data within the application folder. The *bundle*, as it's known, has a specific path. You must use this path to load the files. The standard boilerplate for the path to `aFile.ext` is:

```
NSString *thePath =
[[NSBundle mainBundle] pathForResource: @"aFile" ofType@"ext"];
```

Optionally you can add `inDirectory:subPath` to specify a subpath. To find an array of paths to multiple files with the same extension, use:

```
NSArray *theArrayOfPaths =
[[NSBundle mainBundle] pathForResourcesOfType@"ext"
   inDirectory:subPath];
```

Finding other standard directories

To find a standard system directory such as `/Documents` use:

```
NSString *standardDirectory;
NSArray *paths = NSSearchPathForDirectoriesInDomains
                    (NSDocumentDirectory, NSUserDomainMask, YES);
if ([paths count] > 0)  {
    standardDirectory = [paths objectAtIndex:0];
}
```

In OS X, you can replace `NSDocumentDirectory` with one of the other `NSSearchPathDirectory` constants. You can find the full set of definitions in the Foundation Data Types Reference. For example:

- `NSApplicationDirectory` — the application's root directory
- `NSDesktopDirectory` — the desktop
- `NSDownloadsDirectory` — the default download location

This technique is more reliable than starting with the root system path and appending `/Documents` to it. Standard directories may not be where you expect them to be, and this approach is more likely to return a valid path to them.

Using autocompletion

Cocoa includes an autocompletion feature that scans a target directory for a list of filenames that match a template. This is independent of the file search implemented in Spotlight.

```
NSString *testPath = @"usr/somepath/a";
NSString *longestCompletion;
NSArray *outputArray;

unsigned allMatches =
   [testPath completePathIntoString:&longestCompletion
       caseSensitive:NO
    matchesIntoArray:&outputArray
         filterTypes:NULL];
```

`allMatches` returns a count of matching files that start with "a," and `outputArray` returns an array of full paths to each matching file. For a practical example, set `testPath` to a valid directory path and append the search string to find matching files.

To search for files with specific extensions, load `filterTypes` with an array containing the extensions before running the search:

```
NSArray *filterTypes =
[NSArray arrayWithObjects: @"jpg", @"tiff", nil];
```

Typically you use `testPath` to display a list of possible matches, and then allow the user to select one or more.

Using paths

Once you have a path string, Cocoa's data objects can load data from the file it points to with standard `<object>WithContentsOfFile:` and `initWithContentsOfFile:` methods.

For example, to load a string from a text file at `aPath` use:

```
NSString *textInTheFile =
[NSString stringWithContentsOfFile: aPath
encoding: anEncoding: error: &error];
```

The `encoding` parameter defines the *string encoding*, the mapping between characters and byte values. `NSUTF8StringEncoding` and `NSASCIIStringEncoding` are two standard values. For more information, see Chapter 15. `&error` is a standard system error handler and is introduced in Chapter 9.

`NSArray`, `NSDictionary`, `NSSet`, `NSData`, and `NSImage` have equivalent methods.

The corresponding write operation is

```
[aString writeToFile: aPath
            encoding:anEncoding
          atomically: YES];
```

Again, Cocoa's other data objects have corresponding methods.

The `atomically` parameter is a `BOOL`. When it is `YES`, the file is written in two stages to guarantee that it is closed and readable. When it is `NO`, failed file writes can leave uncompleted file fragments on disk; there isn't any good reason to specify `NO`.

Using file handles

Cocoa's object-oriented file system simplifies object-oriented reads and writes, but complicates byte-level file access. To read and write binary files with pointer control, use `NSFileHandle`. For example:

```
NSFileHandle *aReadHandle =
[NSFileHandle fileHandleForReadingAtPath: aPath];
NSData *someData =
[aReadHandle readDataOfLength: anInteger];
//Read more data or change the seek pointer here, as needed
[aReadHandle closeFile];
```

To move the seek pointer, use

```
[aReadHandle seekToFileOffset: aLongInt];
```

A corresponding `writeData:` method writes data to the file. Use the `synchronizeFile` method to flush caches and buffers and to guarantee that the file contents are updated correctly.

TIP

File handles can also be used for data transfers. `NSFileHandle` supports asynchronous communication with delegate methods that post notifications as data is sent or transmitted. For details, see the Class Reference.

Using the File Manager

You can use a File Manager object to copy, move, and delete files and to list directory contents. Most applications don't need to implement a File Manager. Users can use Finder to work with individual files, so these features are only useful in applications that batch file operations, such as a backup/restore application, or when an application needs to delete temporary files while closing. File open/save operations can be performed with standard file access panes, and directory lists are only useful when creating a customized equivalent — or when performing batch operations that scan the directory structure.

However, you can use a File Manager to compare two files or to change the application's current directory.

The File Manager class is easy to work with. To create a File Manager object, use

```
NSFileManager *thisManager = [NSFileManager defaultManager];
```

This returns the default system file manager, which is the only manager your application should use.

To compare two files, use

```
[thisManager contentsEqualAtPath: aPath andPath: anotherPath];
```

This returns a `BOOL` you can test for equality. Copy/Move operations are similarly straightforward:

```
[thisManager copyItemAtPath: aPath
toPath: anotherPath error:&error];
```

To change the current application directory, use

```
[thisManager changeCurrentDirectoryPath: aNewPath];
```

Creating and Using URLs

Although URLs are partly interchangeable with string paths, they are more complex. They also support additional features. Some objects, such as the `NSOpenPane` and `NSSavePane` objects used to present the user with a file selector, return URL paths automatically. More typically you create a file access URL from a string path; for example:

```
NSURL *thisURL = [NSURL fileURLWithPath: aPath];
```

You can also extract the path string from a URL:

```
NSString *thePath = [thisURL path];
```

CAUTION

NSURL's `string` method returns a string in a URL-specific format. This is neither a valid file path nor a valid Web URL. If you log it, you'll see that it looks nothing like a path. Also, it's best not to use the `URLWithString:` and `initWithString:` methods for file paths — they may not parse a path string correctly.

Understanding paths and references

`NSURL` supports two kinds of path objects: a file path and a file reference. When you create a file URL you don't need to specify the type, because the file system checks the type at runtime and parses the URL data accordingly. But it's useful to be aware of the distinction. You can convert a reference into a path by running `filePathURL:` on it. Use `fileReferenceURL:` to convert a path into a reference. `isFileReferenceURL` and `isFileURL` are `BOOL` methods that return the type.

Using URLs to read and write data

Most objects offer methods that correspond to those used for file path reads and writes. For example, to load a string, use

```
NSString *textInTheFile =
[NSString stringWithContentsOfURL: aURL
encoding: anEncoding: error: &error];
```

But unlike a file path, the URL can be remote:

```
NSURL *aURL =
[NSURLWithString: @"http://www.asite.com/file.txt"];
```

If you know the address of a file on a remote server, you can use this feature to download it, as discussed later in this chapter in the section on Web APIs.

Using Open and Save Panes

Cocoa includes two classes that enable a user to select a file visually and return a URL while opening and saving files. `NSOpenPane` creates the Finder-like window shown in Figure 10.1. `NSSavePane` creates the simpler window shown in Figure 10.2, although it can be expanded to the larger pane shown in Figure 10.3.

Figure 10.1

The standard `NSOpenPane`. The pane slides down from the top of the application window. This animation is automatic — it's built into the class and can't be changed.

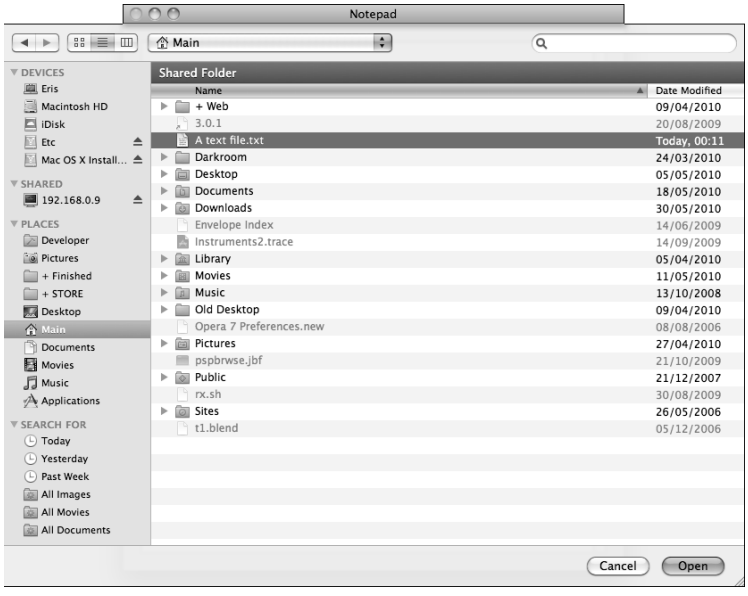


Figure 10.2

The standard small `NSSavePane`. You can use this pane to type in a filename and save it to the current directory.

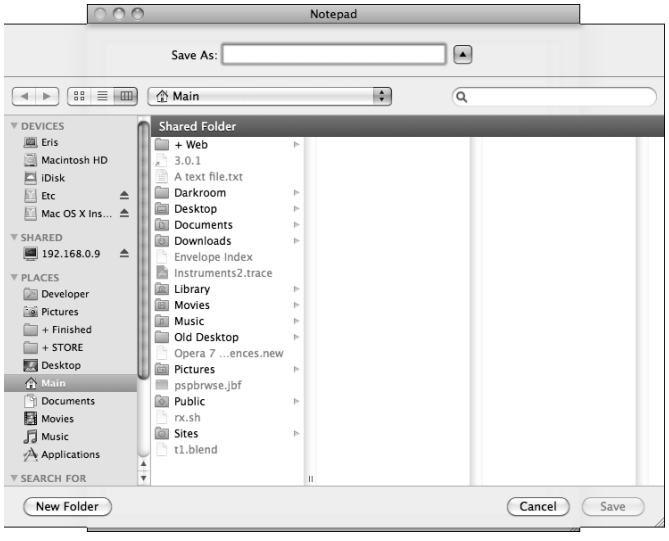
Typically these classes wrap around the code that loads and saves data, using either the features built into Cocoa's data collection classes or a custom `NSCoder` implementation. You can use these classes to:

- Support a configurable window title, optional text message, return button prompt, and other user-friendly labeling features.
- Preselect one or more file types when opening files. Other file types are grayed out and cannot be selected or opened.
- Allow multiple selections while opening.
- Automatically append a file extension while saving.
- Preselect a default directory.
- Enable or disable directory switching.

- Show or hide hidden files.
- Enable either document-modal or application-modal display.
- Refresh and revalidate the file view dynamically.
- Return a Cancel or OK status.
- Return a file path in a variety of formats. (From OS X 10.6 onward, URLs are preferred.)

Figure 10.3

Clicking the downward-pointing triangle next to the file name box reveals `NSSavePane`'s larger view, with full directory access.



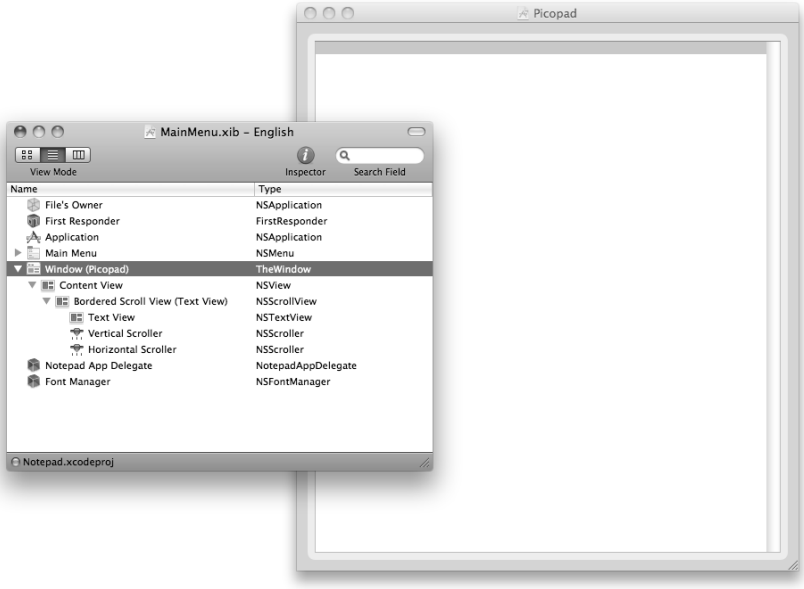
The open and save panes and Cocoa's object-oriented data features make it possible to create very minimal applications, such as the Picopad application available on the Web site. Picopad implements a very simple text editor with file save/load. The editor stores text in an `NSTextView`, which is held inside an `NSScrollView`, as shown in Figure 10.4.

NOTE

You can download the Picopad application at `www.wiley.com/go/cocoadevref`.

Figure 10.4

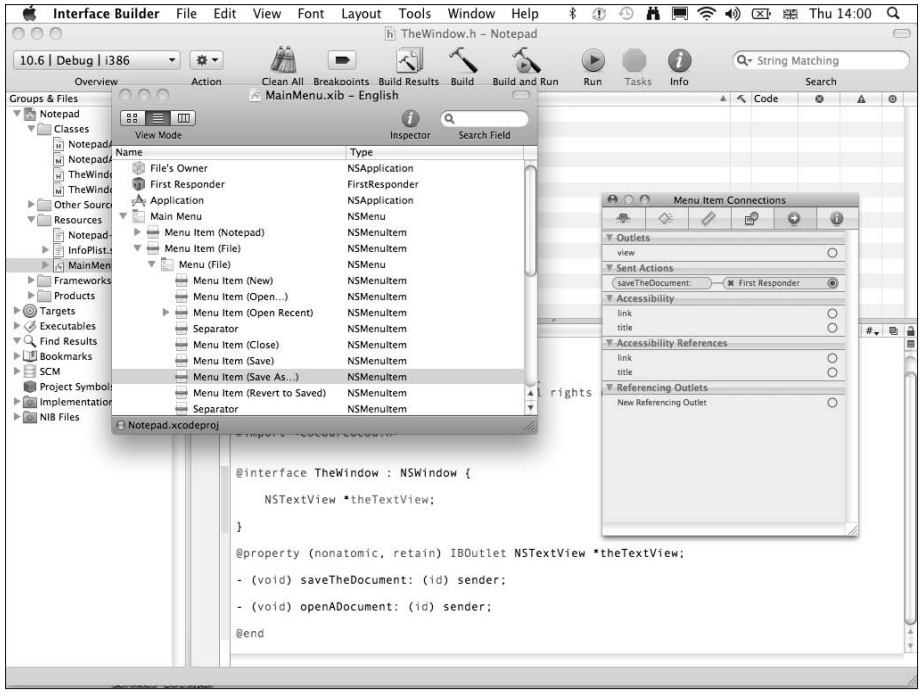
The nib file for Picopad. `NSScrollView` includes an instance of `NSTextView`. In this example, you'll ignore the scroll bars and link directly to the text view; you don't need to access the scroll view's properties.



`NSWindow` is subclassed as `TheWindow`, and it includes a pair of open/save methods, `openADocument:` and `saveTheDocument:`, that are linked to the Open and Save As menu items. `TheWindow` also includes an outlet to the text view, so that the application can read and write its text. The header and one of the links is shown in Figure 10.5.

Figure 10.5

The header file for `TheWindow` includes an outlet to the text view and a pair of methods for opening and saving a file. One of the methods is shown linked to its corresponding menu item. The other item, not shown here, is also linked.



The save and open methods are easy to implement. From OS X 10.6 onward, `NSOpenPane` and `NSSavePane` use inline blocks as return handlers instead of delegate methods. The complete code for `TheWindow` follows:

```
@implementation TheWindow
@synthesize theTextView;
-(void) saveTheDocument: (id) sender {
 //Create a panel
 NSSavePanel *savePanel = [NSSavePanel savePanel];
 savePanel.allowedFileTypes = [NSArray arrayWithObject:@"txt"];
 //Display it, and append the handler block
 [savePanel beginSheetModalForWindow:self
   completionHandler:^(NSInteger result) {
```

```
//This is the handler block
 if (result == NSFileHandlingPanelOKButton) {
  //Save the file on OK, do nothing on Cancel
   [theTextView.string writeToURL: savePanel.URL atomically: YES
   encoding: NSUTF8StringEncoding error:nil];
 }
}]; //This is the end of the handler
}
- (void) openADocument: (id) sender {
  NSOpenPanel *openPanel = [NSOpenPanel openPanel];
  openPanel.allowedFileTypes = [NSArray arrayWithObject:@"txt"];
  [openPanel beginSheetModalForWindow:self
   completionHandler:^(NSInteger result) {

  if (result == NSFileHandlingPanelOKButton) {
   //If OK, load the file
   NSString *theText = [NSString  stringWithContentsOfURL:openPa
   nel.URL encoding: NSUTF8StringEncoding error:nil];
  [theTextView setString: theText];
  }
 }];
}
```

Key points of the code:

1. The first line creates a save panel and initializes it with an array that preselects `.txt` files.
2. The second line displays the sheet. It runs modally, locking out the rest of the application.
3. The result handler is inside a block. It checks the return button status and does nothing if the user cancelled. Otherwise, it writes the data to a file as a string, at the URL returned from the pane.
4. The open method is almost identical, but it loads the data from the selected file into the text view.

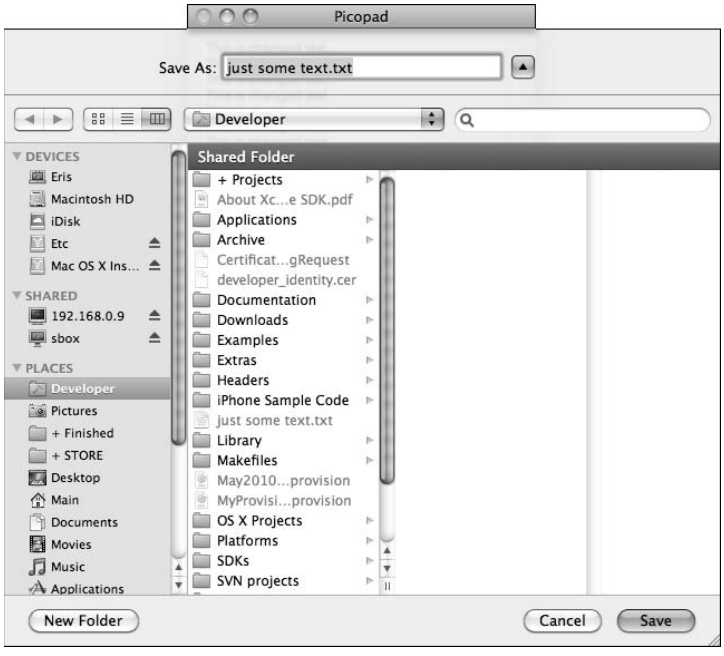
You can customize the open/save panels further by setting other optional properties. This minimal solution implements a full open/save application with editing, copy/paste, and file save and load — all with a few lines of code. The finished application is shown in Figure 10.6.

CAUTION

If your Mac is on a network and you are connected to any remote servers, the open and save panes will scan the network and ping the servers. This isn't usually a problem, but if you are using a network monitor like Little Snitch, don't be surprised when your minimal application triggers a network access warning.

Figure 10.6

Saving a file with Picopad. If you look *very* closely, you may be able to see a very blurry and faint view of the text in the text field, behind the Save As label at the top of the pane. Like all Aqua elements, the pane is slightly translucent and it blurs whatever is behind it.



Using Web APIs

One of the key applications of `NSURL` is Web API access. Google, Amazon, Yahoo, Twitter, Facebook, and other popular Web services offer APIs that developers can use to implement advanced features. For example, you can use Google's new Location API to return a list of businesses around a given location.

`NSURL` includes features that make it possible to connect to any Web API. Typically your application follows four steps:

1. It creates a URL in the very specific format required by the API.
2. It uses an `NSURL` method to send the URL to the server.
3. It downloads the response.
4. It extracts data from the response.

TIP

Although Web APIs are accessed through standard http: URLs, APIs serve data, not Web pages. But you can use `NSURL` to download HTML pages from any site; for example, to spider it, scan it for links, or download a selection of pages for offline review.

In practice, your application must do four things to use any API:

1. Set up access credentials and security.
2. Initialize the URL request correctly.
3. Select either a synchronous or asynchronous download method for the return, and capture the data correctly.
4. Parse the data to extract useful information from it.

Getting started with bit.ly

As an example, I've used the bit.ly URL shortening service, often used on Twitter to create terse URLs. It offers a selection of different API options, from simple text access to XML data. It also implements security and supports both synchronous and asynchronous downloads, as I'll explain below.

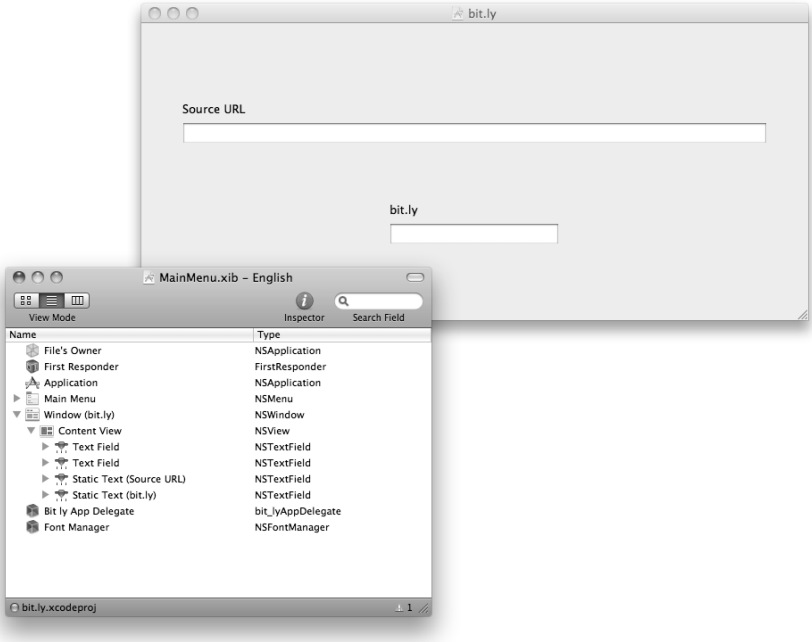
As a first step, you'll create a new application — call it bit.ly — and edit the nib to create the view shown in Figure 10.7. It includes two active text fields, that need outlets, and two static descriptive labels.

To create outlets and define a method that can be triggered to shorten a URL, modify the app delegate header as follows, and link the outlets to their corresponding objects in Interface Builder (IB):

```
@interface bit_lyAppDelegate : NSObject <NSApplicationDelegate,
   NSTextFieldDelegate> {
    NSWindow *window;
    NSTextField *sourceURL;
    NSTextField *bitlyURL;
}
@property (assign) IBOutlet NSWindow *window;
@property (assign) IBOutlet NSTextField *sourceURL;
@property (assign) IBOutlet NSTextField *bitlyURL;
-(void) toBitly;
@end
```

Figure 10.7

Creating the initial bit.ly nib. The source URL text field should be much wider than the bit.ly return field, for reasons that may be obvious.



Getting text from a text field

Note that I've adopted the `NSTextFieldDelegate` protocol. The text field doesn't trigger a selector. Instead it calls a delegate method in the protocol:

```
-(void) controlTextDidEndEditing:
(NSNotification *)aNotification;
```

The protocol supports other delegate methods, but this method is the critical one in this application. It's triggered when the user presses the Return key after editing the text. You can implement the method to read the new text and process it; in this case, passing it to the bit.ly API to shorten it.

To use the protocol, set the delegate of `sourceURL` to `self` when the application launches, as shown below, and implement the `didEndEditing:` method, which you'll do in detail later. You'll also set an initial prompt string to save typing it by hand for every URL.

```
- (void)applicationDidFinishLaunching:
```

```
 (NSNotification *)aNotification {
 sourceURL.delegate = self;
 sourceURL.stringValue = @"http://";
}
```

To access the text in a text field, use the `stringValue` property. (You might expect this to be called `text`, but it isn't.)

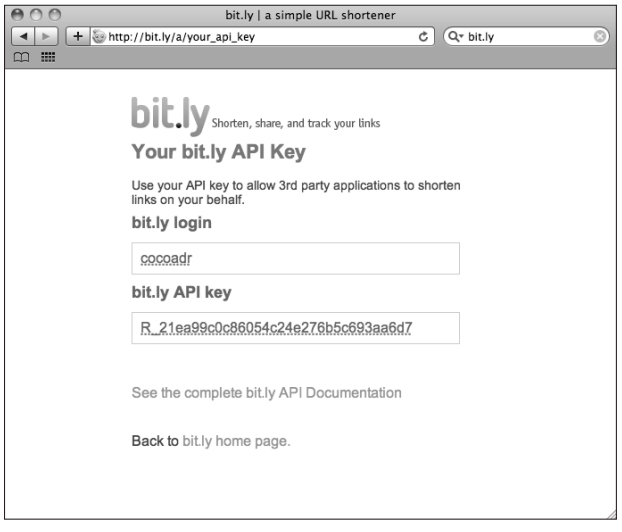
Getting a bit.ly key

Some extra setup is needed first. Most APIs use an access key or password system. The key is embedded in every request URL as plain text. A few APIs use more secure credentialed session systems. Credentialing systems can become very complex, and they are outside the scope of this book. For an example, see the oauth system at `http://oauth.net`, which is used by Twitter.

The bit.ly service uses the simpler key system. Before going further, visit the bit.ly site (`http://bit.ly/`) and sign up to get your key string, as shown in Figure 10.8.

Figure 10.8

Getting a bit.ly access key and username. Do *not* use this key in your own projects. Getting a key is easy, and it's free. If you use this key to test high-performance batched access, it will stop working. This will inconvenience other readers and may result in an unwelcome midnight visit from the author.

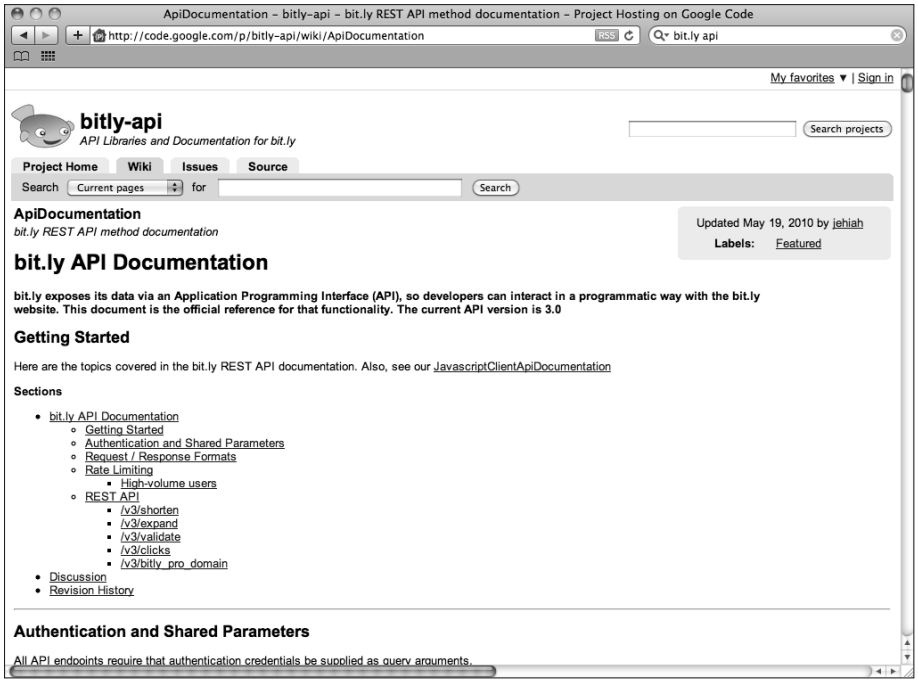


Using the bit.ly API

Before you can use the bit.ly API, you need to find some documentation for it. You can find details of most public APIs with a Google search. You can find the bit.ly API documentation at `http://code.google.com/p/bitly-api/wiki/ApiDocumentation`, as shown in Figure 10.9.

Figure 10.9

Finding the bit.ly API reference documentation. Some APIs are public and open source, so the documentation isn't necessarily available on the main Web site.



The documentation suggests that there are a number of API features: you can shorten a URL, you can find out how many times it has been clicked, you can expand a short URL into a long URL, and so on.

TIP

Documentation quality is very variable. Some APIs have very clear documentation with examples; others are opaque. Examples are usually helpful; if you can't find them in the documentation, try searching for them elsewhere online.

You want to shorten URLs, so you need the shorten feature. Clicking the /v3/shorten link takes you to a page of definitions and examples of different formats. For this example, you want the text format shown highlighted in Figure 10.10.

Figure 10.10

Getting the URL requirements. APIs that support URL access always list the required URL, with full details of each subfield.

To use the API successfully, you must send requests with *exactly* this format:

```
http://api.bit.ly/v3/shorten?login=<name>&apiKey=<key>&longUrl=<e
    scapedURL>&format=txt
```

If you do this correctly, the bit.ly server returns a standard shortened URL string:

```
http://bit.ly<short six character URL>
```

Creating a long URL

You can glue together this long URL from components. For example, the login name and key can be defined at the start of the file as a preamble string constant, and the `&format=text`

string can be stored in a separate string constant. To combine part strings into a complete string, you can use the `[NSString stringWithFormat: ]` method to concatenate a series of input strings. The exact code appears later in the complete code listing.

The long URL is more of a challenge. Looking at the format, you can see that it uses *escaped characters*. A space is converted into %20, a colon into %3A, and so on. This is a standard requirement for many APIs, because it eliminates ambiguity and guarantees that the server can read the URL correctly.

Cocoa — or rather, Core Foundation — includes a function that converts that string into its escaped equivalent. The function is slightly tricky to set up, but the following code works as a boilerplate method that you can drop into any code to convert a string into an escaped string:

```
- (NSString *)urlEncodeValue:(NSString *)str
{
NSString *escapedURL = (NSString *)
CFURLCreateStringByAddingPercentEscapes(kCFAllocatorDefault,
(CFStringRef)str, NULL, CFSTR("~:/?#[]@!$&â€™()*+,;=\"") ,
kCFStringEncodingUTF8);
return escapedURL;
}
```

The characters after `CFSTR` are automatically replaced with their escaped equivalents in the return string. You can add more characters to this list if you need to, but this standard selection handles most requirements.

Finally, after the string processing, you need to convert the finished long URL string into a URL object. This is easy: you can use the `[NSURL urlWithString: ]` method to return the equivalent URL object.

Sending the URL request and processing the return

There are various ways to send a URL request. At this point, Apple's NSURL documentation can be more confusing than helpful. It suggests using various auxiliary URL connection and response objects.

In fact, there's a single line solution. For a simple Web transaction, the easiest option is to use the `stringWithContentsOfURL:` method built into `NSString`. It returns a string object:

```
theReturnString = [NSString stringWithContentsOfURL: aURL
   encoding: NSUTF8Encoding error: nil];
```

This single line of code is extremely powerful. It sends the URL request to a remote server, waits for a response, and downloads the return data into a waiting string object. If the request fails, the return string is `nil`. But if you set up your URL request correctly, this line of code is all you need to retrieve a shortened URL. The finished code shown here copies the return to the return field in the view — and you're done.

```
@synthesize window, sourceURL, bitlyURL;
NSURL *thisURL;
```

```
NSURLRequest *thisRequest;
NSString *returnURL;
NSString *escapedURLString;
NSString *bitlyPreambleString = @"http://api.bit.ly/v3/shorten?lo
   gin=cocoadr&apiKey=R_21ea99c0c86054c24e276b5c693aa6d7&uri=";
NSString *bitlyPostambleString = @"&format=txt";
NSString *bitlyURLString;
- (void)applicationDidFinishLaunching:
        (NSNotification *)aNotification {
  sourceURL.delegate = self;
  sourceURL.stringValue = @"http://";
}
- (void)controlTextDidEndEditing:(NSNotification *)aNotification
{
  [self toBitly];
}
-(void) toBitly{
  escapedURLString =
  [self urlEncodeValue: sourceURL.stringValue];

 thisURL = [NSURL URLWithString:
   [NSString stringWithFormat:@"%@%@%@",
   bitlyPreambleString,
   escapedURLString,
   bitlyPostambleString]];

returnURL = [NSString stringWithContentsOfURL:thisURL
            encoding: NSUTF8StringEncoding error: nil];
if (returnURL != nil)
    bitlyURL.stringValue = returnURL;
else
    bitlyURL.stringValue = @"Error";
}
- (NSString *)urlEncodeValue:(NSString *)str
{
 NSString *escapedURL = (NSString *)
   CFURLCreateStringByAddingPercentEscapes(kCFAllocatorDefault,
   (CFStringRef)str, NULL, CFSTR("~:/?#[]@!$&â€™()*+,;=\"") ,
   kCFStringEncodingUTF8);
 return escapedURL;
}
@end
```

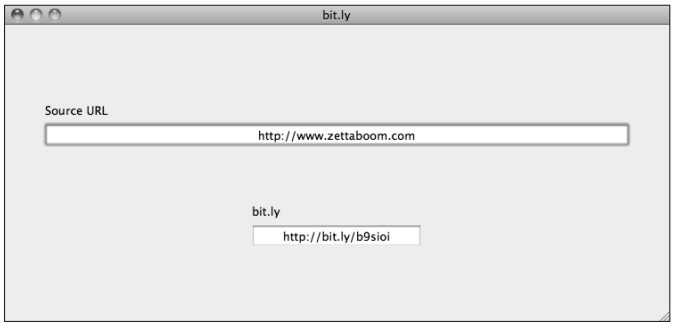
NOTE

In this example you've created a `toBitly` method that creates and sends the request and reads the return. This method is called from the delegate return when the user presses the Return key. The split isn't necessary, but it aids modularity; you might want to use the same method elsewhere.

The finished application is shown in Figure 10.11. To use it, type in a URL and press Return. The shortened bit.ly address appears in the lower text field, or it reports an error if there was a problem.

Figure 10.11

Getting the URL requirements. APIs that support URL access always list the required URL explicitly.



Creating XML requests

Most Web APIs return data in one of two industry-standard formats: JSON (JavaScript Object Notation) and XML. Both formats are text-based (at least, at present) but include special formatting that defines fields and subfields.

JSON is a more recent and more robust format than XML. XML can suffer from white space issues, and parsers may not extract data from an XML formatted string correctly if there are minor errors.

Unfortunately, Cocoa doesn't include native support for JSON. It does include native support for XML, with a selection of XML management classes. But native support isn't always necessary. It's possible to extract data from both JSON and XML API returns with simple string processing. The process isn't elegant, but it is a lightweight way to extract data from one or two specific fields when the surrounding tags are known.

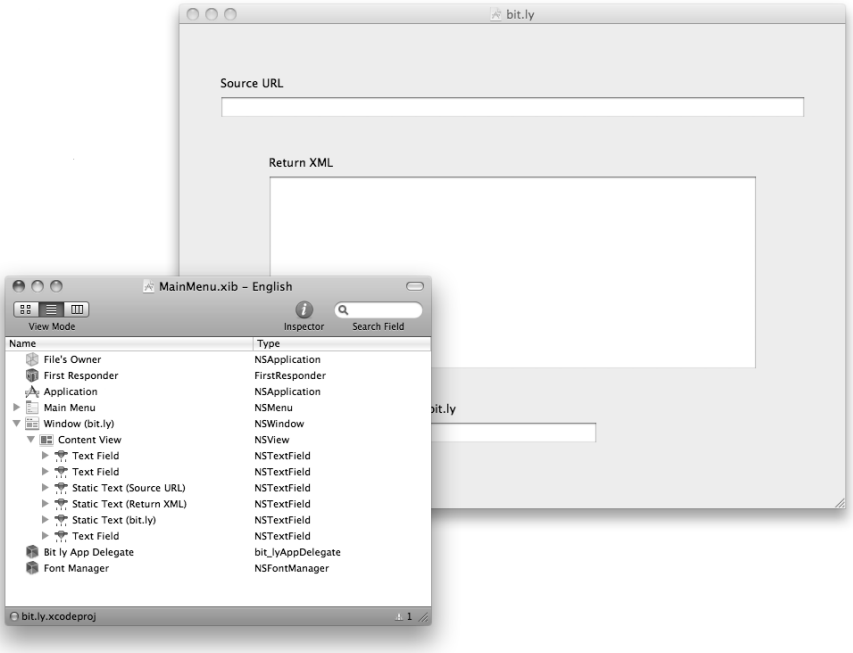
TIP

Developing a full JSON parser is a major project — and it's already been done. You can add JSON support to Cocoa by adding a third-party JSON framework. One of the most popular is json-framework, which is hosted at `http://code.google.com/p/json-framework`.

Figure 10.12 shows an extended version of the simple bit.ly project. Added elements include a single extra text field, an associated text field outlet called `returnXML` in the app delegate header file, and a link between the two.

Figure 10.12

Adding an extra text field to the project, so that you can view the raw XML return



Selecting the XML format

The only other required change is modifying the `&format=txt` defined in the postamble string to `&format=xml`. This tells the bit.ly server that your application wants a packet of XML data instead of a simple text string. The XML will need to be processed to extract the shortened link from it. But you can copy the return string to `returnXML.stringvalue` to see the raw XML returned from the server. Building and running the project should produce the result shown in Figure 10.13.

Why go to the trouble of downloading this extra data? As mentioned earlier, many APIs don't support text returns, so you're often forced to use XML. But as you can see from the return, it includes extra data that isn't provided by the simple text API. For example, the `new_hash` field tells you whether you've shortened this URL before. You can also read a separate status code and status text to tell you more about error conditions if the return fails.

Figure 10.13

The raw XML return. You can see the shortened URL buried inside it, within the `url` tags.

You don't always need this data, but XML and JSON returns always offer more data and, hence, more flexibility. Sometimes they also offer multiple returns; for example, some of the Google APIs return multiple locations when queried with an address or location search string.

It's possible to create generalized XML and JSON parsers that can extract any named value from a return string. A general parser can turn into an advanced project, and it isn't usually necessary. More typically, you'll read one or two values from the return, and assume that its format is fixed. In the case of the bit.ly API, the XML format is standardized. Every valid return formats data in the same way, using the same sequence of elements and subelements.

Parsing XML returns as text

If you don't want to use Cocoa's XML frameworks, you can parse this data as text. Text parsing is less general, but it uses fewer resources. The XML frameworks aren't memory-efficient. Text parsing is a good choice on the iPhone, where memory must be managed carefully.

TIP

Text parsing also makes it possible to extract data from plain HTML. You can use text parsing to summarize or process plain HTML from any Web site.

A simple text parser is listed below. It takes the same return string as an input and extracts data from two fields: `status_txt` and `url`.

```
if (returnURLString != nil) {
  returnXML.stringValue = returnURLString;
  statusRange = [returnURLString rangeOfString: @"<status_txt>"];
  thisRange.location = statusRange.location+12;
  thisRange.length = 2;
  thisString = [returnURLString substringWithRange:thisRange];
  if ([thisString isEqualToString: @"OK"]) {
    NSLog(@"Status OK");
    urlStartRange = [returnURLString rangeOfString: @"<url>"];
    urlEndRange = [returnURLString rangeOfString: @"</url>"];
    thisRange.location = urlStartRange.location+5;
    thisRange.length = urlEndRange.location -   urlStartRange.
   location - 5;
    bitlyURL.stringValue = [returnURLString
   substringWithRange:thisRange];
}
else
 bitlyURL.stringValue = @"Error";
```

This code uses the `rangeOfString:` method to find the range — index and length — of a target string. First, the code looks for `<status_txt>` to find the starting index of the status string. It copies exactly two characters from that index into a new string using the `substringWithRange:` method and checks whether this short new string is equal to `OK`.

If it is, it repeats the same sequence looking for the substring between the `<url>` and `</url>` tags. This code can extract strings of any length between any two tags. In this example, the length of the shortened URL string is fixed, so the code could be simplified slightly. But in this form, it's a workable general solution for extracting the contents of any tag from any XML return.

NOTE

The complete project is available on the Web site (`www.wiley.com/go/cocoadevref`), as is the more complex project discussed next.

Creating asynchronous Web requests

The previous example used the same `stringWithContentsOfURL:` method to send an API request and read its return. The code is simple, but it stalls your application while it waits for the return to complete, or fail. This is called a *synchronous request*.

The more sophisticated alternative is called an *asynchronous request*. An asynchronous request creates a request object, with associated delegate methods. It then triggers the object to begin the request.

Instead of waiting for the request to complete, the application can continue to run. As data is returned from the server, it's collected by the delegate methods and appended to an instance of `NSData`. When the request completes, it calls a method called `connectionDidFinishLoading:`, which can then process the return.

Asynchronous methods can seem complex, but they can be implemented with standard boilerplate code. One possible example follows:

```
-(void) toBitly: (id) sender {
 escapedURLString = [self urlEncodeValue: sourceURL.stringValue];
 thisURL = [NSURL URLWithString: [NSString
   stringWithFormat:@"%@%@%@", bitlyPreambleString,
   escapedURLString, bitlyPostambleString]];
 //Create the request object
 NSURLRequest *theRequest = [NSURLRequest requestWithURL:thisURL
   cachePolicy: NSURLRequestReloadIgnoringLocalCacheData
   timeoutInterval: 30];
 //Start the connection
 [theSpinner startAnimation:self];
 NSURLConnection *theConnection = [[NSURLConnection alloc]
   initWithRequest:theRequest delegate:self];
 if(theConnection) {
   //Connection initialized correctly
   theData = [[NSMutableData data] retain];
  }
 else {
 NSLog(@"Connection didn't initialize");
 return;
 }
}
- (void) connection: (NSURLConnection *)connection
   didReceiveResponse: (NSURLResponse *)response {
   //Reset the data after a response - use in case of redirects
  [theData setLength: 0];
}

- (void) connection: (NSURLConnection *)connection
   didReceiveData: (NSData *)data {
   //Some data arrived - append it
  [theData appendData:data];
}
- (void) connection: (NSURLConnection *)connection
   didFailWithError: (NSError *)error {
  [theSpinner stopAnimation:self];
  NSLog(@"Connection failed");
  [theData release];
  return;
}
```

```
-(void) connectionDidFinishLoading:(NSURLConnection *)connection
{
//Process the data here
}
```

Key features of the code include:

- The connection requires two objects: a URL request and a URL connection.
- The `NSURLRequest` object defines the properties of the connection, including the URL and the timeout interval. If the request doesn't complete before the end of the timeout interval, the request fails.
- The request object also defines the caching policy. Here, the cache is set to reload at every attempt because caching API returns can be risky, especially if the remote data is transient.
- The request is passed to an `NSURLConnection` object, which initiates the connection. It specifies a delegate object for the connection, which monitors the connection status and receives data as it arrives.
- Four delegate methods manage the connection.
- `connection:didReceiveResponse:` is triggered by an active new connection, and it resets the data receiver object.
- `connection:didReceiveData:` appends data to the data object as it arrives.
- `connection:didFailWithError:` reports an error code.
- `connectionDidFinishLoading:` is triggered after the data has been received successfully.

Although this is byte-level Web access — you can use `NSData` in a general way to receive data of any format — you can use this code to manage almost any asynchronous Web request.

Using Cocoa's XML classes

Cocoa's XML implementation is complex, partly because XML can be complex. The core class is `NSXMLDocument`, which can be loaded using the `initWithData:` method. While it's possible to extract data into supporting data types such `NSXMLElement` and `NSXMLNode`, this isn't necessary for a simple task, such as retrieving the text between two tags. It's possible to scan the XML document directly using a query framework called XPath, which is built into `NSXMLDocument`.

XPath is a standard Web technology. A JavaScript version is built into all standard Web browsers. An XPath search assumes that the data in an XML file is formatted in a path-like structure. Queries pass a path to the document, and the search returns the data at that path.

CAUTION

Neither `NSXMLDocument` nor `nodeForXPath:` are available on the iPhone. Use `NSXMLParser` — a dynamic parser class that takes an XML string and triggers various delegate methods as it scans it — instead. `NSXMLParser` is more lightweight than `NSXMLDocument`, but less convenient. To extract specific elements, you must add explicit switches or conditional tests to the delegate methods.

Sample code to extract a text string from the `url` tag follows:

```
theDocument = [[NSXMLDocument alloc] initWithData:theData
   options:NSXMLDocumentTidyXML error:&theError];

if (theDocument) {
   //We have a valid document
   NSLog(@"The XML doc: \r%@", theDocument);
   //Get the text string within the url tags
   NSArray *someNodes = [theDocument
   nodesForXPath:@"/response/data/url/text()"
   error:&theError];
   //This is the string we're looking for
   bitlyURL.stringValue = [someNodes objectAtIndex:0];
   } else {
       bitlyURL.stringValue = @"Doc error";
      return;
}
```

Key features of the code include:

- The document is created with the `NSXMLDocumentTidyXML` option, which tidies the document format and can eliminate minor errors.
- The `nodesForXPath` array returns the elements at the search path.
- Accessing a `text()` element returns a text object that can be read from the array. No further processing or extraction is required.
- The project code includes a spinner activity indicator, a very simple but effective addition that illustrates the download state.

TIP

To find the path you need, review the order of the indented tags in the raw XML. XPath can perform many tricks, including reading multiple returns with the same tag. For more information, see the XPath tutorial at `www.w3schools.com/xpath/xpath_examples.asp`.

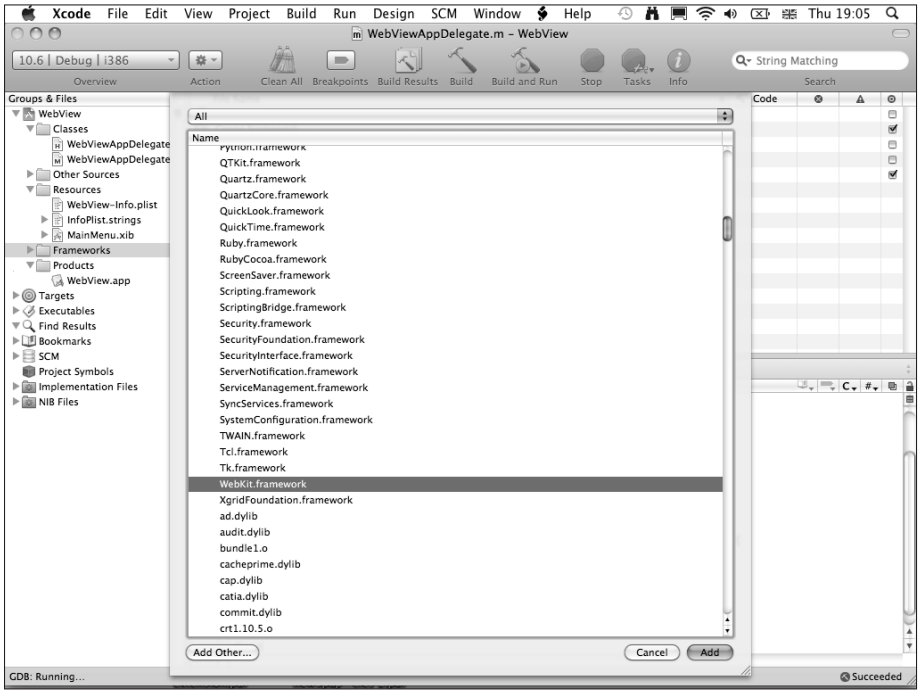
Using WebView

No discussion of URLs would be complete without an introduction to `WebView`, Cocoa's Web browser. You can use `WebView` to add a browser window to any application, with full control over the URL, the size of the window — scroll bars appear automatically, as needed — and standard browser features such as forward and back buttons.

Create a new project, and save it as WebView. Right-click Frameworks, select WebKit.framework from the list that appears, and click Add, as shown in Figure 10.14.

Figure 10.14

Adding the WebKit framework before creating a WebView. WebKit includes a very rich set of features for Web management within Objective-C and JavaScript apps.



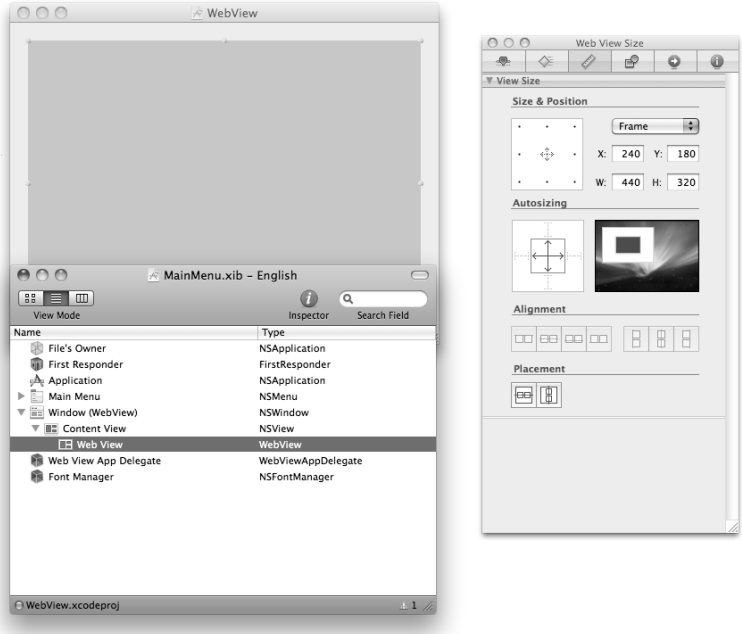
In Interface Builder, add a WebView to the Content View object from the library. Set its autosizing options, as shown in Figure 10.15.

Edit the `WebViewAppDelegate.h` file as follows:

```
@class WebView;
@interface WebViewAppDelegate : NSObject <NSApplicationDelegate>{
    NSWindow *window;
    WebView *webView;
}
@property (assign) IBOutlet NSWindow *window;
@property (assign) IBOutlet WebView *webView;
@end
```

Figure 10.15

Setting the autosizing options for a WebView forces it to follow the window size. If you don't set this feature, the size of the WebView remains fixed, which is sometimes, but not usually, a desirable property.



Save the file. In IB, link the webView outlet to the `WebView` object. Then, edit the `WebViewAppDelegate.m` file as follows:

```
#import "WebViewAppDelegate.h"
#import <WebKit/WebKit.h>
@implementation WebViewAppDelegate
@synthesize window, webView;
- (void)applicationDidFinishLaunching:(NSNotification *)
   aNotification {
    WebFrame *mainFrame = [webView mainFrame];
    NSURL *url = [NSURL URLWithString:@"http://www.apple.com"];
    NSURLRequest *aRequest = [NSURLRequest requestWithURL:url];
    [mainFrame loadRequest: aRequest];
}
```

Build and run the application and wait for the page to load. Then click the Store link. You should see the result shown in Figure 10.16.

Figure 10.16

The WebView running in its window. You can click active links to navigate.

The code is slightly less straightforward than it looks. The key feature is that the `mainFrame` property is the active browser window; a WebView is a wrapper for one or more web frames. `mainFrame` is the most important active frame. In most applications you can ignore the others.

WebView is a very sophisticated class. You can add browser essentials such as forward, back, and refresh buttons, but you can also implement controlled downloading with various delegate methods. There's even an `estimatedProgress` method that can drive a progress bar.

As a trivial exercise, you may want to try adding a URL bar and downloading a new page when the user enters a new URL. More sophisticated applications are also possible.

TIP

Most applications use WebView to access Web pages. Because URLs can be local, you can use WebView to access local HTML, CSS, or image content.

Summary

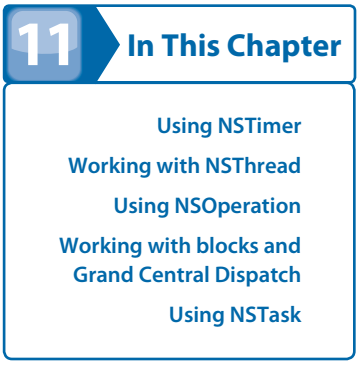
In this chapter, you learned about file paths, URLs, and their similarities and differences. You explored Cocoa's File Manager class, and you discovered how to compare files and build working open/save panes into your application. You also learned how to load and save data from the local file system, how to create and find valid paths, and how to get started with the file features of Cocoa's data objects.

Next, you explored `NSURL` in detail, and you found out how to use URL objects to load online data into Cocoa objects. You experimented with the bit.ly Web API, and you learned how to assemble simple text-based request and complex asynchronous requests that extracted information from an XML return.

Finally, you were introduced to the `WebView` object in WebKit, and you saw how easy it is to add a simple Web browser to any project.

Using Timers, Threads, and Blocks

It's often useful to do many things at the same time. Cocoa offers a selection of classes and techniques for creating multiple simultaneous events and processes and for managing the relative and absolute timing of events. The newest classes use a new code idiom called *blocks*, which was introduced in OS X 10.6. Blocks can be used anywhere in an application, but Cocoa's event and data collection classes are being modified to support blocks where previously they used delegates and calls to other classes.



Using NSTimer

`NSTimer` is the simplest Cocoa event timing class. To trigger events at regular intervals, use:

```
NSTimer *theTimer = [NSTimer
   scheduledTimerWithTimeInterval: float
      target: anObject
    selector: @selector(theTimerMethod:)
    userInfo: anOptionalObject
     repeats: YES];
```

Implement the timer method in the target object:

```
-(void) theTimerMethod: (NSTimer *) theTimer
   {
//This code is called on every timer tick
}
```

To stop the timer, use

```
[myTimer invalidate];
```

The shortest possible interval is 0.0001s, but on typical hardware, the shortest useful interval is 0.01s. Timer events aren't guaranteed to be timed accurately. On a busy system, the event can occur at any moment after the scheduled time. Each event is timed independently, so errors do not accumulate.

TIP

Although formally the time interval's type is `NSTimeInterval`, in practice you can use a float literal.

The `userInfo` property is a placeholder for an object that you want to pass to the timer method. To retrieve it, use

```
anObject = [myTimer userInfo];
```

An alternative method can fire the timer at a specific date/time, set with an `NSDate` object. The timer must be registered with a run loop.

```
NSTimer *theTimer = [[NSTimer alloc] initWithFireDate: aDate
                                  interval: float
                                    target: anObject
                                  selector: @selector(theTimerMethod:)
                                  userInfo: anOptionalObject
                                   repeats: YES];
[[NSRunLoop mainRunLoop]
addTimer: theTimer forMode: NSDefaultRunLoopMode];
```

`NSTimer` is ideal for low-precision counting and timing events. To drive animations, put view update code inside the timer method. The code can create simple view-based animations or update a complex OpenGL scene. For more details, see Chapter 17.

CAUTION

Note that Core Animation uses its own independent timers, for improved performance.

Using performSelector:

`NSTimer` is ideal for repeating events. For one-off timed events, `performSelector:` can be a better solution. It offers more control and better support for parameter passing, and it doesn't require a separate object.

`performSelector:` is built into `NSObject` and can be used in most Cocoa objects. The simplest implementation looks like this:

```
[targetObject performSelector: @selector(aMethod:)
                   withObject: anObject
                   afterDelay: floatLiteral];
```

`aMethod:` is triggered in the target object after the delay period. `anObject` is passed as a parameter. An object can call this method on `self` to send itself a delayed message.

Usefully, it's possible to cancel a message before it's triggered.

```
[NSObject cancelPreviousPerformRequestsWithTarget: targetObject];
```

All pending messages to `targetObject:` are cancelled. Optionally, you can specify a selector and object parameter to selectively cancel a message with those parameter settings.

Implementing a pause method

You can use `performSelector:` to simulate a pause feature. Assume you have a method called `aLongMethod`. To create a pause, split `aLongMethod` into two and use `performSelector:` to trigger the second part after a delay.

```
-(id) aLongMethod {
  //Do things
  //Pause here...
  [self performSelector: @selector(restOfMethod:)
             withObject: anObject
             afterDelay: 1.0];
}

-(id)restOfMethod: optionalObject {
  //Finish the rest after the pause
}
```

You can also use pause execution in the current thread with

```
[NSThread sleepForTimeInterval: 2.0];
```

Running the selector in a separate thread

The `performSelectorInBackground:` method is a simple way to run a method in a separate thread. The method specified by the selector detaches itself from the main thread and runs in the background. You have no control over thread priority. If you need to pass parameters, use `performSelectorInBackground: withObject:` and pack the parameters into an array or dictionary.

If you are not using garbage collection, the background method must implement its own autorelease pool, as described in the Managing thread memory section below.

Messaging across threads

`performSelector:` can trigger a method in a specific thread and wait until the method completes:

```
[targetObject performSelector: @selector(aMethod:)
onThread: aThread withObject: aParameterObject waitUntilDone: YES];
```

Optionally, you can add a `modes` parameter to the end of this signature to specify the thread mode. The parameter takes an `NSArray` of strings that defines the list of supported modes. There is no option to delay the message, but it's possible to use the `performSelector: withDelay:` method to delay a second `performSelector:` that triggers the message.

`aThread` is an instance of `NSThread`. A `performSelectorOnMainThread:` option selects target objects in the main thread, but it's possible to create separate custom instances of `NSThread` to spin off threads from the main application.

Working with NSThread

NSThread is becoming a legacy object in OS X, but it can still be useful in iOS, which has a simpler threading model. A thread is a method that runs independently of the object that triggered it. To create a thread, use

```
[NSThread detachNewThreadSelector: @selector(theThreadMethod:)
                         toTarget: self
                       withObject:anOptionalObject];
```

The method is implemented with

```
-(void) theThreadMethod: (id)theOptionalObject {
//Do things
[NSThread exit];
}
```

You can pack theOptionalObject with as much data as you need. It can be a single NSNumber or a complete dictionary of key-value pairs.

The exit call terminates the thread. Alternatively you can create an independent thread object:

```
NSThread *theThread = [[NSThread alloc]
initWithTarget: selfOrOther selector: @selector(theThreadMethod)
   withObject:anOptionalObject];
```

Pausing a thread

With a separate object, you can set its properties, including name, stackSize, and threadPriority — the latter on a range from 0.0 to 1.0. To run the thread, use the start method. To end a thread externally, send it a cancel message. Depending on the thread code, this may have no effect.

Threads, including the main thread, can be paused with

```
[NSThread sleepForTimeInterval: aFloat];
```

Call this from inside the thread you want to pause. It's a class method, but NSThread implicitly identifies the thread it's called from and pauses it accordingly. A corresponding sleepUntilDate: method sets an absolute wake-up time/date instead of an interval.

CAUTION

If you pause the main thread for more than a couple of seconds, the beach ball icon appears and the application is marked as "not responding."

Managing thread memory

On OS X you can use garbage collection to simplify thread memory management. On the iPhone, you must create a local autorelease pool for the thread when it launches, and `release` it just before the thread exits.

```
NSAutoreleasePool *aPool = [[NSAutoreleasePool alloc] init];
//Thread code goes here
[aPool release];
[NSThread exit];
```

This is standard boilerplate code, and is all that's required to implement an autorelease pool.

Handling UI and thread interactions

Only some of AppKit and UIKit are thread-safe, so accessing UI objects directly from a thread can be risky. Typically you use a thread to perform a background operation such as a download, search, or archiving operation that would otherwise slow down execution.

If you need to trigger UI updates, use

```
[mainThreadObject performSelectorOnMainThread: SEL(aMethod:)
                                   withObject: anObject
                                waitUntilDone: BOOL];
```

Place update code in a special selector/method that performs the update but runs in the main thread and can be triggered remotely. You can choose to pause the thread, while it waits for the main thread to complete the selector, or continue.

To receive messages from the UI, use the equivalent `performSelectorOnThread:` method, nominating the thread and a target object and method inside it.

TIP

It's sometimes useful to run `performSelectorOnMainThread:` just before a thread exits to let the main thread know that the other thread has completed its run. Use `waitUntilDone: YES` to make sure that the main thread has copied or read any results of the run, if it needs to.

Using NSOperation

`NSOperation` is more recent than `NSThread` and has the following advantages:

- **Threads are objects, not methods.**
- **Parameters can be passed through a customized** init **method with any number of items.**

- **Results can be read from the thread object's properties as it runs, or before it terminates.**
- **Queuing and concurrency can be managed.** Operations can run in the background *concurrently* — simultaneously — or sequentially. Concurrent processing can define a maximum number of running threads.

`NSOperation` requires more setup than `NSThread` and is less flexible. Use it for more complex threaded applications when you need more control over threading and when you need to pass a wider a selection of objects and values to a background task.

Next you'll create a simple example that uses `NSOperation` to list primes.

Creating an NSOperation object

An `NSOperation` object must have the following:

- **A custom** init **or** initWith: **method that initializes its local values.**
- **A** main **method with the main thread code.** Other local methods can be defined as needed.

The `init` method is called when the object is created, and the `main` method is triggered when the run queue runs the object. A header for a simple object that defines a `BackgroundTask` class follows:

```
#import <Foundation/Foundation.h>
@interface BackgroundTask : NSOperation {
     int limitInt;
}
@property int limitInt;
- (id) initWithInt: (int) anInt;
@end
```

To create an `NSOperation` object in Xcode, add an `NSObject` to your project in the usual way: right-click the Classes Folder, select Add New File, select OS X Cocoa Class and Objective-C class, and select Subclass of NSObject in the class pop-up menu. Name and save it.

After saving, change the class from `NSObject` to `NSOperation`, as shown previously.

CAUTION

Don't forget to use `GetInfo` to change the Path Type of the new files to Relative To Enclosing Group.

The code for a sample background task follows:

```
#import "BackgroundTask.h"
#import "NSOperationAppDelegate.h"
@implementation BackgroundTask
```

```
@synthesize limitInt;
- (id) initWithInt: (int) anInt
{
    if (![super init]) return nil;
    limitInt = anInt;
    return self;
}
- (void) main {
    for (int i=0; i< limitInt; i++) {
    if ([self isPrime: i])
    NSLog (@"%i is prime", i);
    }
   [[NSOperationAppDelegate shared] performSelectorOnMainThread:
   @selector(taskDidFinish:)
   withObject: self
   waitUntilDone: YES];
   return;
}
-(BOOL) isPrime: (int) testNumber {
    if (testNumber < 1) return NO;
    if ((testNumber ==2) || (testNumber == 3)
            || (testNumber == 1)) return YES;
    if ((testNumber%2 == 0) || (testNumber%3 ==0)) return NO;
    int divisor = 5;
    int limit = 1+sqrt(testNumber);
    while (divisor < limit) {
    if ( ((testNumber%divisor) ==0) ||
          (testNumber%(divisor+2)) ==0) return NO;
    divisor +=6;
    }
    return YES;
}
@end
```

Key features of the code include:

- **The** initWithInt: **method copies the initialization** int **to a local property.** `main` doesn't support parameter passing, so you must pass parameters via `init`**.**
- **The** main **loop cycles through a list of integers, tests them, and logs them if they're prime.** The prime test is in a separate local method. Because it's part of the `NSOperation` object, this method becomes part of the background task.
- performSelectorOnMainThread: **is used to notify the main thread when the thread completes.** This is an optional but useful feature.
- **The thread exits automatically.**
- [NSOperationAppDelegate shared] **returns a pointer to the app delegate that creates and runs this object.** It's described in more detail next.

Using NSOperationQueue

To run an `NSOperation` object, create an instance of `NSOperationQueue`, a queue manager object. Create as many instances of `NSOperation` as you need. Queue them with

```
[aQueue addOperation: anInstance];
```

This adds the operation to the queue. The queue runs the operation when it has a free execution slot, and manages objects before they're executed. Sample code that runs up to four instances simultaneously follows:

```
#import "NSOperationAppDelegate.h"
#import "BackgroundTask.h"
@implementation NSOperationAppDelegate
@synthesize window;
static NSOperationAppDelegate *sharedInstance;
+ (id) shared {
    //Other objects can use shared to get a pointer to this object
    return sharedInstance;
}
- (void)applicationDidFinishLaunching:(NSNotification *)
   aNotification {
   sharedInstance = self;
   NSOperationQueue *theQueue = [[NSOperationQueue alloc] init];
// [theQueue setMaxConcurrentOperationCount:1];
// When set to 1, dispatches tasks sequentially
   [theQueue setMaxConcurrentOperationCount:4];
   //When set to 4, dispatches up to 4 tasks concurrently
   BackgroundTask *task1 =
   [[BackgroundTask alloc] initWithInt:1000];
   [theQueue addOperation:task1];
   BackgroundTask *task2 =
   [[BackgroundTask alloc] initWithInt:2000];
   [theQueue addOperation:task2];
   BackgroundTask *task3 =
   [[BackgroundTask alloc] initWithInt:500];
   [theQueue addOperation:task3];
}
-(void) taskDidFinish: (id) sender {
   NSLog(@"Task %@ finished", sender);
}
@end
```

CAUTION

You can use a `suspend` method to control the queue. It doesn't suspend execution, as you might expect; it suspends dispatch. When the queue is suspended, no more tasks are queued until the suspension is cancelled.

The `+ (id) shared` method is a convenient way to give other objects a pointer to the App Delegate when they need to message it. It's not obligatory, but is a simple way to allow tasks to "call home" if they need to.

The `sharedInstance` property is set to `self`, and when other objects call `shared`, they receive the App Delegate's `self` value. They can then use

```
[[NSOperationAppDelegate shared] sendAMessage];
```

You can implement a `shared` property in this way in any custom class, *as long as there is exactly one instance.*

```
//in yourClass
static yourClass *sharedInstance;
sharedInstance = self;
//In another object
[[yourClass shared] sendAMessage];
```

Don't use this idiom when you create a class that requires multiple instances.

TIP

Don't be confused by the fact that the class name is `NSOperationAppDelegate`. This is a custom class used to demonstrate the features of `NSOperation`. It's not a prebuilt Cocoa class, and you can't use it in your own application. But you can use the `shared` idiom, as long as you slot in your class name, as shown previously.

Other key features of the code include:

- **An instance of** NSOperationQueue **manages the task list.**
- **Tasks are created with different initialization values.**
- **Tasks are added to the run queue with** addOperation**.**
- **The** taskDidFinish: **method is an optional custom method used to report the completion of a task.** It's triggered in each task by calling `performSelectorOnMainThread:`, as shown in the previous listing.

To summarize, follow these steps to use NSOperation:

1. **Subclass NSOperation to create a class for your task.**
2. **Add an** initWith: **method to implement parameter passing.** Copy passed parameters to local properties or values.
3. **Add a main method as an entry point to the task code.** Add extra local methods as needed.
4. **Optionally, add a `performSelectorOnMainThread` method to report when the task exits or when it needs attention.**
5. **In your main thread, create an instance of `NSOperationQueue`.**
6. **Create and initialize instances of your task class, as needed.**
7. **Run them by adding them to the task queue with `addOperation:`.**

8. **Optionally, use** setMaxConcurrentOperations: **to limit the maximum number of parallel tasks.**

9. **Optionally, use** setQueuePriority: **to set the priority of an operation, choosing one of the constants defined at the bottom of the** NSOperation **Class Reference.**

Figure 11.1 shows the results of running with `setMaxConcurrentOperations:` set to 4. The tasks run simultaneously, with equal priority.

Figure 11.1

Allowing concurrent operations makes it possible for the tasks to run in parallel.

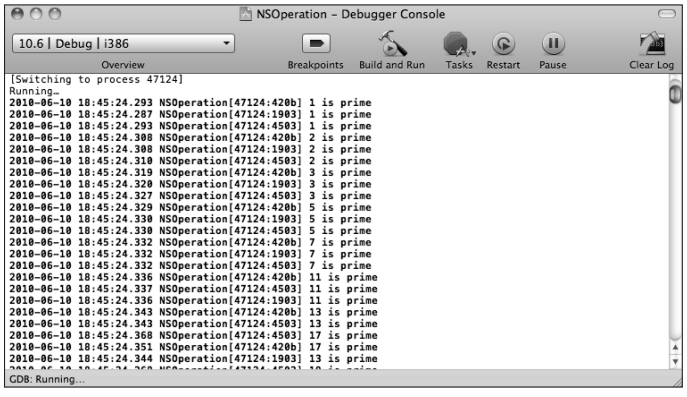
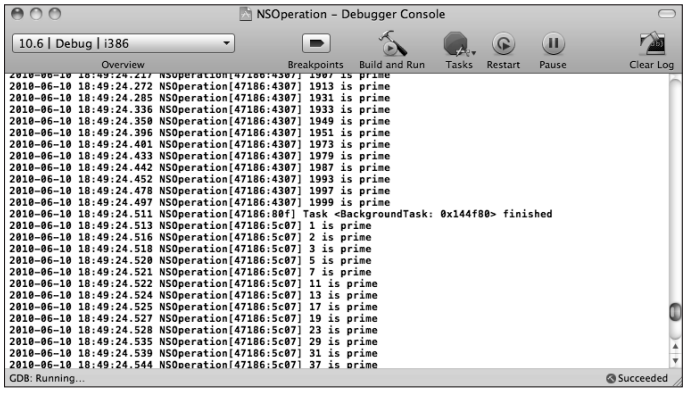


Figure 11.2 shows the result of decreasing `setMaxConcurrentOperations:` to 1. The tasks run sequentially. The queue manager dispatches them in the order they were queued.

Figure 11.2

When concurrent operations are disabled, tasks run sequentially.



One obvious disadvantage of `NSOperation` is its inflexibility. If your application needs to queue many different tasks, it can take a lot of time to create a custom subclass for each.

`NSOperation` has been extended to allow task code to be defined with blocks, which allow the use of inline code or predefined code blocks that are used more fluidly.

Getting Started with Blocks

Blocks are a new feature in Objective-C, borrowed from similar idioms in other languages, such as Ruby. A block is an object, but is used like a function. Block code can be created inline, or it can be defined in a separate declaration and called via a *block pointer*, which is similar to an object pointer, with modified syntax.

A block is an alternative design pattern that can replace delegation and selectors. Both delegates and selectors can be difficult to follow, because related code may be scattered among many objects. Block code is inline, which makes it easy to read and maintain. For example, in the block operation example later in this chapter, code that runs in a separate thread is immediately visible and can be edited in-place. It isn't buried in a method inside a separate object.

Technically a block encapsulates a block of code, *including the state of all relevant variables at the time it is created*. This complicates variable and parameter assignment. By default, variable values are fixed when the block is defined. To make them mutable, you must prefix them with a `__block` directive, defined with a double underscore.

Block support was added to some of Cocoa's key classes in OS X 10.6 and will be developed further in future versions. For some applications, blocks offer a viable alternative to delegation, because they allow inline code that is easier to write and easier to understand.

Although blocks are often associated with multithreading, blocks are a separate technology. They're not inherently multithreaded, and can be used as standard single-threaded Objective-C code. But some of the new Cocoa extensions make it possible to define a task with block code and launch it, in much the same way that `NSTask` calls a selector to define a method and then launches the method in a separate thread.

Understanding block syntax

Blocks have two sections: an optional header and a body that contains the block code. Classes that support blocks allow body-only blocks to be used as inline code. Depending on the context, this may disable certain features, such as parameter passing or support for return values.

Blocks are marked with the ^ (caret) character and delimited with curly brackets, followed by an apostrophe.

```
aBlock = ^{ some code goes here};
```

If no parameters or return values are defined, void is assumed. You can call the block with

```
aBlock();
```

To define a return value, use

```
aBlock = ^(int n) {return n*20};
NSLog(@"%i", aBlock(3)); //Logs 60
```

The full block declaration includes the return type, the name, and the parameter list, followed by a copy of the parameter list, followed by the code.

```
int (^aBlock) (int) = ^(int n) {return n+1};
```

If the return type is missing, the compiler *assumes it from the code* — which is why the previous two lines of code were valid.

Fixing variables and values

If a block references a variable or value when it's created, the value is frozen into the code. Parameters are referenced in the usual way, but if you don't define a variable as a parameter, the block makes a copy of its current state when it's created and doesn't allow it to be changed. For example:

```
int b;
b=2;
aBlock = ^(int a) {return a*b};
NSLog(@"%i", aBlock(3)); //Logs 6
b=3;
NSLog(@"%i", aBlock(3)); //Still logs 6
```

To convert `b` into a block variable, add a block underscore prefix to the original definition:

```
__block int b=3;
NSLog(@"%i", aBlock(3)); //Logs 9
```

Technically, this replaces the constant value of `b` frozen into the code with a reference. The reference points to a location in block memory, which is a separate memory area that persists as long as at least one block is active. In simple terms, `b` can now be accessed as a variable.

Blocks are objects, and they can be copied, stored in arrays and dictionaries, and archived. They can also be triggered remotely like methods, using dot syntax. Internally you can run a block with

```
self.aBlock();
```

You can also declare blocks as properties so that other objects can run them. To make them accessible, declare a `typdef`, which is similar to a standard C `typedef` and can be used like a class name.

```
typedef int (^MyBlockType) (int);
MyBlockType aBlock = ^(int aNum) {
return aNum*42;
};
```

Add the following to the header file:

```
@property (readwrite, copy) MyBlockType aBlock;
```

You can then run the block in an instance of the surrounding class with

```
someInstanceName.aBlock(anInt);
```

Using NSBlockOperation

`NSBlockOperation` supports both named and inline blocks, but does not allow parameters to be passed or returned. The equivalent block-based implementation of your prime generator follows:

```
#import "NSBlockOperationAppDelegate.h"
@implementation NSBlockOperationAppDelegate
@synthesize window;
- (void)applicationDidFinishLaunching:(NSNotification *)
   aNotification {
    NSOperationQueue *theQueue =
    [[NSOperationQueue alloc] init];
    int limitInt = 1000;
    NSBlockOperation *block1 =
    [NSBlockOperation blockOperationWithBlock:
    //***Start of the block
    ^{
        for (int i=0; i< limitInt; i++) {
            if ([self isPrime: i])
            NSLog (@"%i is prime", i);
        }
     }];
     //***End of the block
    [block1 setCompletionBlock:^{
    NSLog(@"Finished");
    }];
    [theQueue addOperation:block1];
     return;
}
-(BOOL) isPrime: (int) testNumber {
    if (testNumber < 1) return NO;
    if ((testNumber ==2) || (testNumber == 3)
            || (testNumber == 1)) return YES;
    if ((testNumber%2 == 0) || (testNumber%3 ==0)) return NO;
    int divisor = 5;
    int limit = 1+sqrt(testNumber);
    while (divisor < limit) {
    if ( ((testNumber%divisor) ==0) ||
          (testNumber%(divisor+2)) ==0) return NO;
    divisor +=6;
    }
    return YES;
}
@end
```

The NSOperationQueue is created and used as before, but is primed with an NSBlock Operation object created in the code. The active code is an inline block that starts immediately after blockOperationWithBlock:.

This example implements an optional feature. A separate setCompletionBlock: method defines another block that is called when the task terminates. You can also add multiple execution blocks to a single block operation object, so they run simultaneously, subject to a concurrency limit.

This code is much simpler than the version that used NSOperation, and it is easier to write. There's no need to create a separate class, and a completion method can be defined inline.

But NSBlockOperation has a significant limitation: it doesn't support parameter passing or return parameters. The signature of a block operation is void (^) (void), and this can't be changed. Fortunately, there is a workaround.

Passing parameters to NSBlockOperation

Passing parameters to a block isn't entirely straightforward. To illustrate this, you'll create a very simple block that logs an int. You'll then queue it twice, with two different parameters. The code is as follows:

```
- (void)applicationDidFinishLaunching:
(NSNotification *)aNotification {
    int limitInt = 0;
    NSOperationQueue *theQueue = [[NSOperationQueue alloc] init];
    //*** Block definition starts here
    void (^aBlock) (int) = ^(int thisInt) {
      NSLog(@"i: %i", thisInt);
    };
    //*** Block definition ends here
    limitInt = 100;
    NSBlockOperation *block1 =
      [NSBlockOperation blockOperationWithBlock: ^{
      aBlock(limitInt);
      }];
    [theQueue addOperation:block1];
    limitInt = 1000;
    NSBlockOperation *block2 =
      [NSBlockOperation blockOperationWithBlock: ^{
      aBlock(limitInt);
      }];
   [theQueue addOperation:block2];
}
@end
```

The block definition defines the block code. This code is used to create two separate tasks with the same code.

The NSOperationQueue is managed as before. addOperation: is used to queue each block operation as it's created.

The block syntax itself is nested. You can't pass parameters to the block defined by the blockOperationWithBlock:, but you can treat that block as a dummy wrapper and call your active block inside it. This solves the problem — with the caveat that aBlock uses a parameter reference rather than a parameter value, taking the current value of limitInt.

If some other process changes limitInt before a block is dispatched, it will take the runtime value, which may not be the value you set here. For reliable code, it's best to pass parameters as literals, or to make a local copy of a value that can be passed to the block. Both options guarantee that the value won't change while the code runs.

Introducing Grand Central Dispatch

NSOperation is a wrapper around a lower-level technology called Grand Central Dispatch (GCD). In most applications, you can do everything you need with NSOperationQueue. GCD offers improved low-level thread management that gives you more control over how threads are grouped and run. It implements a separate collection of queues that runs independently of the main queue, and also gives finer control over priority.

GCD understands blocks; for example, to start a thread with GCD use:

```
dispatch_async(queue, ^{ block code});
```

For more details about GCD, see the Grand Central Dispatch Reference in the Documentation.

Using NSTask

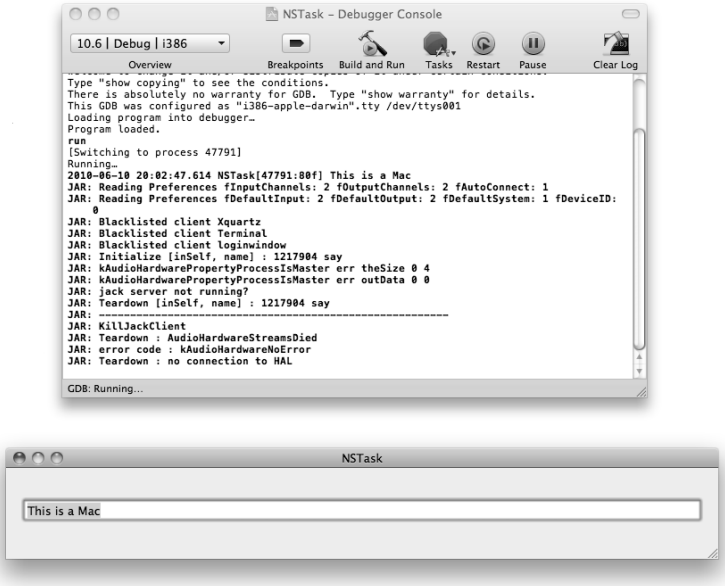
Cocoa includes another class that supports threading and remote execution. NSTask is a wrapper for system-level Unix calls. You can use NSTask to invoke the shell to execute simple commands such as ls, or to run utilities such as traceroute or ping.

By default, NSTask pipes return data back to stdio — the system input/output stream — which is connected to the console. You can run an NSTask and monitor the results in the console window. To capture the output in your own application, create an instance of NSPipe; connect it to the task; create an instance of NSFileHandle; implement an observer method that captures data as it arrives; and pass it back to your application.

There are few applications where this is necessary or useful. A simple example that calls the say speech synthesizer utility from a text view appears below. Figure 11.3 shows the finished application.

Figure 11.3

Make your Mac talk with `NSTask` and `say`. Running `say` triggers a complex sequence of events, which are logged to the console.



The window includes a single text view, which is linked via an outlet to the App Delegate. The `controlTextDidEndEditing:` method is called when the user presses Return. The string is wrapped in quote marks and passed as a parameter to `NSTask`.

`NSTask` requires two values. The first is the launch path that defines the location of the utility. For `say`, the path is `/usr/bin/say`. The second is an array of arguments, which is equivalent to the `args` array used in C applications, but is passed with the `setArguments:` method. In this example, the quoted string is the only parameter. The code is as follows:

```
@synthesize window, theText;

- (void)applicationDidFinishLaunching:(NSNotification *)
   aNotification {
   theText.delegate = self;
}
- (void)controlTextDidEndEditing:(NSNotification *)aNotification
   {
```

```
    NSLog(theText.stringValue);
    NSTask *aTask = [[NSTask alloc] init];
    [aTask setLaunchPath:@"/usr/bin/say"];
    NSString *talkString =
    [NSString stringWithFormat:@"\"%@\"", theText.stringValue];
    NSArray *args = [NSArray arrayWithObjects: talkString,nil];
    [aTask setArguments: args];
    [aTask launch];
}
@end
```

It can take a few moments for the set-up process to load and initialize the synthesizer and begin speaking. Once the code is running, you can create multiple parallel tasks by pressing the Return key over and over. Each runs and terminates independently, and the synthesized speech overlaps.

TIP

The Web site for this book includes an alternative version of the code that can run any command with arguments and has a kill feature. You can download the code at `www.wiley.com/go/cocoadevref`. The standard Unix file path searching feature built into Terminal isn't implemented, so you must specify all file paths explicitly.

Summary

In this chapter, you learned how to manage parallel tasks and events. You were introduced to various applications for `NSTimer`, and explored how to use `performSelector:` to trigger messages after a delay.

Next you learned how to use `NSThread`, and how to create and schedule methods so that they run in a separate thread. You discovered how to use some of the more complex variations of `performSelector:` to pass messages back to the main thread, to manage the UI, or to collect data.

You were introduced to `NSOperation`, and you discovered how to create subclasses that can run as separate background tasks, how to manage the operation queue, and how to control thread priority.

You explored blocks, discovered how to use `NSBlockOperation` to schedule block events, and investigated some of the limitations of parameter passing. You were also introduced briefly to Grand Central Dispatch (GCD), and you discovered how to schedule block tasks using an alternative low-level interface.

Finally, you experimented with `NSTask`, creating a simple interface to the Mac's built-in speech synthesizer.

Managing Data and Memory in Cocoa

Cocoa includes unique data collection objects, which are used almost as regularly as floats, ints, and character strings are in C and Java. You can use these objects as custom object stores, but you'll also spend time packing data into a collection and passing it to a Cocoa class, and unpacking data from a collection returned by a Cocoa class.

CAUTION

Cocoa doesn't always use data collections efficiently. Certain classes force you to pack data into an array or dictionary when you want to pass a single string or a number. This adds overhead and complicates the code, but it is sometimes obligatory.

Basic data collection skills include:

- Understanding the difference between mutable and standard classes.
- Understanding the difference between arrays, sets, dictionaries, and byte data.
- Finding objects by index, key, or other search options.
- Enumerating objects and processing them.
- Creating single-object collections.
- Counting, listing, sorting, and summarizing the contents of a collection.
- Using key-value coding to initialize and read the settings and parameters used in some of Cocoa's frameworks; for example, some of the media classes require and return key-value dictionaries.
- Saving and loading objects to and from disk.

TIP

Data collection objects have very little in common with their simpler C-language counterparts. A C array is a very simple data type, with limited features. `NSArray` shares the same indexed access model, but it supports powerful features such as counting, searching, and enumeration.

In This Chapter

Introducing data collection objects

Using `NSCoder` and `NSData`

Archiving and de-archiving object properties

Managing memory

Data collection objects can be grouped and nested. It's not unusual to fill an array with dictionaries, and key-value paths can become complex. It can be *very* useful to sketch class graphs that illustrate how objects are linked and how they interpret key-value combinations.

Introducing Data Collection Objects

Cocoa's data collection objects have a number of unusual features:

- **They implement their own disk access methods.** You can save and load the contents of any data object to disk. You can also load data from a URL into the object.
- **By default, data is defined at initialization and is read-only.** If you need to change the contents of a collection, use a *mutable* subclass.
- **Data collections store objects.** Simple C types must be "objectified." Typically this means converting ints, floats, and other types into an instance of `NSNumber`, or using `NSValue` as a wrapper for other data types. Strings must be an instance of `NSString`.
- **Data collections support implicit enumeration methods.** You can run a method on every object in the collection with a single line of code.
- **Modifying a mutable data collection doesn't trigger a Key-Value Observing (KVO) message or modify the value of an object in a collection.** If you want a collection that triggers KVO responses, you'll need to add some work-around code.

Table 12.1 introduces the key features of each object.

Table 12.1 Cocoa Data Objects

Data objects	*Description*
`NSArray`	Stores objects in an indexed list. Supports enumeration, predicate searching, and sorting.
`NSMutableArray`	Supports insertion and removal of objects. Inserting an object automatically increases the indexes of the objects after it.
`NSDictionary`	Stores objects as key-value pairs. Supports key and value listing, key enumeration, object enumeration, and predicate searching.
`NSMutableDictionary`	Allows insertion and deletion of key-value pairs, and allows modification of the value or object linked to a key.
`NSSet`	Stores an unordered collection of objects without indexes. `NSSet` is faster than `NSArray` when you need to test whether an object is in a collection.
`NSMutableSet`	Supports insertion and deletion of objects, and supports union, intersection, and difference operations between sets.

Data objects	Description
`NSValue`	Typically used the data element in collections. Works as a wrapper for conventional C or other Objective-C data types. Objectifies them and provides a standard interface for accessing, copying, and comparing them. Also supports pointers, `NSRange`, and the `NSPoint`, `NSRect`, and `NSSize` types used in graphics.
`NSNumber`	`NSNumber` is a subclass of `NSValue` and works with numerical values.
`NSIndexSet`	A set of array indexes. Used for multiple array operations. `NSMutableIndexSet` can be modified dynamically.

Using objects, keys, and values

Key-value pairings are a fundamental feature of the data collection objects. But it's easy to misunderstand how keys, objects, and values are related.

Understanding objects and values

You might expect values to access the contents of an object. As Figure 12.1 shows, they don't.

The code creates a mutable dictionary, and adds various objects and values to it. The console output shows that the `valueForKey:` and `objectForKey:` methods *return the same object*.

When you're working with string keys, the two methods are synonymous. The critical difference between objects and values is more subtle and is not intuitively obvious.

- When you use `valueForKey:`, the key must be an `NSString`.
- When you use `objectForKey:`, the key can be any object in any class.

`objectForKey:` is immensely flexible and powerful. You can search, enumerate, and organize objects using any data type or class as a key. You can use this feature to search and organize collections of any object — pointers, graphical data, game tokens, and so on — and to pair objects in useful ways.

`valueForKey:` is a poor relation. It forces you to pair an object with a string. Some Cocoa frameworks use `valueForKey:` exclusively in their class definitions, so you're often compelled to use it.

Using setValue: forKey

`NSArray` and `NSSet` implement the `setValue: forKey:` method. This enumerates the items in the array or set and runs `setValue: forKey:` on each in turn.

Although these classes are nominally read-only, you can use this method to modify the values inside mutable collections stored inside a non-mutable object.

Figure 12.1

Illustrating how `valueForKey:` and `objectForKey:` return the same object. The difference between these methods is in the key, not in the object/value.

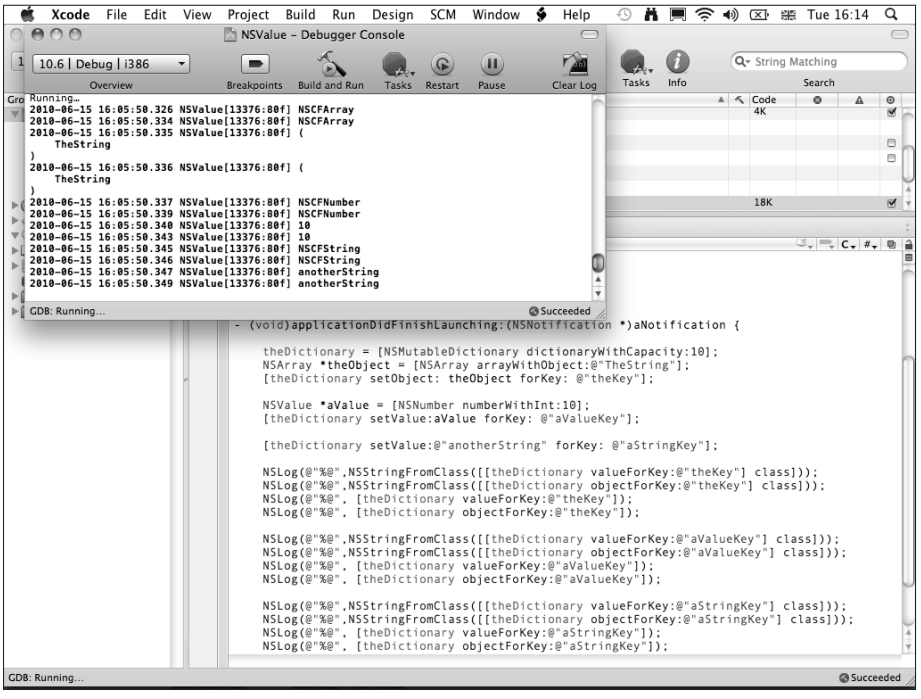


Figure 12.2 shows an example. An `NSArray` holds a series of `NSMutableDictionary` items. Running the `setValue: forKey:` method on the array modifies the values of the matching keys in every dictionary.

You can use this technique to implement a quick reset feature. For example, the array might contain an array of player information in a game. You can reset any one key value for every player with a single line of code.

Implementing Key-Value Observing

Data collection objects are not observable. Adding, removing, counting, and searching items don't trigger Key-Value Observing (KVO). To implement KVO, you must add it manually. Typically you wrap changes inside one or more editing methods. Each method can specify a constant that tells KVO about the edit. For example:

```
- (void) editMethod: (id) objectsToReplace
          atindexes: (NSIndexSet *) indexes {
   [self willChange: NSKeyValueChangeReplacement
```

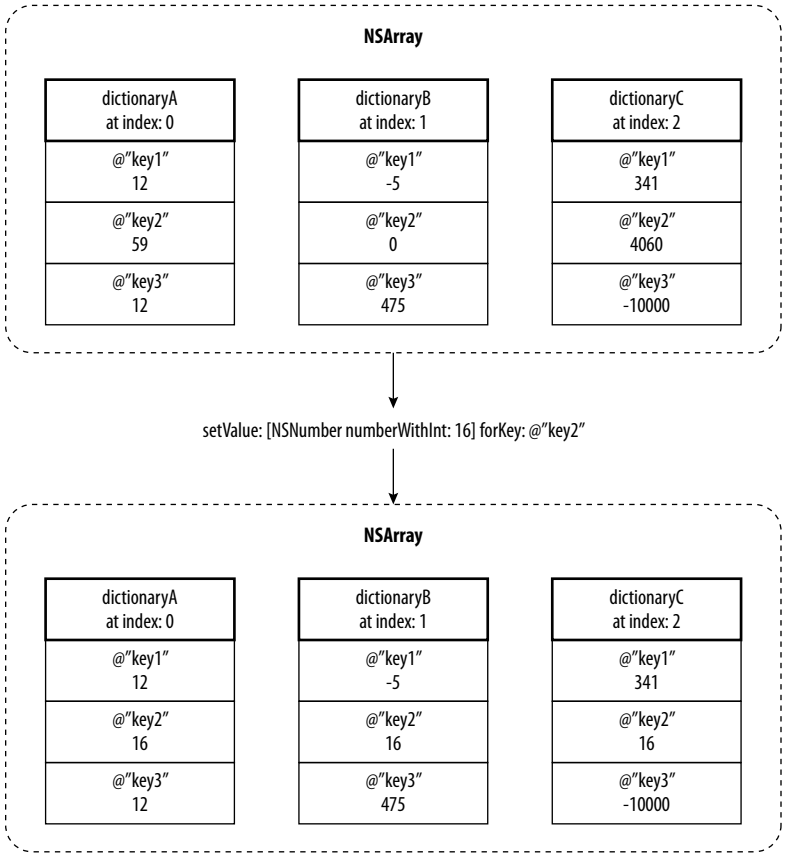
```
    valuesAtIndexes: indexes forKey: @"these objects"];
//Add code to replace some of the objects in the collection
    [self didChange: NSKeyValueChangeReplacement
    valuesAtIndexes: indexes forKey: @"these objects"];
}
```

The supported constants are:

- `NSKeyValueChangeReplacement`
- `NSKeyValueChangeRemoval`
- `NSKeyValueChangeInsertion`

Figure 12.2

`setValue: forKey:` can change values inside mutable objects — even when they're stored inside a non-mutable collection.



If your code doesn't need to know when items have been changed, removed, or inserted, you can trigger KVO with a dummy assignment after an edit.

```
self.dataCollectionObject = dataCollectionObject;
```

This tells KVO that the object has been accessed, but not if or how the collection was changed. It's a minimal solution, but it may be enough in some applications.

Using NSValue and NSNumber

You can add any object to any data collection. You can also mix objects in the same collection, which is one reason why the data collection objects are so powerful. By default, you can fill any data slot in any data object with any class.

NOTE

Cocoa has no concept of a typed array or set. If you need to enforce typing, you can implement it manually.

Because data collections store objects and not naked values, you often have to use `NSValue` and `NSNumber` to "objectify"' an existing value.

`NSNumber` is relatively easy to work with. To create an `NSNumber` object, use

```
NSNumber *aNumber = [NSNumber numberWith<Type>: rawValue];
```

The `NSNumber` Class Reference lists the supported types. Long and unsigned variants are available.

To convert an `NSNumber` back into a numeric C type, use

```
<a type> rawValue = [aNumber <type>Value];
```

For example, to convert to and from a float, use

```
NSNumber *aNumber = [NSNumber numberWithFloat: 2.71828];
float theNumber = [aNumber floatValue];
```

You can use a `stringValue` method to convert a number into a string. The formatting is arbitrary, so it's better to use the `NSString stringWithFormat:` method for conversions, because you can apply standard C format specifiers. For details, see Chapter 15.

`NSValue` is more open-ended, and it supports the handful of Cocoa data types that aren't objects, such as `NSRange`, `NSRect`, `NSPoint`, and others. You can also use it to "objectify" custom C data structures. The most useful built-in types include:

- Pointer
- Geometrical point (`NSPoint`)

- Geometrical rectangle (`NSRect`)
- Geometrical rectangle size (`NSSize`)
- Two-valued range (`NSRange`)

The syntax for these standard types is similar to that used by `NSNumber`:

```
NSValue *aValue = [NSValue numberWith<Type>: rawValue];
<a type> rawValue = [aValue <type>Value];
```

NSValue also supports arbitrary custom C `typedefs`, with a conversion feature.

```
typedef struct {
//Assorted data types
} StructName;
StructName letsMakeOne;
//Code to set struct values
NSValue *aStructWrapper = [NSValue value: &letsMakeOne
                              withObjCType: @encode(StructName)];
```

This creates an object called `aStructWrapper`, which contains the values in `letsMakeOne`, which is an instance of `StructName`. Once `aStructWrapper` is initialized, you can insert it into a data collection object.

The `withObjCType:` parameter tells `NSValue` how to organize the interval values. The value parameter tells `NSValue` what the values are.

To extract the values, use

```
[letsMakeOne getValue: &aStructWrapper];
```

This unwraps the contents of `aStructWrapper` and copies them into a variable with a matching `typedef`.

Using NSArray

`NSArray`'s init methods can create an array from a variety of data sources, including the following:

- Another array
- A file
- A data source accessed via a URL, which can be a local file or an online data source (see Chapter 10 for details).
- A single object
- A list of objects
- A C array of objects

You can usually ignore the `initWith...` methods and use their class equivalents. Use

```
NSArray *aNewArray = [NSArray arrayWith<data source>];
```

instead of

```
NSArray *aNewArray = [[NSArray alloc] initWith<data source>];
```

To create an array with a single object, use

```
NSArray *theArray = [NSArray arrayWithObject: theObject];
```

This is required throughout Cocoa when passing an object to another object that takes an array parameter. The array parameter can have many objects, but sometimes only one is needed.

When defining a list of objects with the plural variant of this method, terminate it with `nil`.

```
NSArray *theArray = [NSArray arrayWithObjects: object1, object2,
   nil];
```

As mentioned in earlier chapters, you can copy arrays by creating a new array with the contents of an existing array. `NSArray` includes methods that can modify the array as it's copied. They can extract a range of objects from the array, add a new object, or combine two arrays. For example:

```
NSArray *newArray =
[oldArray arrayByAddingObject: newObjectToAdd];
```

This copies the objects from `oldArray` to `newArray` and then appends `newObjectToAdd`.

Although `NSArray` isn't mutable, these modification methods make it possible to mutate the data in the array, at the cost of some extra code and another array object.

You can also copy an array directly using the `copyWithZone:` method. This is implemented by all data collection objects.

```
NSArray *copiedArray = [oldArray copyWithZone: NULL];
```

Use `NULL` to specify the default memory pool.

Using NSMutableArray

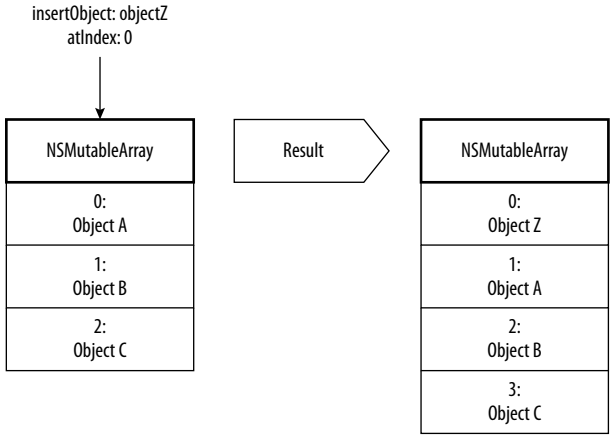
`NSMutableArray` supports editing but enforces a contiguous list.

```
[aMutableArray insertObject: anObject AtIndex: n];
```

The indexes of existing objects above n are automatically increased by one, as shown in Figure 12.3. The `removeObjectAtIndex:` method recalculates existing indexes in an analogous way.

Figure 12.3

Inserting an object into a mutable array automatically adjusts the existing indexes.



Related methods are available for removing, replacing, and exchanging objects. You can use `removeAllObjects` to empty the array.

You can hint a capacity for `NSMutableArray` by creating it with

```
aMutableArray = [NSMutableArray arrayWithCapacity: integer];
```

Mutable arrays expand and contract as needed. Hinting a capacity preassigns a block of memory, slightly increasing performance.

The capacity does not set the array bounds as it would for a C array. The bounds limit is set by the current item count, not the capacity.

The array contents must be a contiguous list. Gaps aren't permitted. Attempting to insert an item at `index:50` in an empty array raises an exception, even when the capacity is 50.

If you need to implement a non-contiguous array, use `NSDictionary`, pairing objects with arbitrary index keys.

Searching NSArray and NSMutableArray

`NSArray` and `NSMutableArray` organize data with the usual index model. `index:0` holds the first object, `index:1` the next, and so on. But these classes also implement powerful search options that allow random access.

The basic query method is `objectAtIndex:`

```
(id) theObject = [anArray objectAtIndex: anInteger];
```

This returns the object at the integer index. The object can be of any type. Your code must support `id` for the object, implement type checking, or enforce typing in some other way.

You can also search the array for a specific object, returning an index if it exists.

```
theIndex = [anArray indexOfObject: aSearchObject];
```

This method returns the first match, ignoring other matches. If no match is found, `theIndex` is set to the constant `NSNotFound`, which is defined as a very large number.

Optionally, you can limit the search to a range of indexes with an extra `inRange:` parameter that takes an instance of `NSRange`.

From OS X 10.6 onward, `NSArray` supports searches with arbitrary inline search code. This feature uses the new block technology described in Chapter 11. The inline code is completely customizable. You can implement any search condition or test. `NSArray` enumerates its contents, applies the test code, and returns an index if there's a match; for example:

```
theIndex = [anArray indexOfObjectPassingTest:
(BOOL (^)(id obj, NSUInteger idx, BOOL *stop)){
   if ([obj isEqual: searchObject]) {
   stop = YES;
   return idx;
   }
}];
```

The conditional defines the test. Here the test is a simple comparison, but the test can be arbitrarily complex, optionally referencing some of the internal properties of each entry in the array. The `stop` Boolean terminates the search.

Sorting NSArray and NSMutableArray

There is no way to sort an `NSArray` in place, and you can only sort an `NSMutableArray` in-place with custom code.

However, you can use the `sortedWith...` methods to create another array with the same contents sorted using any arbitrary function, selector, or sort descriptor.

Sorting is implicitly type-dependent, and it's impossible to define a general sorting solution for an array of arbitrary objects.

Typically you assume the array contains data of a single type, and then create a sort algorithm accordingly. For example a sorting function takes two items of a given type and returns three `NSInteger` comparison constants — `NSOrderedAscending`, `NSOrderedDescending`, and `NSOrderedSame`.

```
NSInteger aSort (id thingOne, id thingTwo, void *aContext) {
   v1 = [thingOne <type>Value];
   v2 = [thingTwo <type>Value];
   if (v1 < v2) return NSOrderedAscending;
   if (v1 = v2) return NSOrderedSame;
   if (v1 > v2) return NSOrderedDescending;
}
```

This assumes that values are available and direct comparisons are possible. Complex objects require a more complex solution.

You can then use the sorting function to create a sorted array.

```
NSArray *sortedArray =
[sourceArray sortedArrayUsingFunction: aSort context: NULL];
```

Sorting with a selector is similar, with the difference that the selector runs on each object in the array, passing another object as a parameter. The return codes are the same.

Arrays without using NSArray

If you don't need the features of `NSArray`, you can create a C array of object pointers with

```
ClassName *arrayName[numberOfEntriesConstant];
```

Each entry is a pointer to an instance of `ClassName`. This is equivalent to creating a list of related pointers, but it also supports indexed access.

```
ClassName *arrayName1;
ClassName *arrayName2;
etc…
```

You can set items individually:

```
NSString *someStrings [3];
someStrings[0] = @"aString";
someStrings[1] = anObject.name;
someStrings[2] = someStrings[0];
```

Note that `someStrings` is an array of pointer variables, *not* an array of string objects.

```
NSString *aString;
aString = @"One fish";
someStrings[0] = aString;
aString = @"Two fish";
//someStrings [0] is still @"One fish"
```

Be careful when using manual memory management. You can leak objects if you don't release them.

Using NSDictionary

`NSDictionary` collects key object/value pairs. It's more flexible than `NSArray`, allowing arbitrary indexing and object pairing. The keys can represent almost any data in any format, so it's possible to emulate a noncontiguous array with numerical keys.

You can create a dictionary by merging two arrays to create key-value pairs:

```
aDictionary =
[NSDictionary dictionaryWithObjects:
objectArray forKeys: keyArray];
```

Use `allKeys` to reverse this process and return an array of keys, and use `allValues` to return an array of objects/values.

You can also specify a list of alternating keys and values:

```
bDictionary =
[NSDictionary dictionaryWithObjectsAndKeys:
value1, @"key1", value2, @"key2", nil];
```

The sorting and enumeration features are similar to those in `NSArray`.

Accessing File Attributes

`NSDictionary` includes convenience methods for accessing a file handler's attributes dictionary. Instead of specifying an explicit key, you can call these methods directly to return various attributes; for example:

```
NSDate *creationDate = [anAttributesDictionary fileCreationDate];
```

You can use the attributes dictionary to access low-level file information. This information isn't usually required for simple read/write archiving operations.

Using NSMutableDictionary

Use `setObject: forKey:` and `setValue: forKey:` to add items to the dictionary or to replace existing items.

You can copy an entire existing dictionary with the `setDictionary:` method. Use this to make a mutable copy of dictionary data before editing it.

`removeAllObjects`, `removeObjectForKey:`, and `removeObjectsForKeys:` manage deletion. The latter takes an array of keys.

`NSMutableDictionary` supports capacity hinting. The key system doesn't implement bounds checking.

Using NSSet and NSMutableSet

Use `NSSet` to create an unordered collection of objects — specifically when ordering and indexing aren't important, but your application needs to check whether an object is included in a set or in some combination of two sets. Table 12.2 lists the most useful methods.

Table 12.2 Useful NSSet and NSMutableSet Methods

Method	*Description*
NSSet methods	
`intersectsSet:`	Returns TRUE if two sets share at least one common object.
`isEqualToSet:`	Returns TRUE if two sets are identical.
`isSubsetOfSet:`	Returns TRUE if every object in the original set is also in another set.
NSMutableSet methods	
`unionSet:`	Merges two sets.
`setSet:`	Replaces one set with another.
`minusSet:`	Removes items from the source set if they also appear in the target set.
`intersectSet:`	Removes items from the source set if they don't also appear in the target set.

`NSSet` also implements the enumeration, count, and search methods implemented by other collection objects. You can convert an array into a set with the `setWithArray:` initializer. However, you can't easily convert a set back into an array. Potentially, you can enumerate each item in a set and add it to an array, but, naturally, the ordering will be undefined.

Enumerating items

There are three ways to enumerate the items in a data collection.

Using NSEnumerator

You can ask the class to create an `NSEnumerator` object using one of the available class methods, and then use the `nextObject` method to step through each item in turn. For example, you would enumerate an array as follows:

```
NSEnumerator *enumerator = [anArray objectEnumerator];
id object;
while (object = [enumerator nextObject]) {
   //do something with the object
}
```

Each class can return different enumerators. Table 12.3 lists the most useful options. Only `NSArray` implies a defined order. For other classes, the order generated by `objectEnumerator` isn't explicitly defined.

Table 12.3 essential Enumeration methods

Data objects	*Description*
`NSArray`	`objectEnumerator` `reverseObjectEnumerator`

`NSDictionary`	`objectEnumerator`
	`keyEnumerator`
	`enumerateKeysAndObjectsUsingBlock{}`

`NSSet`	`objectEnumerator` `enumerateObjectsUsingBlock{}`

Using fast enumeration

Cocoa also supports *fast enumeration,* which is implemented with minimal code, optimized for efficiency.

```
for (id anItem in dataCollectionObject) {
   //Process anItem
}
```

If the items in the data collection are of a single type, replace `id` with the type.

You can use `break` to terminate the enumeration loop, and use a counter variable — incremented manually — to count indexes and optionally apply a conditional to select a specific item, apply a range, and so on.

```
for (id anItem in dataCollectionObject) {
   if ([anItem isEqual: aTestItem])
       //Optionally, do something with anItem
       break;
   else
       //Process the other items
   }
}
```

Using implicit enumeration

The `makeObjectsPerformSelector:` method enumerates the collection and runs the selector on each item. It's equivalent to

```
for (id anItem in dataCollectionObject) {
   [anItem performSelector: @selector(aMethod)];
}
```

Optionally, you can pass an object to the method as a parameter, using the `withObject:` extension. The method parameter should be `id`. Strong typing isn't supported.

Performance considerations

Fast enumeration is fast. It runs slightly faster than a for-next loop or an enumerator object. Cocoa's core is optimized for object processing, so data collection enumeration can be as fast as C array indexing. Generally, you can use fast enumeration without performance worries.

Enumerating mutable collections

Don't enumerate a mutable collection while it's being modified. If the data in the collection changes, the enumerator raises an exception.

The safe way to enumerate mutable objects is to copy them to a temporary read-only object, and then run the enumeration on the read-only data. Enumerators aren't suitable for conditional editing.

```
//Don't do this!
for (id anItem in mutableCollection) {
   if ([anItem isEqual: aTestItem])
       //code to replace anItem or delete it from the collection
   }
}
```

Counting items

All the data collection objects implement the *count* method, which returns an integer with the number of items; for example:

```
int items = [aDictionary count];
```

Archiving and de-archiving collection objects

All the collection objects implement direct disk access. You can save and load data directly to and from disk with an arbitrary filename by calling a disk access method on a collection object. For example, to save data from an array, use

```
[anArray writeToFile: aFilePath atomically: YES];
```

You can reload the data with

```
anArray = [NSArray arrayWithContentsOfFile: aFilePath];
```

You don't need to create a file manager object. These methods handle low-level disk access for you.

CROSS-REF

For information about creating and using file paths and URLs, see Chapter 10.

A typical application may use tens or hundreds of data collection objects in its model. It would be inefficient to create a separate file for each one. Fortunately, Cocoa includes a "collection of collections" class that can create a single data buffer from a group of collections, save it to disk as a single file, and retrieve the original collection data when it's reloaded. The class is called `NSCoder`.

Using NSCoder and NSData

`NSData` is Cocoa's simplest data collection object. It's a wrapper for an arbitrary blob of binary data. `NSData` doesn't care what the data means, or how it's organized. It doesn't support keys, indexing, sets, dictionaries, or other access methods. It simply treats the data as a contiguous byte buffer.

`NSMutableData` adds byte-level editing. You can change the length of the buffer, append more bytes or data from another `NSData` object, and replace a range of bytes in the buffer.

`NSData` is sometimes used for generic storage in an application. More typically, it's used as a file or data transfer buffer. Chapter 10 has an example of an asynchronous Web download that appends incoming data to a buffer as it's downloaded. In the following section, `NSData` is paired with NSCoder to create a file buffer for application data.

Introducing archiving and coding

Most applications save documents and data into a single file. Cocoa's `NSCoder` class merges an object's property values into a single `NSData` object. The data can then be saved to disk as a single file. `NSCoder` supports a decoding method that reverses the process, unpacking the merged data and copying it back to an object's properties.

There are two components to an `NSCoder` implementation. First, every class that supports archiving must implement two methods:

- `encodeWithCoder:` defines how the class converts its properties into blocks of binary that can be added to a file buffer.
- `initWithCoder:` defines how binary is converted back into property values.

The conversion is managed by two methods called `encode:` and `decode:`. These methods are run on each property in turn within `encodeWithCoder:` and `decodeWithCoder:`.

- `encode:` generates a block of binary for a single property and links it to a key string.
- `decode:` searches for a key string and converts the associated binary back into an object or value.

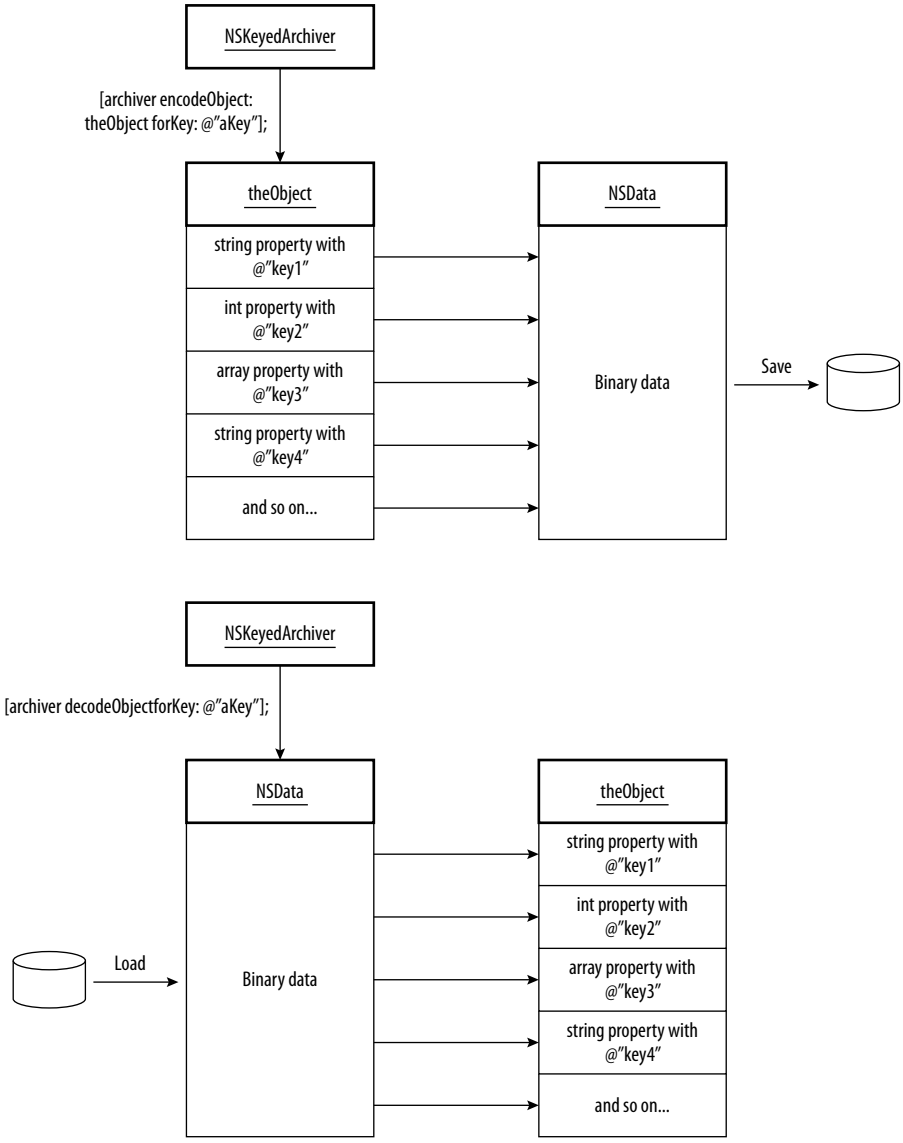
Second, the class responsible for saving and loading data uses an instance of `NSKeyedArchiver` to manage the conversion of object properties to an `NSData` block, and back again.

`NSKeyedArchiver` runs the `encodeWithCoder:` and `initWithCoder:` methods for you. You never need to call them directly. You can simply tell `NSKeyedArchive` that you want to convert an object into binary, or convert a file of binary back into an object, passing an `NSData` object as a parameter. The conversion is automatic.

Even more usefully, the `NSCoder` process is recursive. Data collection objects automatically run `NSCoder` on their elements. If they need to, the elements run `NSCoder` on their subelements, and so on, until no more recursion is needed. As long as every class in the tree implements `NSCoder` correctly, this happens effortlessly with no extra code. Figure 12.4 summarizes the process.

Figure 12.4

Saving and loading composite objects to disk. The `NSCoder` methods are built into the class implementation. They're called indirectly by `NSKeyedArchiver`.



Creating a class with NSCoder

I'll show you how to create a simple implementation of `NSCoder` and `NSKeyedArchiver` to save and load data in a form. The data is stored in a custom data model class. In a commercial application, multiple instances of the class could be used to store many forms. In this example, you'll concentrate on the code that saves and loads a single instance, using a single file with a fixed name. The code writes data from the form UI before saving the values. When the data is reloaded, the code creates a new data instance with the saved values, and copies them back to the form for display and editing.

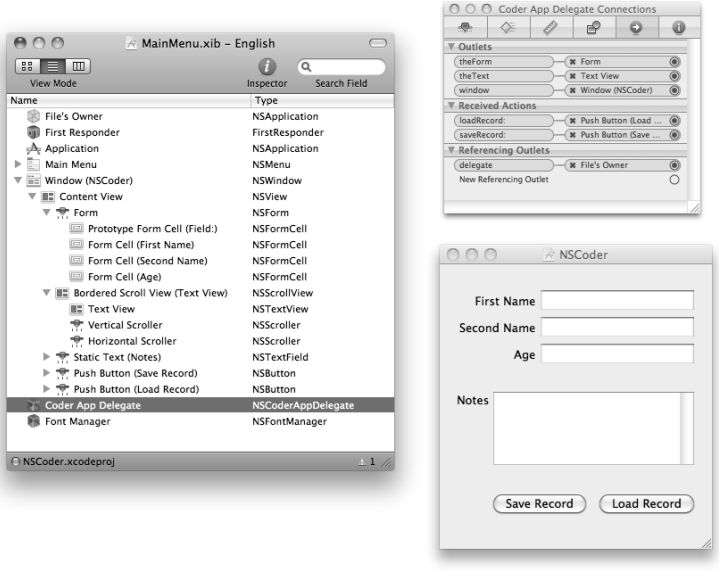
TIP

If you want to experiment with saving and loading multiple files, you can combine this project with the sample code for save/open panes in Chapter 10.

The project nib is shown in Figure 12.5. It uses an `NSForm` object to store three text strings, one of which is interpreted as an integer. An associated `NSTextView` object provides a free text field for notes. The `NSForm` and `NSTextView` are linked to outlets in the App Delegate. Two buttons trigger disk save and load actions.

Figure 12.5

The application nib file uses an `NSForm`, which is simply an array of text fields, accessed by index.



The code that follows lists the header and implementation for the custom data class. Only properties with `encode:` and `decode:` methods are archived. This is a minimal example; in a practical class, you can add other methods to implement other features, as needed.

TIP

In this example, you'll implement full archiving of every property. It's sometimes useful to archive selectively. You can do this easily — just leave some of the properties out of the coding methods.

The keys associated with each property are arbitrary. The key strings are for encoding and decoding only. Each key must be unique.

```
@interface DataClass : NSObject {
   NSString *firstName;
   NSString *secondName;
   int       theAge;
   NSString *theNotes;
}
@property (retain) NSString  *firstName;
@property (retain) NSString  *secondName;
@property (retain) NSString  *theNotes;
@property int theAge;
-(void)encodeWithCoder: (NSCoder *)encoder;
-(id) initWithCoder: (NSCoder *)decoder;
@end
@implementation DataClass
@synthesize firstName, secondName, theNotes, theAge;
-(void)encodeWithCoder: (NSCoder *)encoder
{
   [encoder encodeObject: firstName forKey: @"firstName"];
   [encoder encodeObject: secondName forKey: @"secondName"];
   [encoder encodeObject: theNotes forKey: @"theNotes"];
   [encoder encodeInt: theAge forKey:@"theAge"];
}
-(id) initWithCoder: (NSCoder *)decoder;
{
   if (self = [super init])
   {
      firstName = [decoder decodeObjectForKey:@"firstName"];
      secondName = [decoder decodeObjectForKey:@"secondName"];
      theNotes = [decoder decodeObjectForKey:@"theNotes"];
      theAge = [decoder decodeIntForKey:@"theAge"];
   }
   return self;
}
@end
```

There are a few subtleties in this code. If an object is a subclass of `NSObject`, as here, initialize it with

```
if (self = [super init])
{…
```

If it's a subclass of some other class, initialize it with

```
if (self = [super initWithCoder: coder])
{…
```

This implements the recursion, which stops automatically when the superclass is `NSObject`.

Object coding should happen automatically, irrespective of the complexity or internal structure of an object. For simple types, use type-specific encode and decode statements.

```
encodeBool: forKey:
encodeInt: ForKey:
encodeFloat: ForKey:
encodeDouble: ForKey:

decodeBoolForKey:
decodeIntForKey:
decodeFloatForKey:
decodeDoubleForKey:
```

Other numeric types can be wrapped inside an `NSNumber` or `NSValue`.

Archiving and de-archiving an object

The code for saving and loading a record follows. In this example, the code is in the App Delegate, but you can implement it wherever a class needs to load and save the values in another class.

```
#import "NSCoderAppDelegate.h"
#import "DataClass.h"
@implementation NSCoderAppDelegate
@synthesize window, theForm, theText;
NSString *savePath;
- (void)applicationDidFinishLaunching:
   (NSNotification *)aNotification {
   //Create the file path
   NSString *documentsDirectory =
   [NSHomeDirectory()
     stringByAppendingPathComponent:@"Documents"];
   savePath = [documentsDirectory
   stringByAppendingPathComponent:@"SaveState.obj"];
}
- (IBAction) saveRecord: (id) sender {
   //Create an instance of the record object
```

```
    DataClass *anInstance = [[DataClass alloc] init];
    //Prepare it for archiving by copying the form data to it
    anInstance.firstName = [[theForm cellAtIndex:0] stringValue];
    anInstance.secondName = [[theForm cellAtIndex:1] stringValue];
    anInstance.theAge = [[theForm cellAtIndex:2] intValue];
    anInstance.theNotes = theText.string;
    //Run the archiver
    NSMutableData *saveData = [[NSMutableData alloc] init];
    NSKeyedArchiver *archiver =
    [[NSKeyedArchiver alloc]
    initForWritingWithMutableData:saveData];
    [archiver encodeObject: anInstance forKey:@"SaveState"];
    [archiver finishEncoding];
    if (![saveData writeToFile:savePath atomically:YES])
    NSLog(@"Save error");
}
- (IBAction) loadRecord: (id) sender {
    //Load a blob of data from the saved file
    NSData *restoreData =
    [[NSMutableData alloc] initWithContentsOfFile:savePath];
    //Initialize an unarchiver with the loaded data
    NSKeyedUnarchiver *unarchiver =
    [[NSKeyedUnarchiver alloc]
    initForReadingWithData:restoreData];
    //Create a new data object instance
    DataClass *anInstance = [[DataClass alloc] init];
    //Initialize the properties with saved data
    anInstance = [unarchiver decodeObjectForKey:@"SaveState"];
    //Copy the values back to the form
    [[theForm cellAtIndex:0]
           setStringValue: anInstance.firstName];
    [[theForm cellAtIndex:1]
           setStringValue: anInstance.secondName];
    [[theForm cellAtIndex:2] setStringValue:
    [NSString stringWithFormat: @"%i",anInstance.theAge]];
    theText.string = anInstance.theNotes;
}
@end
```

Key features of the code include:

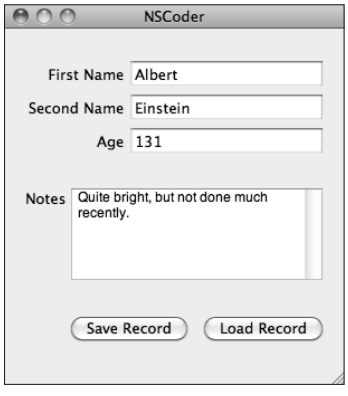
- **The file path is a static path to a single file in the Documents directory.**
- **The** saveRecord: **method copies information from the form to the data record before saving it.** In another application, data might be loaded into a record in some other way, or generated automatically. The details aren't important, as long as the data object holds valid values before it's archived.
- **The** NSForm **object is an array of text fields.** The contents of each cell in the list are accessed by index.

- **The archiver is prepared by creating an instance of** NSMutableData **and passing it to** NSKeyedArchiver **as an initialization parameter.** This sets up an empty buffer for the archiver.
- **The key in the** encodeObject: **method is arbitrary.** It doesn't matter what it is, but it must match the key in the `decodeObjectForKey:` method in the load code.
- **The** finishEncoding: **method implements the archiving process.** You must place this method after `encodeObject:`.
- The unarchiving code is a mirror image of the archiver, but it uses NSKeyedUnarchiver, **which runs the** decode: **method for each property in the file.**

Figure 12.6 shows the application running. Clicking Save Record saves the data in the UI. Clicking Load Record restores saved data from a file.

Figure 12.6

The single form example saves and loads a record in this one form. You can easily extend the code to support multiple forms.



Because `NSCoder` works recursively, it's almost trivial to extend this example to support multiple forms. Add another data class that uses an array or dictionary to hold multiple records, and then implement custom `NSCoder` methods to support archiving for the class. Run the archiving methods on your new composite class to save all records into a single file. With some additions to the UI, you can build a useful card index application with very little effort, adding optional search and listing features using the enumeration, sorting, and search features built into `NSArray` and `NSDictionary`.

TIP

For an alternative card index application, see Chapter 14.

Managing Memory

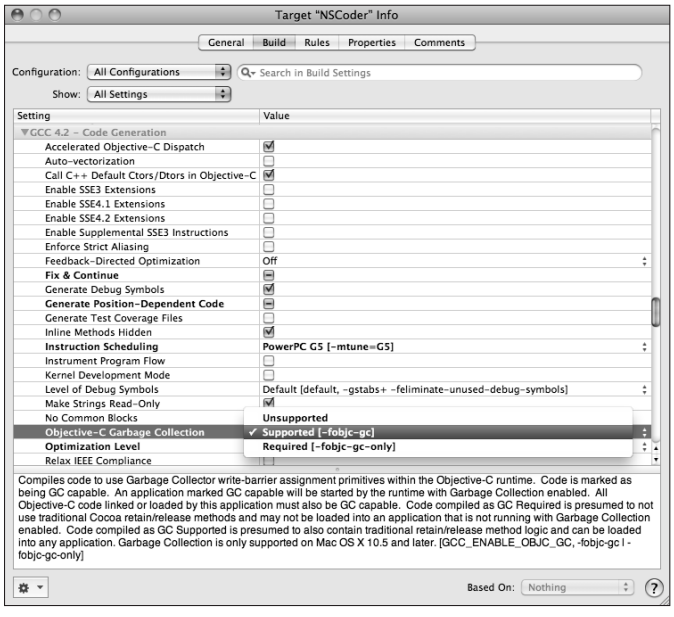
Earlier chapters introduced the idea of memory management. Cocoa is optimized for automatic memory management using a behind-the-scenes technology called *garbage collection*. Garbage collection automatically keeps track of items in memory and frees them when they're not being used. It "just works," and when it's enabled, you can mostly leave it to manage memory for you.

Using garbage collection

Examples in this book assume that memory is managed with garbage collection. The sample projects on the Web site are built with garbage collection, and they should work without changes. But when you create a new project, garbage collection is turned off. You must enable it by hand, as shown in Figure 12.7.

Figure 12.7

Enabling garbage collection, which is off by default, causing random crashes.



To find the Garbage Collection options, find the Targets item in the Groups & Files pane and right-click the name of your application under it. You'll see the dialog in Figure 12.7. Scroll down to the Code Generation section, and find the Objective-C Garbage Collection entry. Click in the selectable field next to the label, and choose the Supported option. You don't need to save the changes — your project is updated automatically.

CAUTION

You *must* enable garbage collection before you add code to a new project. If you don't, the compiler assumes that you're adding manual memory management features to your code.

CAUTION

If those features are missing, *your code will crash.*

Implementing manual memory management

You can choose to manage memory manually. This is usually a bad choice — except on the iPhone, where you're not given a choice at all. Manual memory management is notoriously difficult. The theory is simple, but the practice is unlikely to work as you expect, for reasons that can be caused by obvious mistakes or by subtle issues built into Cocoa classes.

The key feature of manual memory management is the *retain count*. Every object counts the number of times a retain method is called on it. Using *retain* or referencing the object increments the counter. Using *release* decrements the counter. When the counter is zero, the object is released from memory. Trying to access it again creates a crash. The essential memory management methods are shown in Table 12.4.

Table 12.4 essential Memory Management Methods

Method	*Description*
`[object assign];`	Duplicates a pointer, but doesn't change the retain count.
`[object retain];`	Increments the retain count.
`[object release];`	Decrements the retain count.
`[object autorelease];`	Hands the object over to an `autorelease` pool. This is risky — you never know when autoreleased items will disappear.
`object2 = [object copy];`	Calls the object's `copy` method. The results depend on the object's implementation of `copy`.

CAUTION

The `autorelease` feature looks like a simple fix for memory management issues — but it isn't, because it autoreleases objects on its own schedule, which may not match yours. To guarantee that an object is available when you need it, you must take part-ownership of it and `retain` it.

Manual memory management code typically follows this cycle:

1. `alloc/init` an object.
2. Use the object.
3. `release` the object.

The golden rule for memory management is that every reference is balanced by a `release`. For example, on the iPhone you create, display, and release an alert with the following:

```
*UIAlert *myAlert = [UIAlert initWith<list of properties>];
[myAlert show];
[myAlert release];
```

This seems simple, but it's important to understand that `release` doesn't immediately free the object. Instead, it tells the memory manager that your code is no longer referencing the object. In this example, iOS retains the alert until the user finishes with it — then it releases it. The final release happens out of sight, within the OS. Your release statement gives the OS permission to release the object. It doesn't necessarily trigger a release event.

To manage memory correctly, all objects must implement the following features:

- An `init` method that returns nil if not enough memory exists to create an object.
- Memory-aware setters.
- A `dealloc` method that is called in every object just before it's released, giving it a chance to release the objects it references.

Cocoa objects should implement these methods correctly internally. But because Cocoa objects are opaque, it's not possible to see what's happening inside them. Some classes hold onto objects when you're not expecting them to. In the worst case, it's possible to create a *retain cycle* where objects reference each other.

Creating a standard init method

The suggested boilerplate code for an `init` method is as follows:

```
-(init) {
if (self = [super init]) {
//Set up instance variables, if needed
}
return self;
}
```

Add this to every custom class, and use it. Some classes use `[super init]` to `release` the original object created by `alloc` and replace it with a new one. This code allows for that possibility, which happens rarely, though not rarely enough to ignore.

Creating a standard setter method

The suggested boilerplate for a memory-aware setter method for a property called `item` is:

```
- (void) setItem: (id) newItem {
    if (newItem != item) {
        [item release];
        item = [newItem retain];
    }
}
```

`@synthesize` implements this for you, but you should use this code in any custom setters you write. It looks like a complex solution to a simple problem, but it's the only way to guarantee that values are released correctly when they're updated.

A simpler setter is guaranteed to leak memory.

```
-(void) setItem: (id) item {
    //Don't do this
    item = newItem;
}
```

The original memory referenced by `item` is now lost. No pointer holds a reference to it, and it can never be released.

CAUTION

The `!=` conditional is pseudocode. In practice, you'll need to implement it with some variant of the standard `isEqual:` method.

Using dealloc

Add a `dealloc` method to every custom class you create.

```
-(void) dealloc {
    [super dealloc];
    [propertyOne release];
    …
    [lastProperty release];
}
```

This method defines what happens when your object is released. `[super dealloc]` allows a superclass to run `dealloc` recursively if it needs to. The individual `release` methods free the memory used by your class properties. If you don't implement `dealloc`, your custom class is unlikely to release memory correctly.

Using memory management heuristics

Finally, there are some heuristic rules that can help with memory management. These are unofficial and not a substitute for formally analyzing the structure of your code. But they can help solve a temporary memory problem while you get other parts of your code working. These are quick-fix, non-final suggestions and shouldn't be used in production code.

- Random crashes tell you that objects aren't being retained correctly.
- References don't persist across methods. If you create an object inside a method, you should `retain` it if you want it to be available in another method.
- Set pointers to `nil` after releasing them. Releasing a `nil` pointer doesn't cause a crash.
- Beware of delegate objects. Setting a delegate object can do unhelpful things to the retain count of some of its properties.
- Beware of Core Animation, which may not manage memory as you'd expect it to.

TIP

The Object Allocations instrument built into Xcode is invaluable for tracking memory errors. A robust application doesn't crash because of missing objects, and it doesn't leak memory because object references are being overwritten. See Chapter 18 for more on this instrument and other diagnostic tips.

Summary

In this chapter, you learned about data collection objects, memory management, and archiving methods. You explored the key features of `NSArray`, `NSDictionary`, and `NSSet`, and you discovered how to enumerate the items in a collection object, how to search items, and how to process each item with custom code.

Next, you discovered `NSCoder` and investigated how it worked with `NSKeyedArchiver` and `NSKeyedUnarchiver` to create composite object collections that could be saved to a binary file and reloaded and unpacked to restore original object properties.

Finally, you were introduced to practical garbage collection and to some of the less obvious challenges and possible solutions required to work with manual memory management.

13 Using Preferences and Bindings

In the same way that space is big, bindings are complex. Apple's documentation might not be considered a model of clarity, and it's difficult to find good information online from other sources, in part because bindings cause so much confusion to so many developers that even Cocoa experts have trouble with them. This chapter is designed to cut through the confusion that surrounds bindings and present them in a simple and clear way. To use bindings successfully you must understand:

- What bindings are
- How bindings relate to other technologies, such as properties, Key-Value Observing (KVO), and Key-Value Coding (KVC)
- How bindings are implemented in code
- How bindings are implemented in Interface Builder (IB)
- How bindings are connected to controller objects
- How bindings and controllers collect and transform values

Fortunately expert-level insight into each of these features isn't required.

13 In This Chapter

Understanding bindings

Using bindings with controllers

Implementing preferences with bindings

Creating and using value transformers

Understanding Bindings

Bindings are a step up from Key-Value Observing (KVO). They link object properties. When one property changes, the property or UI object at the other end of the binding also changes. But bindings add intelligence to KVO. With some care, it's possible to bind strings to numbers, and vice versa.

Updates can be read-only, with one object — such as a User Interface element in a view — observing and reporting the value of another. They can also be read-write; changing the value of one object automatically changes the corresponding value in another. When bindings are set up correctly, this happens without code.

Bindings are typically used in two ways:

- **In a simple application, they can take the place of outlets.** UI objects can be automatically connected to code objects without declaring any `IBOutlets` or linking them in IB. The binding links the two objects, with optional added intelligence. For example, a text field can display a number value automatically, without explicit conversion.
- **In a more complex application, bindings are used as data pipes that synchronize blocks of data.** Bindings can synchronize arrays, dictionaries, and other complex data types; for example, a drop-down list can be populated automatically by binding it to a data source. Even more usefully, they can synchronize the *editing* of data, making it possible to add, update, and remove data with a simplified UI and minimal supporting code.

Confusion arises because the Apple documentation implies that bindings require controller objects, specifically `NSObjectController`, `NSArrayController`, `NSDictionaryController`, `NSTreeController`, and `NSUserDefaultsController`.

You can use bindings without these objects. You can bind any object to any other as long as both support KVC and KVO. For simple applications, adding a controller adds an unnecessary layer of confusion and complication. Table 13.1 summarizes other common misunderstandings about bindings.

Table 13.1 Facts and Fiction about Bindings

Fiction	*Fact*
Bindings require a controller object.	You can use bindings without a controller.
Bindings "just work."	Bindings work as long as you always access values through accessor methods *and* bind compatible objects to each other. Simple assignments don't trigger the KVO mechanism.
Bindings are always two-way.	Some bindings are read-only. Displayed values may not be editable.
Bindings are designed to work with Interface Builder.	You can create bindings programmatically without using IB.
Bindings eliminate unnecessary code.	Bindings can eliminate pages of code in larger projects. For smaller projects where bindings are used in an outlet-like way, the benefits may be less obvious.

Getting started with bindings

To use bindings successfully, keep these requirements in mind:

- **Bindings work with object properties.** You can't bind to a private variable.
- **Bindings rely on KVO.** All properties and assignments must be KVO-compliant.

Using accessors

As outlined in Chapter 9, you *must* use accessor methods when setting the values accessed by bindings.

```
value = value*10;        //This doesn't work
self.value = value*10;   //This does - it's obligatory
```

This is a prime source of confusion. If you don't update values correctly, bindings don't work.

TIP

You can bind to system objects as well as to objects in your own code. If their properties are KVO-compliant, bindings will track them as their values change. If they're not KVO-compliant, nothing happens. Class References occasionally mention that properties are KVO-friendly. If they don't, you can test them by trying to set up a KVO observer.

Creating a simple binding

The SimpleBindings1 project on the Web site for this book demonstrates how to bind a counter value to a slider object. You can find the project at `www.wiley.com/go/cocoadevref`. Load the project before continuing. All the code is in the app delegate. The header defines a counter as a property:

```
#import <Cocoa/Cocoa.h>
@interface SimpleBindingsAppDelegate : NSObject
   <NSApplicationDelegate> {
    NSWindow *window;
    int sliderCount;
}
@property (assign) IBOutlet NSWindow *window;
@property int sliderCount;
@end
```

The implementation creates a timer, which fires a `timerMethod`. `sliderCount` counts to a maximum value, reverses direction, counts down, and reverses direction again at 0.

```
#import "SimpleBindingsAppDelegate.h"
@implementation SimpleBindingsAppDelegate
@synthesize window, sliderCount;
int delta = 1;
- (void)applicationDidFinishLaunching:(NSNotification *)
   aNotification {
   NSTimer *thisTimer = [NSTimer
   scheduledTimerWithTimeInterval:0.05 target: self selector:
   @selector(timerMethod) userInfo: nil repeats: YES];
}
```

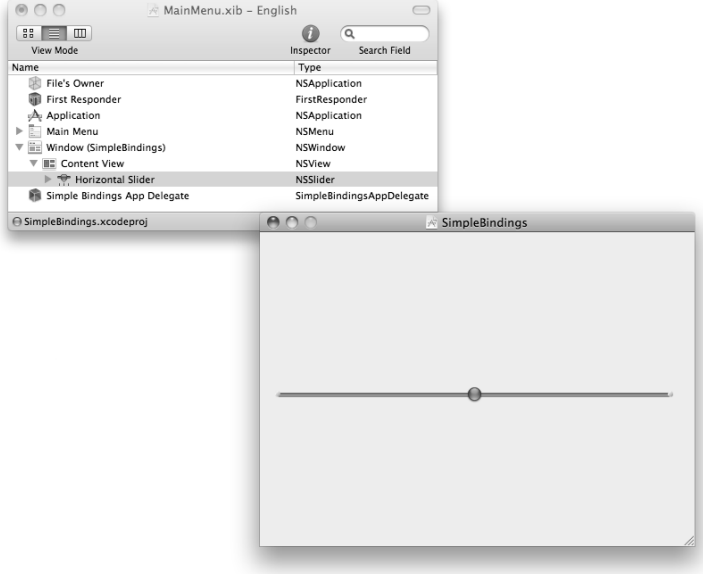
```
-(void) timerMethod {
  self.sliderCount += delta;

  if (sliderCount > 100)
  delta = -1;
  if (sliderCount < 0)
  delta = 1;
  //NSLog(@"%i", sliderCount);
}
@end
```

If you run this code and uncomment the `NSLog` line and view the console, you'll see `sliderCount` cycling up and down.

Figure 13.1 shows the project nib file. It's identical to the standard blank template, with an added slider. Note that no outlets or links have been defined for the slider.

Figure 13.1

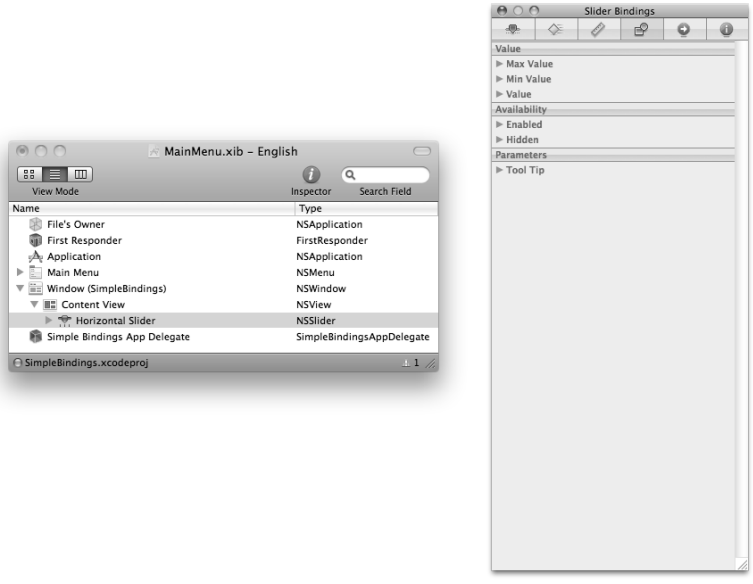
Adding a slider. You'll use a binding to link the slider to the `sliderCount` property.



To create a binding, select the slider in the Doc window and select the Bindings tab; it's the one with two green objects, third from the right. Figure 13.2 shows how the window appears when no bindings are selected.

Figure 13.2

Displaying the slider's bind-able properties



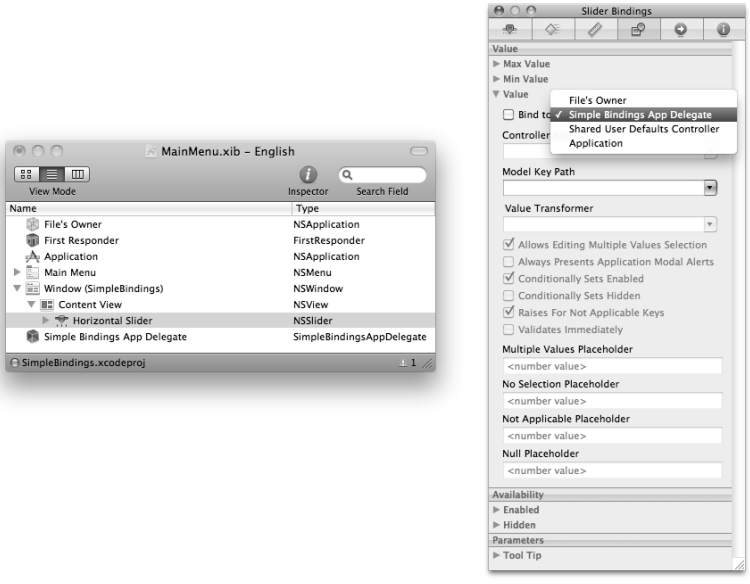
Before you create a binding, take a closer look at this pane. It's displaying a list of some of the slider object's properties. If you review the `NSSlider` Class Reference, you'll see these items in the properties list. When you bind to one of these properties, you link it to a value in a different object.

Outlets can connect to any property, but bindings only support a subset of object properties. If a property doesn't appear in this list, you can't bind it.

To define the other object, select the slider's value property. Click on the pop-up menu next to the Bind to: check box. You'll see a list of objects in the nib, as shown in Figure 13.3.

Figure 13.3

Displaying a menu of possible objects to bind to



Select the Simple Bindings App Delegate, as shown. Select the Bind to: check box next to it. This tells IB that you are binding the slider's Value property to one of the properties of the app delegate.

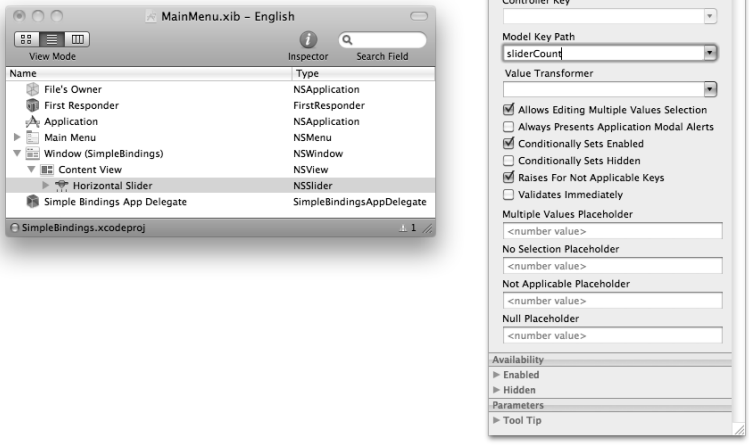
You select that property by typing its name into the Model Key Path pop-up menu, as shown in Figure 13.4. In this example, there's only one possible property to bind to — `sliderCount` — so that's what you type here.

There are two possible sources of confusion in this step. The first is that the Model Key Path combo box usually defaults to `self`, which is meaningless, unhelpful, distracting, and just plain wrong.

The second is that the pop-up menu doesn't show a list of valid properties, even though it could, and should. Bindings would be much less confusing if this feature were available. It would underline the similarities with the outlet-method-linking system elsewhere in IB. Because this feature isn't available, you have to type in the name manually instead of selecting it from a list of possible properties.

Figure 13.4

Selecting a property in the object selected in Figure 13.3



Here's a recap of the stages so far:

1. **Add an object to the nib.**
2. **Select the Bindings pane.** Pick the property you want to bind to. The list in the Bindings pane is final and non-negotiable; if a property doesn't appear here, you can't bind to it.
3. **Choose the object at the other end of the binding from the Bind to: pop-up.**
4. **Select a property in that object by typing its name into the Model Key Path combo box.**
5. **Save the nib file, Build and Run.**

TIP

It's useful to commit this sequence to memory, and then work through it over and over with other examples until it becomes second nature. You'll need this initial level of understanding to use controller objects, described later in this chapter.

Figure 13.5 shows the result. The slider moves from side to side, in an automated way.

Open the console window, and move the slider with the mouse. You'll see that the value of `sliderCount` automatically updates itself when you release the mouse.

The binding is bidirectional. When the code sets `sliderCount`, the slider follows it. When the user moves the slider, `sliderCount` is updated with the new slider value.

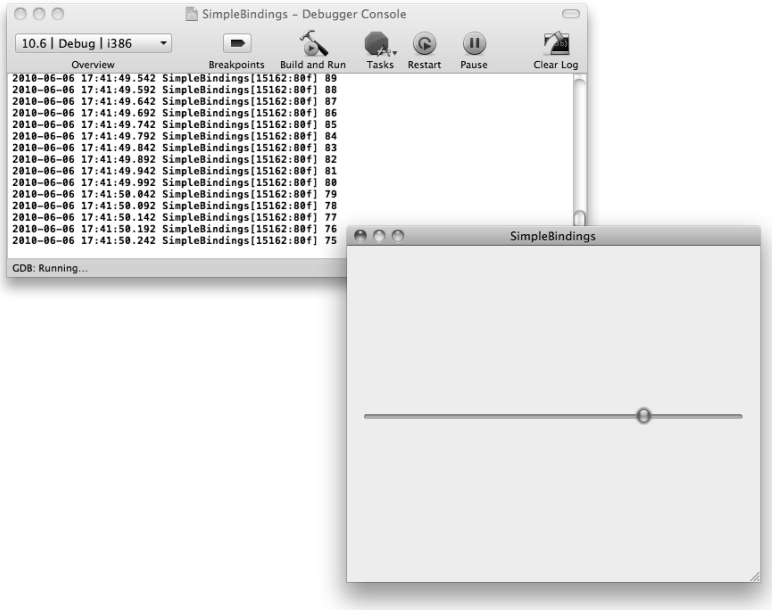
In a typical application, `sliderCount` — or some other property — wouldn't be set by a timer. Bindings are valuable because the `timerMethod` could be replaced by a mouse tracking method, a disk monitor method reporting free space, or a value pushed from the Internet. As long as there's an accessor assignment, a binding can display it.

TIP

See if you can use the delta value in the code, or an associated Boolean property, and the Enable binding to enable the slider while the count is increasing and disable it while it's decreasing.

Figure 13.5

The binding links the slider's value property to `sliderCount` in Simple Bindings App Delegate. As the count changes, the slider follows it.



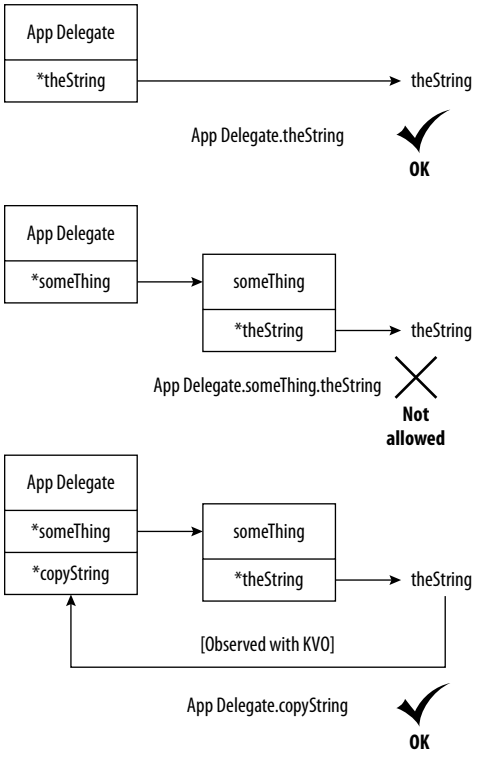
Working around keypath limitations

The model keypath feature suggests you should be able to access subproperties. For example, the app delegate has a property called `someThing` of a class with an internal property called `aString`, as shown in Figure 13.6. You might think `someThing.aString` would be a valid model keypath.

It isn't. This idiom is valid in some of the various object controllers, but it doesn't work with simple bindings. If you want to bind to subproperties, you have to use KVO to copy a property value to the top-level object after an update, and bind to the copy.

Figure 13.6

Working around keypath limitations. This is an ugly hack, but the model keypath can't usually bind to subproperties directly.

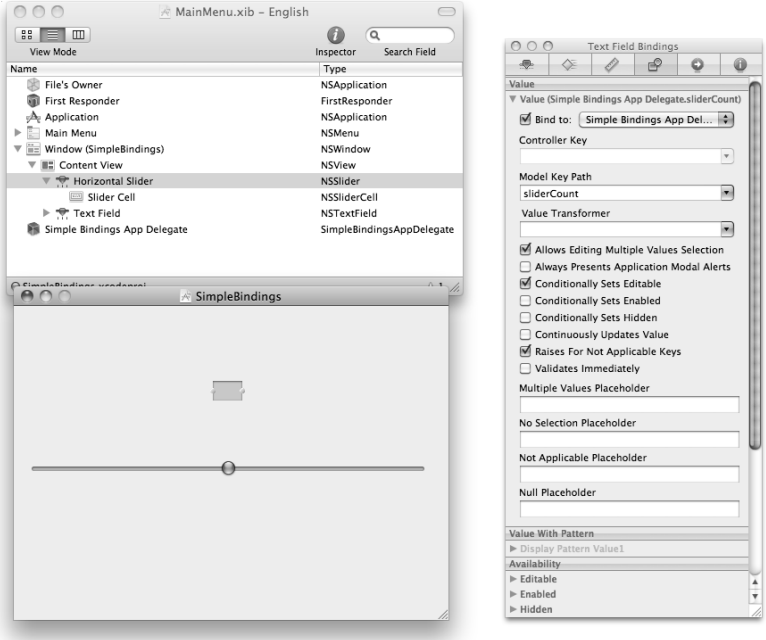


Binding incompatible objects

The nib for a slightly modified version of the same project is shown in Figure 13.7. A text field has been added and bound to the `sliderCount` property, as before.

Figure 13.7

An extended version of the SimpleBindings application, with a text field that can display the slider value numerically.



The result is shown in Figure 13.8. The slider continues to move, and the text field displays its value.

This looks like a trivial change, but a lot is happening behind the scenes. The string in the text field is displaying an `int` value without code. *Bindings don't just link two properties; they can also translate values between different data types.*

Typically, converting an `int` to a string requires extra formatting code:

```
NSString *sliderString =
[NSString stringWithFormat: @"%i", sliderCount];
```

Bindings perform a selection of the most useful translations automatically. There's no need for extra code or for explicit format specifications. Figure 13.9 illustrates this graphically.

Figure 13.8

An extended version of the SimpleBindings application, with a text field that can display the slider value numerically

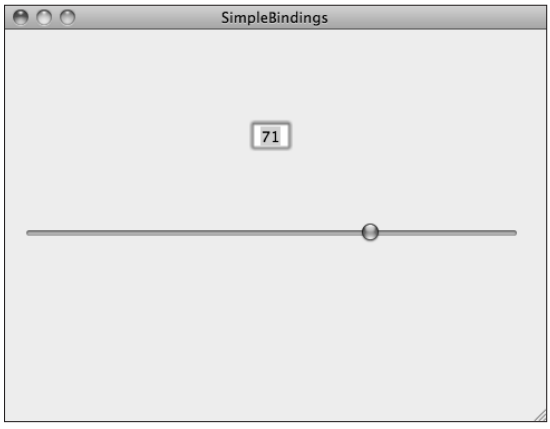
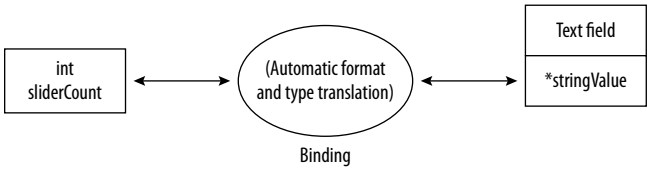


Figure 13.9

Don't think of bindings only as property links; they can also interconvert data types.



If you open the Console window, type a number into the text box, and press Return, you'll see that this binding is still bidirectional; it sets `sliderCount` and the slider position. The type conversion is intelligent enough to work in both directions.

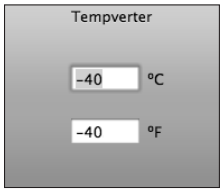
Using bindings to manage interactivity

You can use a combination of bindings and KVO to create semi-automated interfaces. Sometimes UI properties depend on each other. For example, selecting one UI feature can disable or enable others, or modifying a value can update its dependent values.

With care it's possible to create UIs with mutual dependence; the user can change any element, and all related elements update themselves automatically. Figure 13.10 shows a very simple example: a two-way temperature converter micro-application. Typing a number into either text field and pressing Return generates an updated value in the other.

Figure 13.10

Tempverter: a very simple temperature conversion application, implemented with KVO and bindings



It's possible to solve this problem with outlets and the text field's delegate methods, using `(id) sender` to discover which text field was changed, and adding code to update the other. Bindings not only make it easy to manage multiple dependencies, but they also simplify the code. The two temperature values can be floats. Bindings convert floats into text for display, and they convert text into floats for user input. There's no need to add formatting code or supporting intermediate values.

In a more complex application, this can be extended to create multiple propagating dependencies with very little code. But to create a complex UI, you need to understand the key features of a simple one.

The nib file for the project is shown in Figure 13.11. The nib is a standard content view with four text fields. Two are static labels; the other two are used for temperature input and display.

NOTE

Figure 13.11 includes a view of the Window Attributes. The window size is fixed by deselecting the Resize control. Close and Minimize have also been disabled. Texture and Shadow effects are applied. These features don't affect the application's operation, but they do improve its appearance.

The header file for the App Delegate is shown below:

```
#import <Cocoa/Cocoa.h>
@interface TextFieldAppDelegate : NSObject
   <NSApplicationDelegate>
{
    NSWindow *window;
    float celsFloat;
    float fahrFloat;
}
@property (assign) IBOutlet NSWindow *window;
@property float celsFloat;
@property float fahrFloat;
@end
```

Figure 13.11

The Tempverter nib file. There are four text fields.

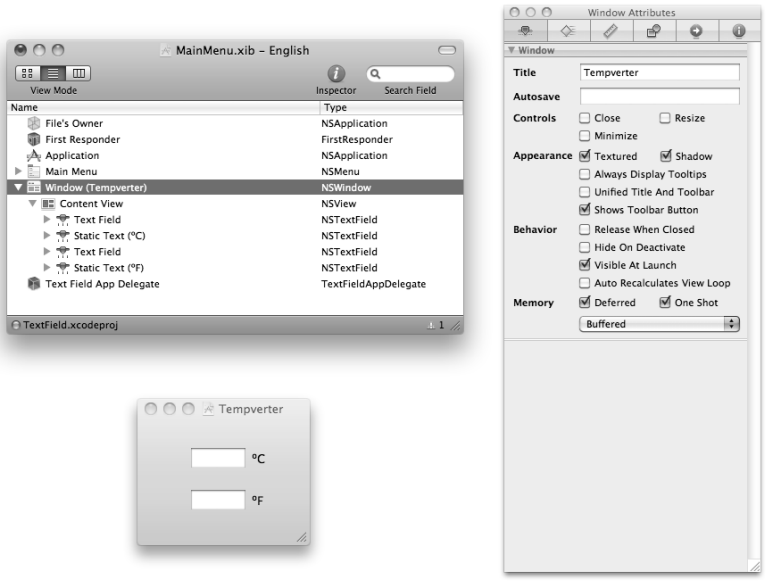


Figure 13.12 illustrates how the bindings are assigned. As in the previous example, the bindings select an object — the App Delegate — and the Model Key Path names a property in that object.

This is enough to create a two-way link between the text field and `celsFloat`. If the user types a number into the Celsius text field and presses Return, it's converted into a float value and is copied to `celsFloat`. If the application writes a new value to `celsFloat`, it's converted into a string and copied to the text field.

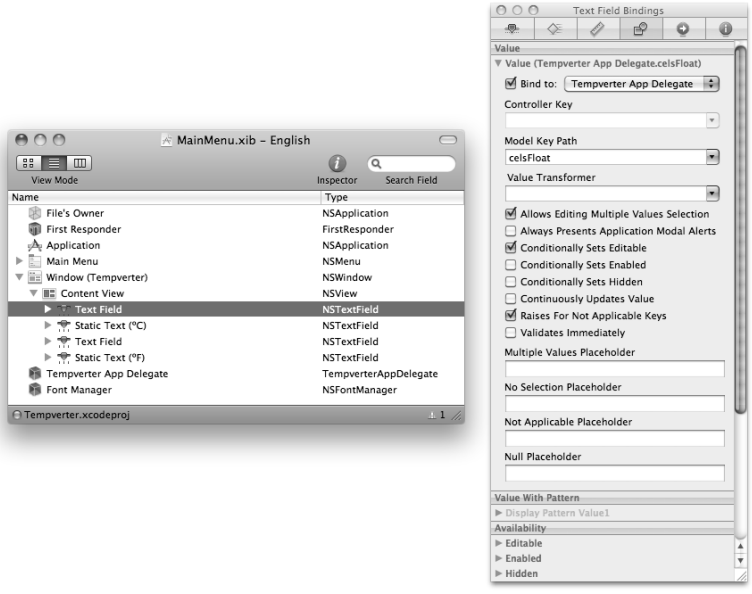
Adding a corresponding binding between the Fahrenheit text field and `fahrFloat` is almost enough to create a simple application. There are still two features missing: the temperature conversion code and KVO management to trigger dependent updates.

NOTE

Internally, the bindings use the `stringFromFloat:` and `floatValue` methods built into `NSString`. Text is interpreted as an error and produces a float value of 0.

Figure 13.12

Binding the Celsius text field to the corresponding float in the App Delegate



Using KVO to manage bindings

Converting one number into another is almost easy. You add a KVO observer to both floats. When the user types in a number, the binding updates the corresponding float. This triggers the KVO observer method. You can do some simple arithmetic in the observer, and then write the converted value to the other float … which triggers its KVO observer, which does some arithmetic on the other value … and loses itself in an infinite loop.

It would be possible to enable and remove KVO selectively, ignoring each value as it's being updated. In this example you've created a blanket method that turns off KVO for both floats whenever they're being updated, and re-enables it when the update has completed.

A simple KVO switch would be useful, but Cocoa doesn't have one. KVO can only be enabled and disabled separately for each possible key. This puts some practical limits on KVO, which can't usually observe more than a few tens of properties. Trying to observe more results in code can be difficult to debug and can also create a significant performance hit, because the technology used to implement KVO isn't fast or efficient.

NOTE

Behind the scenes, KVO works by using a trick called `isa swizzling` — a fancy way of saying that classes are copied and modified on the fly, and updated versions impersonate the originals. This isn't a speedy process.

So in this example, you explicitly enable and disable KVO for all properties with a method called `isObserving:` in the App Delegate. The method is listed below:

```
#import "TempverterAppDelegate.h"
@implementation TempverterAppDelegate
@synthesize window, celsFloat, fahrFloat;
- (void)applicationDidFinishLaunching:(NSNotification *)
   aNotification {
   self.celsFloat = 100.0;
   self.fahrFloat = 212.0;
  [self isObserving: YES];
}
- (void) isObserving: (BOOL) observing {
if (observing) {
   [self addObserver:self forKeyPath:@"celsFloat"
   options:NSKeyValueObservingOptionNew context:NULL];
   [self addObserver:self forKeyPath:@"fahrFloat"
   options:NSKeyValueObservingOptionNew context:NULL];
} else {
   [self removeObserver: self  forKeyPath:@"celsFloat"];
   [self removeObserver: self  forKeyPath:@"fahrFloat"];
  }
}
- (void) observeValueForKeyPath:(NSString *)keyPath ofObject:(id)
   object change:(NSDictionary *)change
   context:(void *)context {
   NSLog(@"%Keypath:%@ Value: %@", keyPath,
   [self valueForKey:keyPath]);
   [self isObserving: NO];
   if ([keyPath isEqual: @"celsFloat"])
      self.fahrFloat = 32+celsFloat*1.8;

   if ([keyPath isEqual: @"fahrFloat"])
      self.celsFloat = (fahrFloat-32)/1.8;

   [self isObserving: YES];
}
@end
```

Key features of the code include:

- `applicationDidFinishLaunching:` sets initial defaults for both temperatures and then turns on KVO.
- The `isObserving:` method takes a `BOOL` and adds and removes observers for the two temperatures dynamically.
- The standard KVO `observeValueForKeyPath:` method is triggered when either float changes — but only if KVO is enabled. It logs the trigger event, turns off KVO to prevent a loop, updates one of the floats, running code chosen by testing the keyPath, and turns KVO on again after the update.

It's important to understand that bindings and KVO act independently. You can turn off KVO without affecting bindings, because the bindings manager is a separate object. Behind the scenes, out of sight of your code, it's running its own KVO monitoring. It doesn't care how you use KVO elsewhere.

NOTE

Don't forget to prefix properties with `self` to make them visible to KVO and bindings.

Using formatters

The disadvantage of the automatic conversion in the bindings manager is that it doesn't support format control. If you type values into the Fahrenheit box, you'll see that the equivalent Celsius value is often a recurring decimal. The text field truncates this visually. A wider text field would show them.

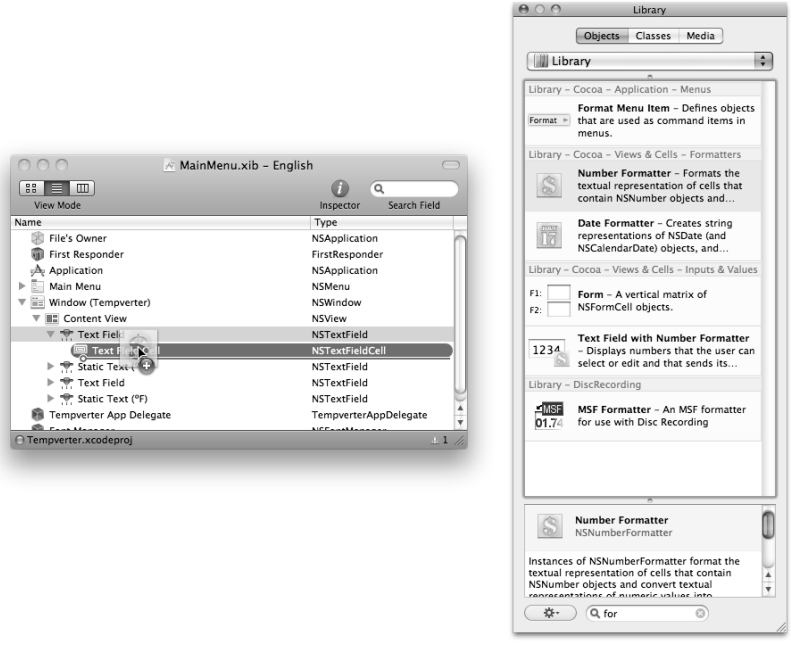
`NSNumberFormatter` is a simple drop-in formatting object that can solve this problem. It's not directly related to bindings, but it's often used to fine-tune the format of the strings produced by the bindings manager. Figure 13.13 illustrates how to add an `NSNumberFormatter` to a text field.

CAUTION

The formatter works on a text cell, so you must drop it on the `NSTextFieldCell` in the nib, not on the text field itself.

Figure 13.13

Adding a number formatter to a text field to tidy up the string representation of the bound float.

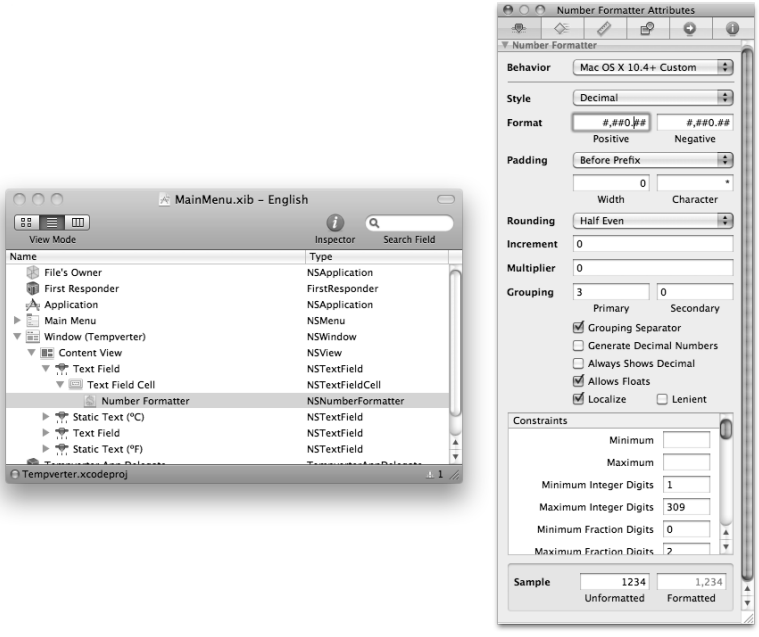


`NSNumberFormatter` has many options and can perform unusual tricks. For example, it can convert numbers into text strings such as "ten" or "fifty three," with associated local language support. In this example, you'll use the settings shown in Figure 13.14 to limit the text field to two decimal places. The two Format text boxes near the top of the Number Formatter Attributes pane set the format, and the number of hash signs after the decimal point sets the number of decimal places. You must select the Mac OS X 10.4+ Custom Behavior at the top of the pane to see the Format options.

To complete the application, add another formatter with the same settings to the Fahrenheit text field.

Figure 13.14

Setting two decimal places in the Format boxes. You can use a formatter to define scientific and financial formats, with optional special characters, including currency signs.



TIP

You can use the associated `NSDateFormatter` objects to control the format of date values.

Using Bindings with Controllers

Controller objects — `NSObjectController`, `NSArrayController`, `NSDictionaryController`, `NSTreeController`, and `NSUserDefaultsController` — add an extra level of sophistication to bindings, and four or five extra layers of complexity. Controllers become useful when bindings are used as data pipes. In the same way that simple bindings can connect different properties without formatting or conversion code, controllers can connect complex objects and translate data between them.

You can use controllers to produce complex applications with very little code. For example, `NSTableView` is the standard Cocoa object used to display and manage tables. It's a complex class with many features and delegate methods. To implement a TableView without bindings, you must write *glue code* — code that connects two objects or features together and translates data between them. For example, an array can be translated into a list of values. The code is embedded in delegate methods that are called when the table needs to display, refresh, or modify data. It can easily become very complex.

Controller objects can minimize the glue code. In some applications, they can eliminate it completely. To demonstrate, the example below demonstrates how you can create an application that displays a list of running processes in OS X — with exactly one line of active code.

The App Delegate header file looks like this:

```
#import <Cocoa/Cocoa.h>
@interface TableBindingAppDelegate : NSObject
   <NSApplicationDelegate> {
    NSArray *runningApps;
    NSWindow *window;
}
@property (assign) IBOutlet NSWindow *window;
@property (nonatomic, retain) NSArray *runningApps;
@end
```

It's a copy of the standard template, with an added `NSArray` property called `runningApps`.

The App Delegate implementation looks like this:

```
#import "TableBindingAppDelegate.h"
@implementation TableBindingAppDelegate
@synthesize window, runningApps;
- (void)applicationDidFinishLaunching:
      (NSNotification *)aNotification
{
   self.runningApps =
     [[NSWorkspace sharedWorkspace] launchedApplications];

   NSLog(@"%@", runningApps);
}
@end
```

This code doesn't do much. It calls the `NSWorkspace` object and asks for a list of running applications, which is copied to the array. An `NSLog` call lists the running apps to the console. The logging is optional, but it's very useful, as you'll see next.

Figure 13.15 shows the finished application. It has the following features:

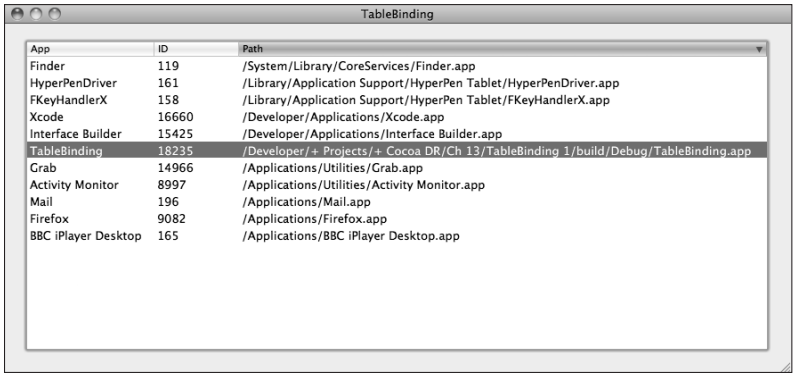
- There are three independent columns of data showing values from the `runningApps` array.
- The table dividers can be moved.
- Rows can be selected by clicking on them, and there is an optional selection index used to return information about the number of the current selection.
- Columns can be sorted by clicking on each column header to select ascending or descending sorting.

The application has some limitations. It doesn't auto-refresh the table; the array is loaded once at launch, and isn't updated. In its basic state, it's a display-only app. There are no features for starting and stopping apps, saving the list to a file, maintaining a log file, and so on.

But it's doing a lot with very little code. And most of the intelligence is built into a controller object, coupled to the array and the table view via bindings.

Figure 13.15

Filling a table view with bindings. Most of the active code is built into a controller object and runs automatically.



Adding a controller object

Figure 13.16 shows the project nib file. The table view automatically adds a surrounding scroll view. The default table view has two columns. Another column has been added by copying and pasting it. Each column's name — the text string that appears in the divider, at the top of the header — is set in the Attributes pane in Inspector. The Size pane sets the default width and optional width limits. Take some time to explore these features before continuing.

Figure 13.16 shows the list of controller objects in the Library window. In this example, you're using an array as a data source, so you add an instance of `NSArrayController` to the project by dragging it from the window.

Remember that adding an object to a nib automatically generates an instance in memory when the nib loads. You don't need to alloc/init the controller in your code — it's already loaded from the nib.

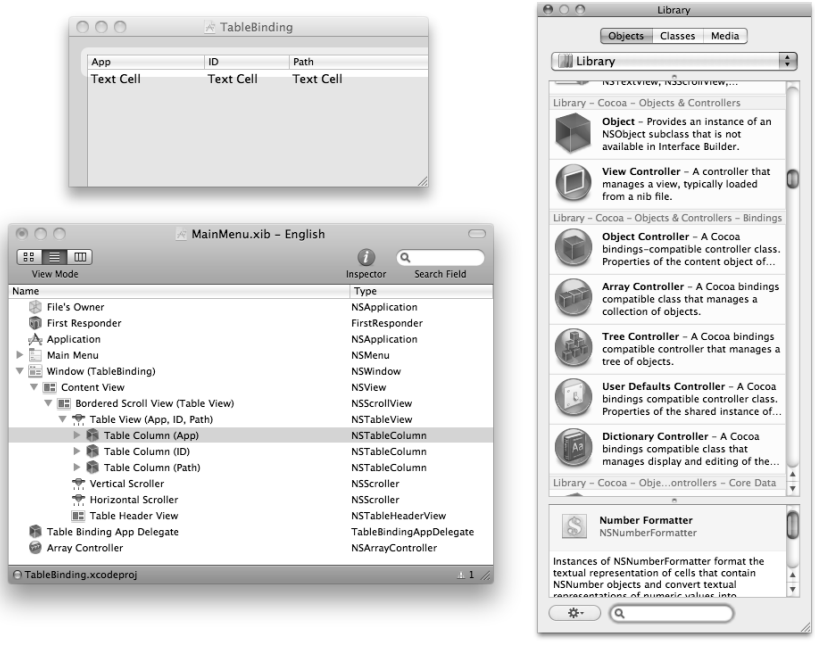
Optionally, you could subclass it to access its internal features, adding a supporting subclass — perhaps called `ArrayControllerSubclass` — and using IB's identity tab to reclass the default controller as your subclass. You don't need to do that here. Instead, you'll discover how to use the new controller.

TIP

When you add a column, its default width may be so wide you can't see it, because it has fallen off the right-hand edge of the table. Setting the widths manually can recover it. Once a column is visible, you can set its width by dragging it with the mouse. This is a tricky process: you'll need to click on the header tab between two and four times, depending on the table's selection state.

Figure 13.16

Filling a table view with bindings. Most of the active code is built into a controller object and runs automatically.



Setting up the controller's data source

Controllers act as a bridge object between a source of data and a view object. To use a controller, you bind it twice. The first binding selects a data source for the controller. It defines the controller input, although, unfortunately, Apple doesn't call it that — controllers would be easier to understand if this relationship were clearer.

The second binding links a view to the processed data generated by the controller — in other words it displays the output, although again Apple's naming scheme doesn't make this obvious.

Defining a controller data source is as easy as creating any other binding. You select a source object, choose a property in that object, and type the property name into the Model Key Path combo box.

A key source of confusion is the cryptic nature of controller properties. Controllers are general-purpose objects, designed to work with a variety of data sources. The controller doesn't care if your array holds application statistics, ages, salaries, or a list of probabilities of Earth impact for every major asteroid.

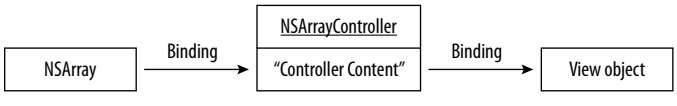
Instead, the controller properties *generalize* the data connection. So `NSArrayController` has a property called Controller Content. This is a general wrapper object that holds the same content as the data source. In the same way that File's Owner is a placeholder for the object that created a nib, think of Controller Content as a placeholder for your data source. When you bind to the controller to display data from it, it *becomes* the data source. It's then translated, through binding magic, into a format that a compatible view object can display without code. Figure 13.17 shows how this works.

CAUTION

Controller Content is a property of the `NSController` superclass. All the controller objects support it. But in an array controller, the content is an array. In a dictionary controller, it's a dictionary, and so on.

Figure 13.17

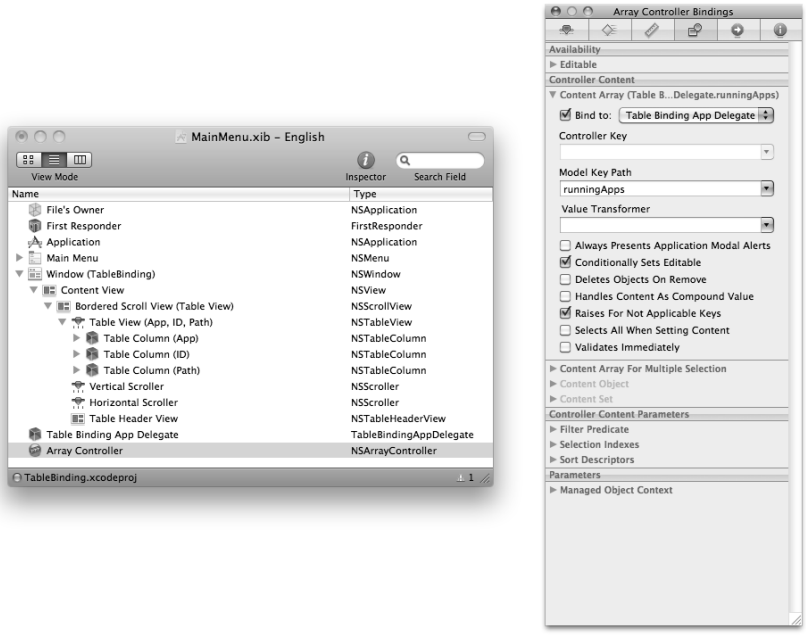
Using a controller object as a two-way binding. The Controller Content property is a placeholder for data from the source object.



Now that you've been introduced to the Controller Content property, it's easy to see how to connect the array data to the controller. Selecting the Controller Content tab, you bind it to your data source: the `runningApps` array in the App Delegate, as shown in Figure 13.18.

Figure 13.18

Binding to the Controller Content property of the controller, so that it uses the `runningApps` array as a data source



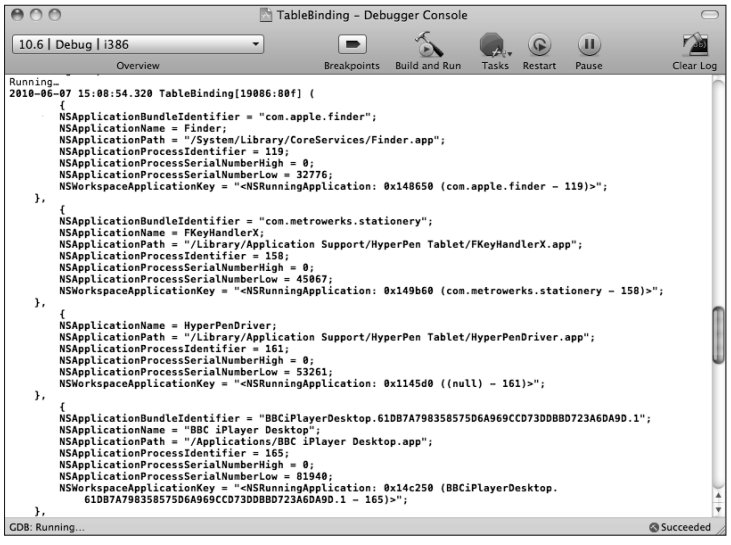
Reading data from the controller into a view

Reading translated data from the controller is more difficult. Instead of following through with the Controller Content metaphor, this side of the process uses different properties. Effectively, you're going to look inside the Controller Content object and pull out various slices through the data.

The first — critical — point is that it's only possible to pull data out of the array if its objects support key-value property access. Figure 13.19 shows a log of the contents of the `runningApps` array. Usefully, this array is a collection of dictionary objects, one for each running app. It's easy to pick out the relevant keys from the log; for example, `NSApplicationName` is the application name. However, not all objects support this kind of access. If they don't, you can't use bindings to display their data.

Figure 13.19

Understanding the format of the data in the `runningApps` array. Every item is a dictionary.



When using bindings, this point is crucial. You *must* be able specify a keypath for each displayed property, and it must return a useful value. If you want to use bindings with custom objects, you must design them in a way that supports keypath value access. Dictionaries are ideal for this because they explicitly implement key-value access.

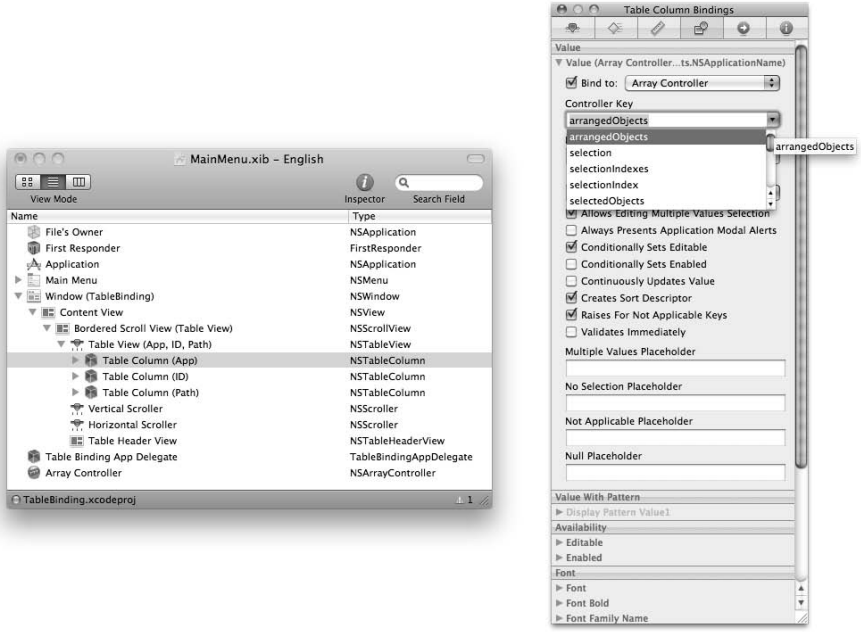
You can't, however, access a simple linear array of strings, because the strings won't have an associated key. An array index isn't enough to support bindings, and bindings don't support index access. You can't use `someArray.0` as a keypath.

This complicates the design of applications, because typically you need to pack array data into a dictionary before it can be accessed. Other workarounds are possible, but wrapping content into a dictionary is often the simplest solution.

In this example, dictionaries are already available. So you can move on to the second part of the problem: selecting the formatted data. Figure 13.20 shows the Controller Key combo box. This is another element that seems mysterious, but it is quite simple in practice.

Figure 13.20

Listing the possible Controller Key options to select how you want to view the data in the source array



Understanding controller keys

In the same way that the controller abstracts data from the data source, it also abstracts its display. Each item in this combo box is one possible interpretation or view of the data copied to the controller. Again, these items aren't related to specific named properties in the original data, and the controller takes no interest in what the data represents or how it's named.

Instead, these Controller Keys select one particular summary or element of the data. Table 13.2 describes the more useful items in this list.

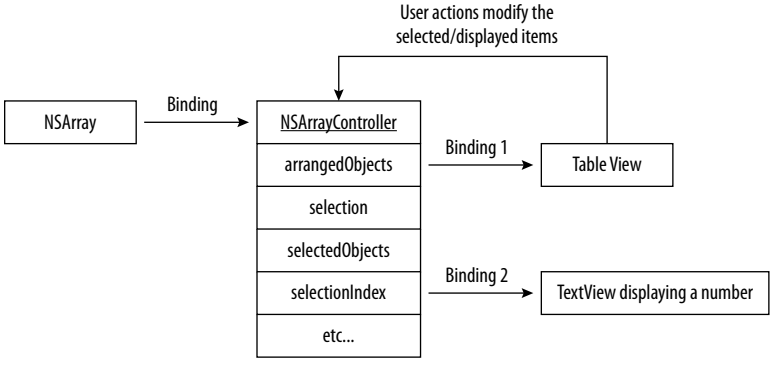
A key feature of all controller keys is that the data selected by these keys is determined by the user's actions. The datasource array doesn't change, but the view of the data is modified as the user interacts with the table view or other display object. Figure 13.21 illustrates the process graphically.

Table 13.2 Some common Controller Keys

Name	*Description*
`arrangedObjects`	An array holding a sorted copy of the source data. When the user changes the sort options in the table view, this array is re-sorted and updated automatically.
`Selection`	A copy of the current selected object, updated when the user selects an object. (Use keypaths to extract values from it.)
`selectionIndexes`	An `NSIndexSet` of currently selected objects, if there are multiple selections.
`selectionIndex`	An `NSUInteger` index of the currently selected object. (Invalid for multiple selections.)
`selectedObjects`	An array of the objects that are currently selected.

Figure 13.21

When binding to visible objects in the UI, the controller can display various interpretations of the data, including selected elements chosen by the user.



TIP

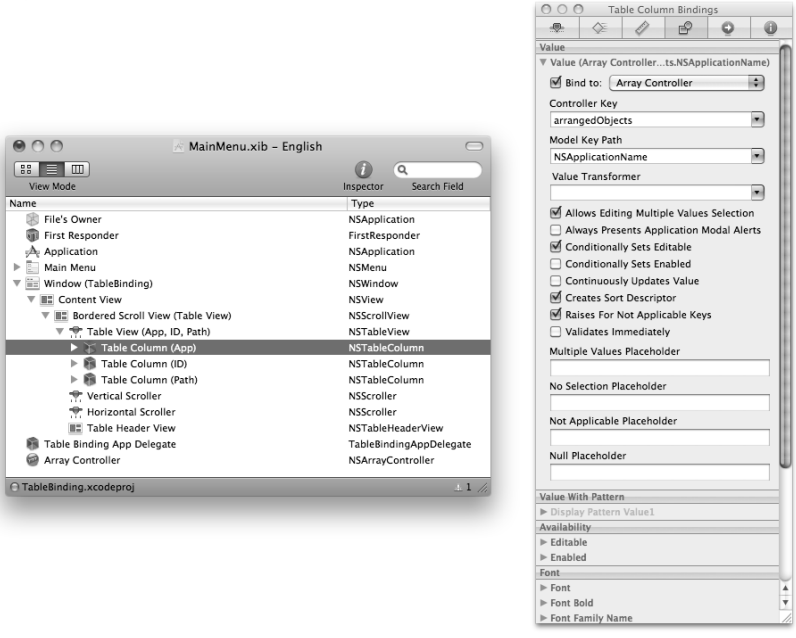
Lower down the list, you'll see keys like canInsert. You can use these to enable and disable editing features. Creating an editable table is more of a challenge, and some extra code is needed. Typically you'll subclass the data source array, the controller, or both to implement full editing.

Selecting controller keys

Now that you know what controller keys do, it's easy to select the one you need to bind a table column to a slice of data from the `runningApps` array. Figure 13.22 illustrates how to do this. You select arrangedObjects to display a sorted list of apps in the array, and then choose the `NSApplicationName` key to retrieve the name string of each app.

Figure 13.22

Selecting a Controller Key and an associated Model Key Path to copy values from the data source array into a table view



When the application runs, the binding manager automatically accesses each running application in turn, pulls out the value associated with the `NSApplicationName` key, and adds it to the table column, creating the list shown previously in the left-hand column in Figure 13.15.

You can repeat this for other properties in the other two columns. In this example, the middle column displays the `NSApplicationProcessIdentifier` key, and the rightmost column displays the `NSApplicationPath` key.

Here's a review of the steps that created the finished application:

1. **You added code to initialize a data source with data.**
2. **You added a controller object to the nib.**
3. **You bound the controller object to the data source, copying data to its Controller Content property.**
4. **You added view objects to the nib.**
5. **You bound the view objects to view content generated by the controller, selecting different possible view content with the Controller Keys.**
6. **Finally, you selected the values that appear by typing a property name into the Model Key Path.**

TIP

You can find two versions of this project on the Web site for this book at `www.wiley.com/go/cocoadevref`. A slightly extended version adds an extra text field that you can experiment with. Try binding selection, selectionIndex, and some of the Boolean values to the text box to see how they change as you select items in the table.

Implementing Preferences with Bindings

Among the controller objects used with bindings is `NSUserDefaultsController`. This is a special object that takes its data from the application's preferences. The controller is shared. You can create multiple instances of it in multiple nibs, and every instance accesses the same data. You don't need to bind it to a data source, because it's already bound to the preferences system.

Bindings can simplify preferences in two ways:

- **You can bind objects and views in a preferences pane directly to the** NSUserDefaultsController**.** The controller will save and load the application preferences automatically, so this pane will always be correct.
- **You can bind objects in the main application nib to** NSUserDefaultsController **to display them, or a modified version of them, in a different location.** For example, you may want to include a hint or reminder item in a toolbar that shows the current value of a preference.

A full preferences implementation should read settings from the code version of `NSUserDefaultsController`, which is called `NSUserDefaults`, and is used like this:

```
NSUserDefaults *thePrefs = [NSUserDefaults standardUserDefaults];
```

You can then use key-value access to read the defaults. More creatively, you can also use Key-Value Observing (KVO) to monitor changes to preferences, updating selected settings immediately. Poor implementations of preferences force the user to close and re-open an application before changes are registered by the rest of the application. A more professional implementation can use KVO and bindings to respond to changes as they happen.

CAUTION

If you're using bindings and a separate preferences window, it's best not to set the defaults by setting the values in `NSUserDefaults` with code. Limit changes to the preferences UI and its bindings, and use code elsewhere in the application to read values from `NSUserDefaults`, or track changes to them with KVO.

Understanding preferences

The Cocoa preferences system is unusual. Preference keys and values must be *registered* before they can be used. Internally, the preferences system keeps a copy of the original registered values and only saves changes to them.

Initializing preferences

To register keys, run an *initialize* method inside a preferences method or in your application delegate. `initialize` — not to be confused with `init` — is the very first method run by every class as the application loads. You can use it to run start-up code that initializes critical values. A typical preferences method that uses `NSUserDefaultsController` looks like this:

```
+ (void) initialize {
   NSMutableDictionary *prefsDictionary =
        [NSMutableDictionary dictionary];

   [prefsDictionary setObject: <an object> for Key: <a key];

   [[NSUserDefaultsController sharedUserDefaultsController
   setInitialValues: prefsDictionary]];
}
```

Add one `setObject:` line for every item in the preferences file.

NOTE

`initialize` is a class method.

Including this code defines the names and keys used in the preferences data. You can read and write the keys elsewhere in your application.

Because preferences are objects, you must "objectify" them. Numbers must be packed into

```
[NSNumber numberWith<Type>: value];
```

Generic objects must be archived into an `NSData` object. For example, to include a color use

```
[NSData *thisColor =
[NSArchiver archivedDataWithRootObject: [NSColor whiteColor]];
[prefsDictionary setObject: thisColor forKey: <a key>];
```

Setting preference keys

It can be useful to define global, application-wide, preference key constants. Literal strings won't trigger a compiler error if you type them incorrectly. Explicitly defined constants will. This is an optional extra feature, but it can help make an application more robust.

```
extern NSString * const AKey = @"A key name";
```

to define the key constant.

You can then use `AKey` as a substitute for `@"A key name"` throughout your application.

Reading preferences values

To read a preferences value in your application, access the preferences dictionary with a standard `valueForKey:` method. It's useful to initialize a pointer to the preferences dictionary once at the start of the application and then refer to it thereafter.

```
NSUserDefaults *prefs;
…
prefs = [NSUserDefaults standardUserDefaults];
 <type> aPref = [prefs valueForKey: <a key>];
```

When you read an archived or "objectified" key to check its value in your application, you must reverse the archiving or objectification process.

```
float aFloat =
[[prefs valueForKey: <a key which is a float>] floatValue];
NSColor *thePrefColor = [NSUnarchiver unarchiveObjectWithData:
   [prefs valueForKey: <a key>];
```

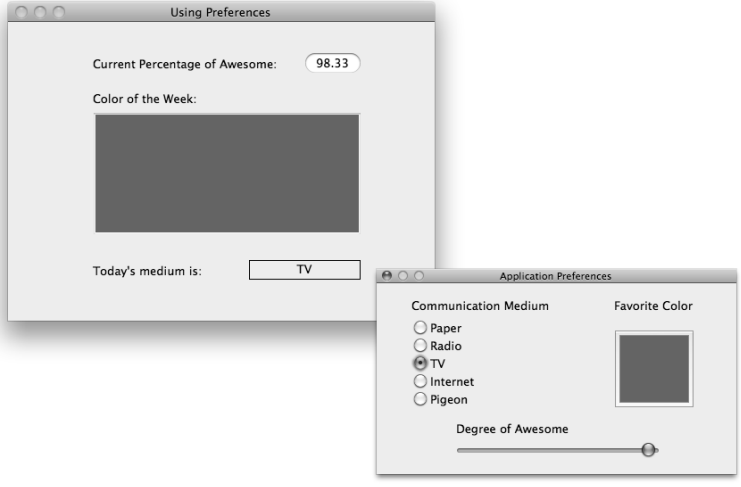
You can eliminate some of the conversions by binding variables to objects that display or control a preferences setting. For example, you can bind a slider directly to an NSNumber value in the preferences without converting it.

Creating an application with preferences

Figure 13.23 shows a simple application with a preferences panel. KVO and bindings are used to monitor changes to the panel and refresh the main application window.

Figure 13.23

When binding to visible objects in the UI, the controller can display various interpretations of the data, including selected elements chosen by the user.



The code for the Application Delegate is:

```
#import "PreferencesAppDelegate.h"

@implementation PreferencesAppDelegate
@synthesize window, preferencesController, theColorField;
NSUserDefaults *prefs;
- (void)applicationDidFinishLaunching:(
   NSNotification *)aNotification {
   prefs = [NSUserDefaults standardUserDefaults];
//Using KVO on the prefs values, we can trigger updates when the
//prefs change
   [prefs addObserver:self forKeyPath:CDRFavColorKey options:
   NSKeyValueObservingOptionNew context:NULL];
   [prefs addObserver:self forKeyPath:CDRCommMediumKey options:
   NSKeyValueObservingOptionNew context:NULL];
   [prefs addObserver:self forKeyPath:CDRDegOfAwesomeKey options:
   NSKeyValueObservingOptionNew context:NULL];
   //Reload color from prefs on startup
   [self updateColor];
   //Automatically show the prefs window on load
   [self showPreferences:nil]; }
- (void) showPreferences: (id) sender {
//Create a new preferences controller object
    NSLog(@"Showing preferences");
   if (!preferencesController)
   preferencesController = [[PreferencesController alloc] init];
   [preferencesController showWindow: self];
}
-(void) updateColor {
   //Get the curent color from the prefs and apply it
   NSColor *favoriteColor = [NSUnarchiver
   unarchiveObjectWithData: [prefs valueForKey:CDRFavColorKey]];
   theColorField.backgroundColor = favoriteColor;
}

- (void) observeValueForKeyPath:(NSString *)keyPath ofObject:(id)
   object
change:(NSDictionary *)change
context:(void *)context {
   if (object == prefs) {
   NSLog(@"%Keypath:%@ Value: %@", keyPath, [object
   valueForKey:keyPath]);
   if ([keyPath isEqual: CDRFavColorKey])
      [self updateColor];
   }
}
@end
```

Key features of the application include:

- **The Preferences item in the application menu triggers a** showPreferences: **method in the App Delegate.**
- **The method allocates an instance of a custom Preferences Controller object.**
- **When the Preferences Controller is created, it uses a custom** init **method to load and display the contents of a separate Preferences Panel nib file.** The nib, shown in Figure 13.23, includes some UI elements and an instance of `NSUserDefaultsController`. The UI elements are bound to the controller. The code initializes the preferences. UI updates, saving, and loading are handled automatically by the controller. No other code is needed.
- **The main application window includes a couple of UI elements that are bound to another copy of** NSUserDefaultsController**.** (Because this is a shared object, every copy accesses the same data.)
- **Color data can't be bound, so color updates are monitored with KVO.** Observers are initialized for every preferences key. The common observer method checks the key and runs update code to change the background color of the main text view when the color preference is updated.
- **In a more complex application, code here could add further auto-refresh features.**
- **One of the bound fields uses a *value transformer*, described below.**

NOTE

You can add an empty nib file to any project by right-clicking the Resources group in Xcode, choosing Add ➪ New File, selecting the User Interface tab and Empty XIB, and then saving and naming the file in the usual way.

Figure 13.24 shows the preferences panel nib. Instead of an `NSWindow`, the panel uses an instance of `NSPanel` to create a miniwindow suitable for preferences and other lightweight display tasks. Otherwise, it's a standard nib with a content view, some controls, and the Shared Defaults controller.

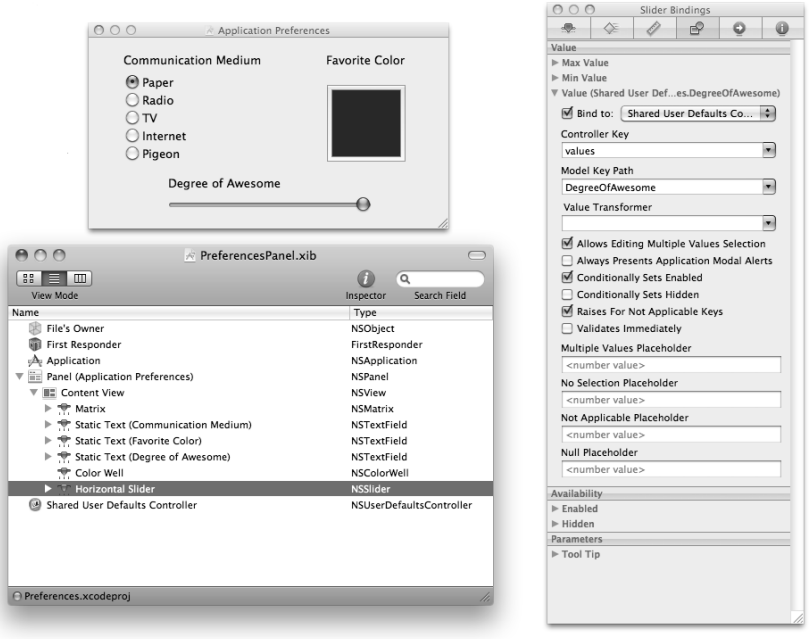
To bind to values in `NSUserDefaultsController`, use the values Controller Key. In the same way that `NSArrayController` offers arrangedObjects, selection, selectionIndex, and other predefined keys, `NSUserDefaultsController` offers the simpler values controller. As the name suggests, this is a simple list of its values. To access a value, type in its key name in the Model Key Path.

TIP

`NSUserDefaultsController` is added to a nib automatically when you bind to it as a data source. You don't need to add it by hand.

Figure 13.24

Binding preferences objects. Selecting the Shared User Defaults Controller as the source makes it possible to access values in the application's preferences.



Creating and Using Value Transformers

The preferences panel for the application includes a radio button, whose setting is saved as a numerical index. It's possible to convert the index to a string in the application using a switch statement, but it would be useful to create a binding that displays the correct string automatically.

Value transformers are a customizable extension to the transformation code built into bindings. You can use them to convert a value from any object into any other object, or to process values in more complex ways, such as performing math, changing data formats, or even sending values over the Internet to a remote server.

A value transformer is a subclass of the `NSValueTransformer` class. To create and use a transformer, follow these steps:

1. **Create a new object in Xcode.** Replace the header and the start of the implementation with the value transformer boilerplate code provided below.
2. **Add more boilerplate code to the App Delegate's initialize method to register the transformer.**
3. **Type the transformer's name into the Value Transformer box under the Model Key Path.**

When you save the nib file and build and run the application, the transformer processes the value pulled from the data source and replaces it with a transformed value.

In this example, you'll create a transformer called `IndexToNameTransformer` that takes a radio button index and returns a corresponding string. The boilerplate to register the transformer is shown below. Add equivalent code to the App Delegate, replacing the name field with the name of your own transformer object. You can allocate multiple transformers. Each must have a unique name string.

```
+ (void) initialize {
 NSValueTransformer *transformer =
[[IndexToNameTransformer alloc] init];
[NSValueTransformer setValueTransformer:transformer
   forName:@"IndexToNameTransformer"];
}
```

The transformer object header declares your custom subclass of `NSValueTransformer`, which defines the transformer's name.

```
#import <Cocoa/Cocoa.h>
#import <Foundation/Foundation.h>
@interface IndexToNameTransformer : NSValueTransformer
{}
@end
```

Properties and other variables are declared in the implementation, listed here. A switch statement converts the incoming `int` into a text string, which it returns.

```
#import "IndexToNameTransformer.h"

@implementation IndexToNameTransformer

//These two methods are boilerplate
+ (Class)transformedValueClass
{
    return [NSString class];
}
+ (BOOL)allowsReverseTransformation
{
    return NO;
}
//The transformer method signature is fixed, the code is
   customizable
- (id)transformedValue:(id)aValue
{
   int thisIndex = [aValue intValue];
   switch (thisIndex) {
   case 1:
   return @"Radio";
   break;
   case 2:
   return @"TV";
   break;
   case 3:
   return @"Internet";
   break;
   case 4:
   return @"Pigeon (RFC 1149)";
   break;

   default:
   return @"Paper";
   break;
   }
}
@end
```

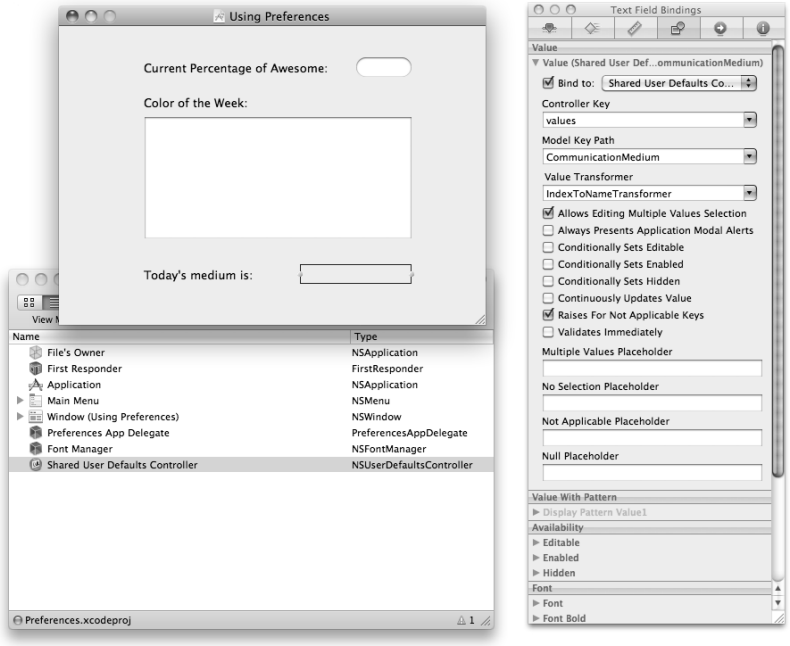
Figure 13.25 illustrates how to use the value transformer. Type the name manually into the Value Transformer combo box under the Model Key Path. The result was shown in Figure 13.23: the index was automatically converted into an associated string, which appeared in a text box. Without the transformer, binding to the radio button preference would have displayed a number.

TIP

The bindings manager includes a selection of predefined transformers. The two most useful are `NSUnarchiveFromDataTransformerName` and `NSKeyedUnarchiveFromDataTransformerName`, which can convert archived preferences objects back into objects. For an example, see the Favorite Color binding in the preferences application.

Figure 13.25

Setting a value transformer. Once the transformer is registered in code, you type the name into the Value Transformer box to apply it to the data source selected in the Model Key Path.



Summary

In this chapter, you learned how to use bindings. You began with a look at why bindings are useful, and then explored some of their less well-known features. Next you looked at a practical example of creating and using simple bindings that connected a UI control to a value set inside an application, in both directions.

To show you how to create more sophisticated applications, you were introduced to controller objects, and you learned about their features and functions and how they can be used to eliminate glue code.

You discovered how to use bindings and KVO to create a complete preferences solution for applications, with some sophisticated features. Finally, you learned about value transformers, and you saw a simple sketch of a transformer class that you can customize and use in your own projects.

14 Using Core Data

Core Data is a collection of classes designed to simplify the editing and archiving of linked data that can be used as the model in any application.

You use Core Data when your application needs to manage data collections in which objects take some of their properties from other objects. This can include user-oriented data collections such as music playlists, EXIF (Exchangeable Image File Format) data managers from photo collections, and contact databases. It can also include application-specific data that is never accessed directly by users. For example, you can use Core Data to store the complete project state for a music sequencer or video editor, with all of its subelements.

You can implement simple linking with Cocoa's data collection objects, but Core Data offers the following features "free of charge":

- Undo
- Data persistence without explicit archiving and de-archiving
- Built-in support for bindings
- An automated UI generator in Interface Builder (IB)
- An abstracted interface that can report the edit/undo state of objects
- Visual editing of object relationships

14 In This Chapter

Creating a Core Data application visually

Understanding Core Data's objects and programming model

Displaying search results

CAUTION

Core Data isn't a full relational database, and it doesn't support some of the relationships and features that are standard in SQL and other database environments. For certain features, work-arounds may be required.

For newcomers to Cocoa, Core Data can seem abstract and difficult to understand. The programming model uses many unique classes and introduces some new concepts. But if you can understand Key-Value Coding (KVC), with some effort, you can use Core Data.

There are two approaches to Core Data development:

- **Create an application visually, with Xcode's Core Data editor, and then add extra code.** This is a good approach for any record-based application that supports user editing, and it is the one demonstrated in this chapter.

- **Define a data model and interfaces with code.** Optionally, you can create a custom UI. This is the more advanced approach and is often used for application-specific data. Once you understand the key features of Core Data, you can use it in this general way.

CAUTION

You will need to understand bindings from the previous chapter to follow the project in this chapter. You should know how to bind a table view to an array controller and how to bind the controller to a data source array.

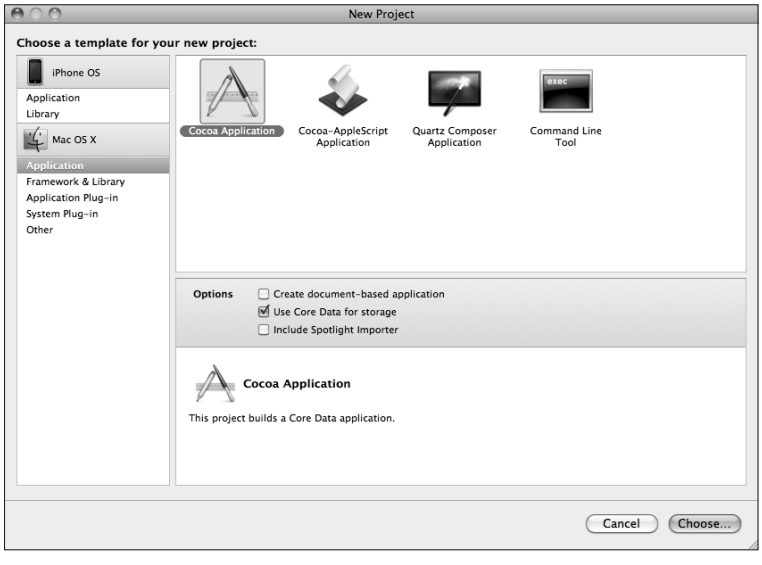
Creating a Core Data Application Visually

In this chapter, you'll use Core Data to create a music database that links artists to albums, and vice versa.

Xcode includes a Core Data editor and associated Core Data application templates. First, you create a new application, as shown in Figure 14.1. Make sure the "Use Core Data for storage" check box is selected. This is a small change, but it tells Xcode to initialize the project with the standard Core Data template, which includes extra boilerplate that sets up Core Data features and supporting objects. Save the project as CoreData.

Figure 14.1

Creating a Core Data project. The "Use Core Data for storage" check box selects a project template with a generous helping of Core Data setup code.



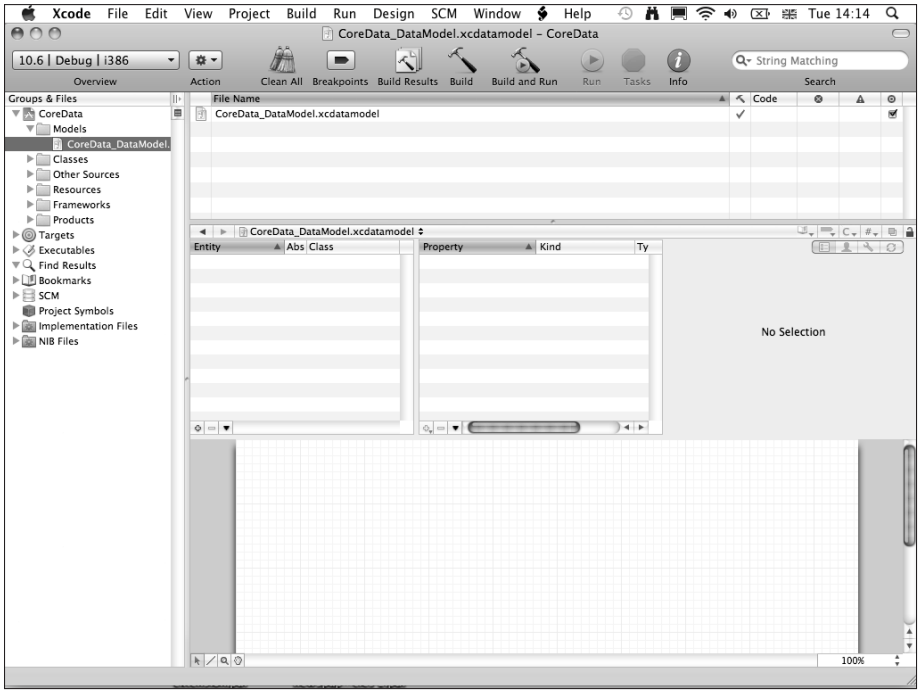
The template creates a project with an extra group, called Models, at the top of the Groups & Files pane. Open Models and you'll see an item called CoreData_DataModel.xcddatamodel. Xcode loads a visual editor for the model, as shown in Figure 14.2.

There are three panes in the middle of this window.

- **The Entity pane** lists the items in the model. An entity is a data object. Use this pane to view a list of entities, add new entities, and delete existing ones.
- **The Property pane** lists the properties in each entity. Properties are key-value pairs. Each property has a name and an associated field that stores values. Use this pane to list, add, and delete properties.
- **The Inspector pane,** on the right, lists the attributes and features of entities and properties as they're selected. Use this pane to set the type of a property, define how it connects to other entities and properties, and so on.

Figure 14.2

Loading the visual editor for the model. The graph-paper area at the bottom of the editor is used to display and link the objects in the data model.

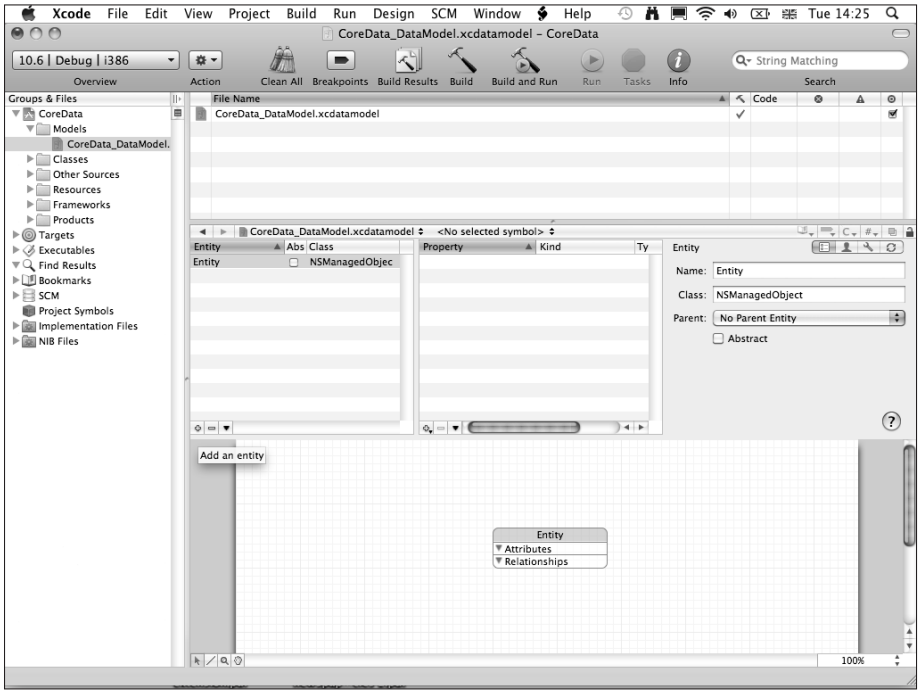


Adding an entity

To add an entity, click the + (plus) icon at the bottom left of the Property pane, as shown in Figure 14.3. A new entity appears in the graph at the bottom of the window.

Figure 14.3

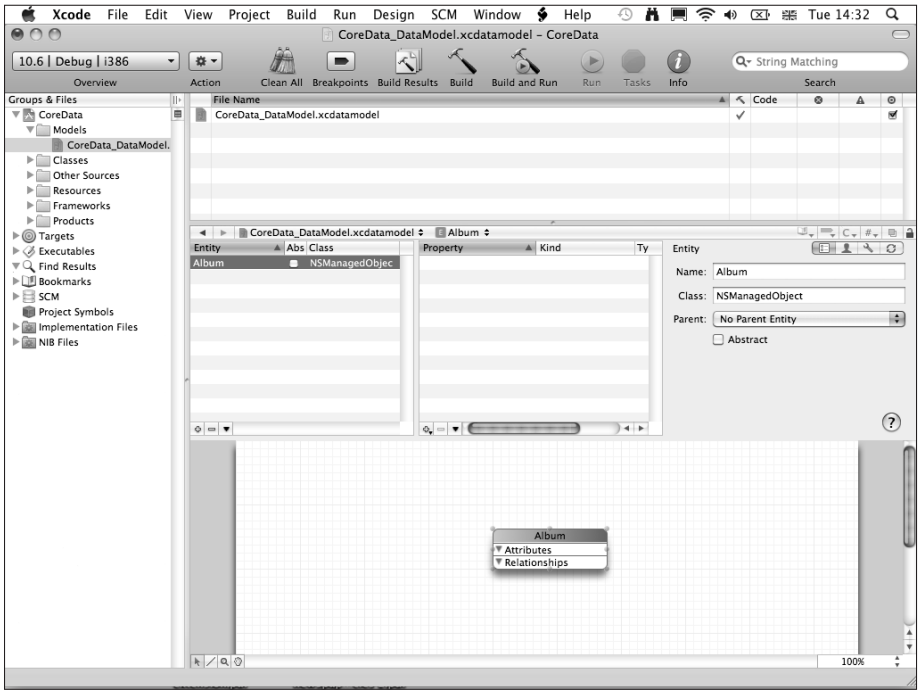
Adding a new entity. You can also right-click in the graph area and select Add Entity from the pop-up menu.



In the Inspector pane, type **Album** into the Name field. Press Return to apply the name, as shown in Figure 14.4.

Figure 14.4

Naming an entity. Note that the Inspector pane also lists the entity's class. The `NSManagedObject` class is a special wrapper class used by Core Data.

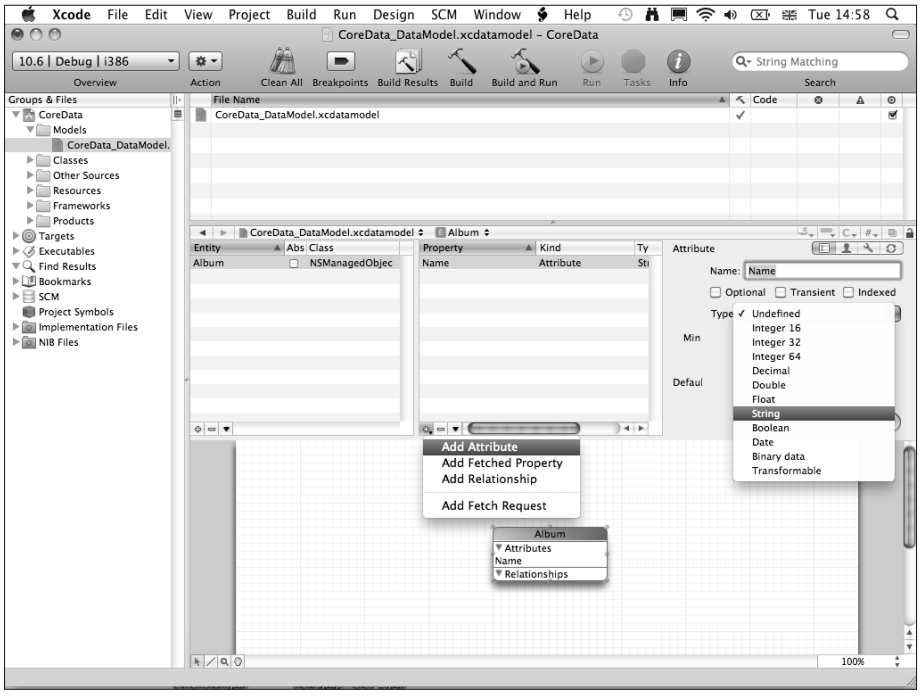


Adding properties

Next, you'll add some properties to the Album entity. Click the + icon at the bottom left of the Property pane, and select Add Attribute, as shown in Figure 14.5. Type **Name** into the name field in the Inspector window. Deselect Optional. Select String from the Type pop-up menu. This adds a new property called Name, which is a string, and must be present. Repeat the steps to add another property. Name it **Year**. Set its Type to Date and deselect Optional.

Figure 14.5

Adding a property. Use the Inspector pane to set the name and type. Deselecting Optional means that the property must be set. (Note: This image is a composite showing both menus.)



Create another entity called Artist. Add a nonoptional property called Name. The model graph should look like Figure 14.6.

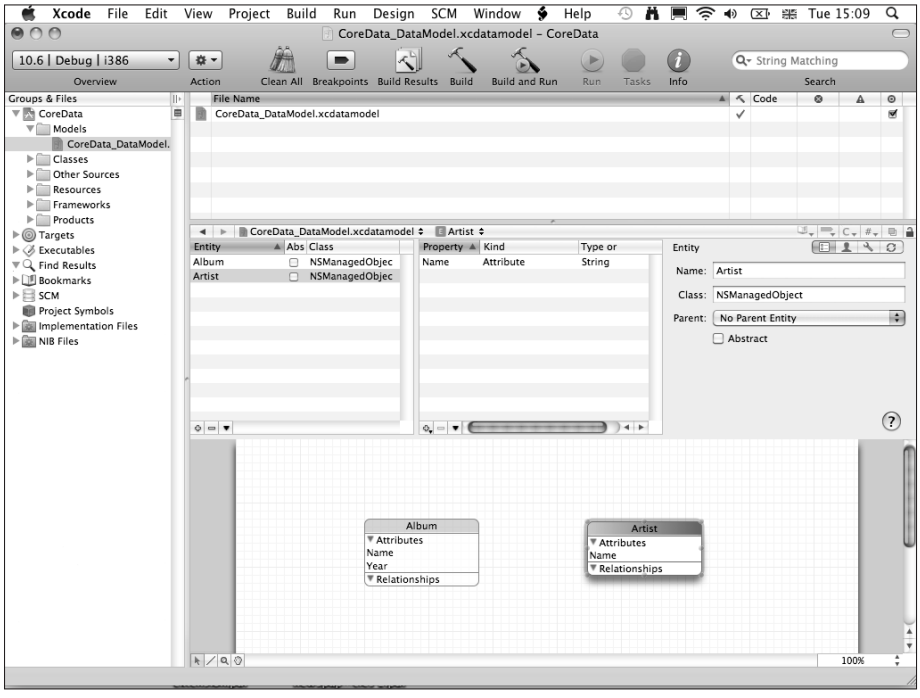
Creating relationships

A relationship is a link between one entity and another. Relationships are used to pull data out of associated entities. They define properties indirectly.

In this example, Albums are created by Artists, so there needs to be a relationship between the Album and Artist entities. There should also be a relationship in the other direction, because Artists can record more than one Album.

Figure 14.6

Your model is starting to take shape, with two entities.



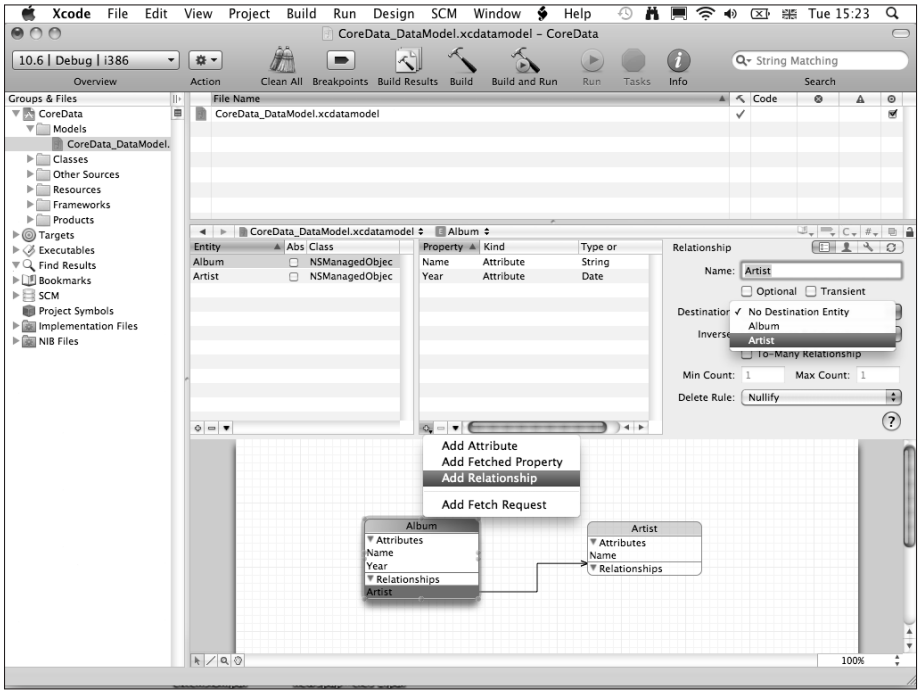
A relationship from one object to multiple objects is called a *to-many relationship* — one entity is connected to many others. Unlike a full relational database, Core Data's support for to-many relationships is limited.

To illustrate this, you'll create the Album ➪ Artist relationship as a to-one, and the Artist ➪ Album relationship as a to-many.

Select the Album entity, and click the + icon at the bottom left of the Property pane. Select Add Relationship from the pop-up menu, as shown in Figure 14.7. Name the relationship **Artist.** Deselect Optional. In the Destination pop-up, select Artist. The editor automatically adds an arrow linking the two entities.

Figure 14.7

Creating and defining a relationship. The arrow appears automatically.



NOTE

Although the editor is visual, the layout of the model on the graph isn't important. Names and properties are important, and so is the fact that a relationship exists. But you can move entities to different locations on the graph without affecting either.

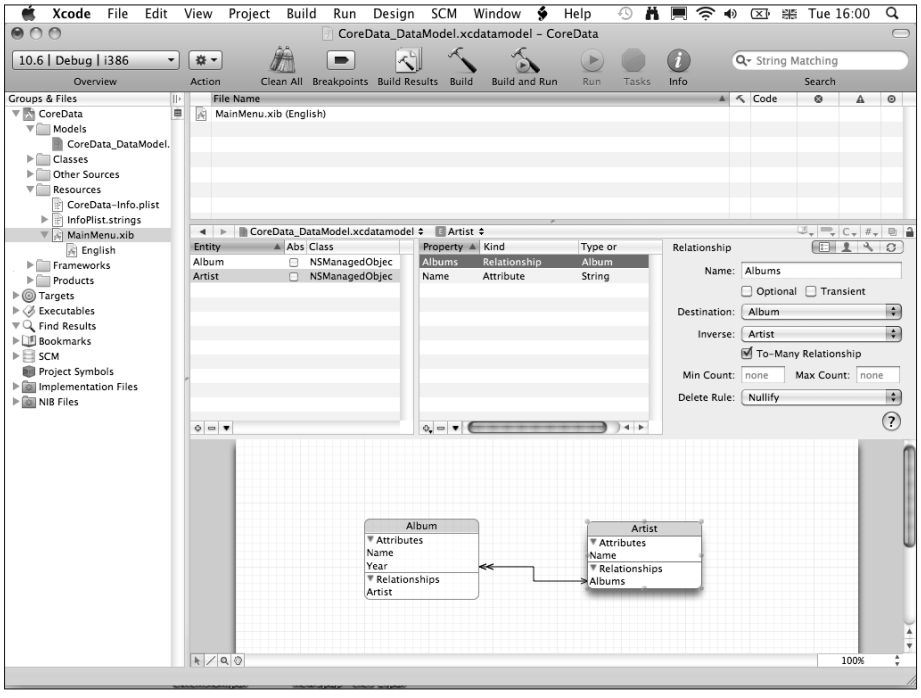
Repeat the process for the Artist entity, creating a relationship to Album, as shown in Figure 14.8. Selecting Artist in the Inverse pop-up menu creates a two-way relationship and replaces two arrows with a single arrow with arrowheads at both ends.

TIP

The double arrowhead indicates a to-many relationship. The single arrowhead indicates a to-one relationship.

Figure 14.8

Creating a reciprocal relationship. The two entities are now connected in both directions.



Generating a user interface

Interface Builder can take a Core Data entity and build an interface for it. This feature is very easy to use and generates a default nib file that you can customize to taste. Bindings are included to link objects in the UI to the objects managed by Core Data.

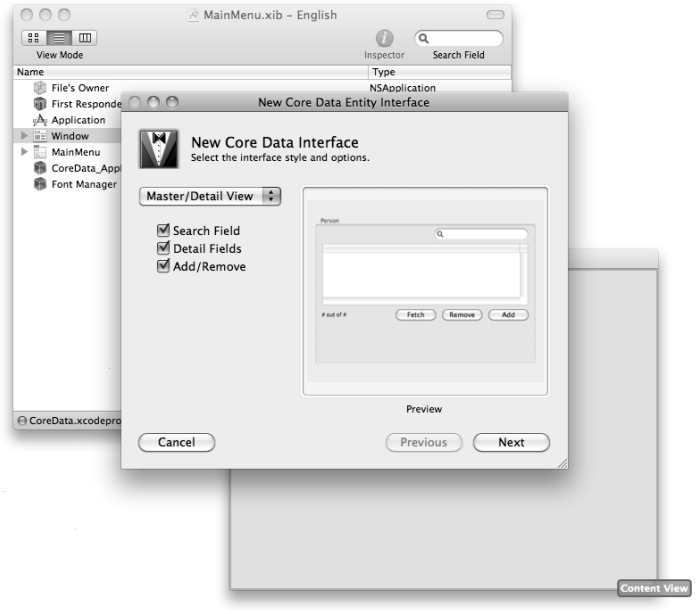
Open the Resources Group, and double-click `MainMenu.xib`. Open the Document window if it isn't already visible. Double-click the Window object to open its edit window. Press Option+drag the Album entity and release it on the empty edit window. You'll see the dialog shown in Figure 14.9. Select the Master/Detail view option from the pop-up menu, and select all the boxes. When the next dialog appears, leave the property boxes and select Finish.

TIP

This is very much easier with two monitors. If you have a single monitor, you'll have to resize Xcode and Interface Builder to fit both onto the screen at the same time.

Figure 14.9

Creating a Core Data interface in IB. The check boxes define the elements that appear in the interface.

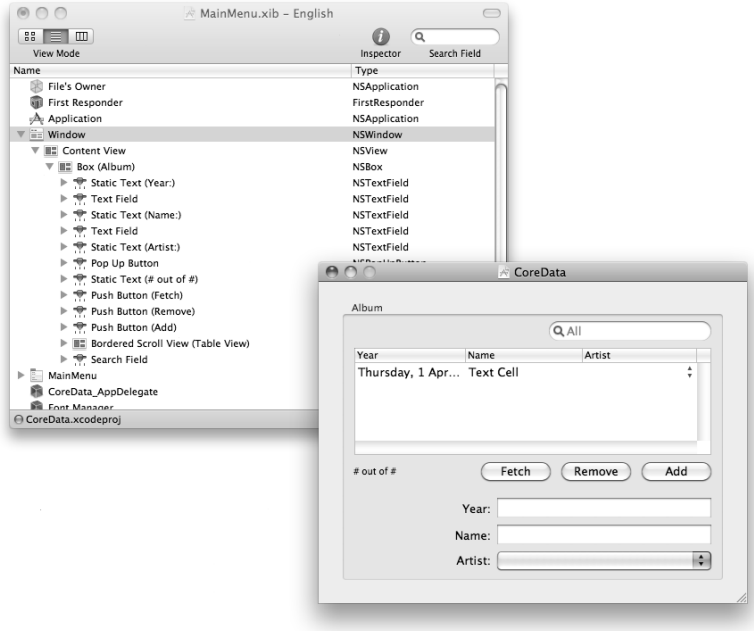


IB adds the UI shown in Figure 14.10 to the window. Any properties that are included in the source entity will appear here as text fields. To-one relationships linked to other entities generate a pop-up. As you can see at the left, this is a standard nib file. You can move elements around, add autosizing, and so on.

Make the window much wider to add another UI panel, and then press Option+drag the Artist entity into the window. Make the same selections as before to create another Master-Detail view showing the Artist properties. The finished UI is shown in Figure 14.11. You won't see a pop-up menu for albums, because the UI doesn't support to-many relationships. This simple UI isn't intelligent enough to allow to-many editing.

Figure 14.10

The auto-generated user interface for the project, with its associated nib.



TIP

The new panel appears in the middle of the window, overlapping the original panel. If you mis-click while trying to drag it, it can be difficult to select it again. The easy way to select it before moving it is to double-click the Box (Artist) element in the Doc window.

Building the application

Once the UI is complete, save the nib file and Build and Run the application. No code is needed! The completed application is shown in Figure 14.12. Click the Add button to add some artists, then add some albums in the same way. Use the Artist pop-up menu to select the artist name for each album. You can also click the entries under the Artist column directly.

Figure 14.11

Adding another panel for the Artist entity. Repositioning this panel can be tricky.

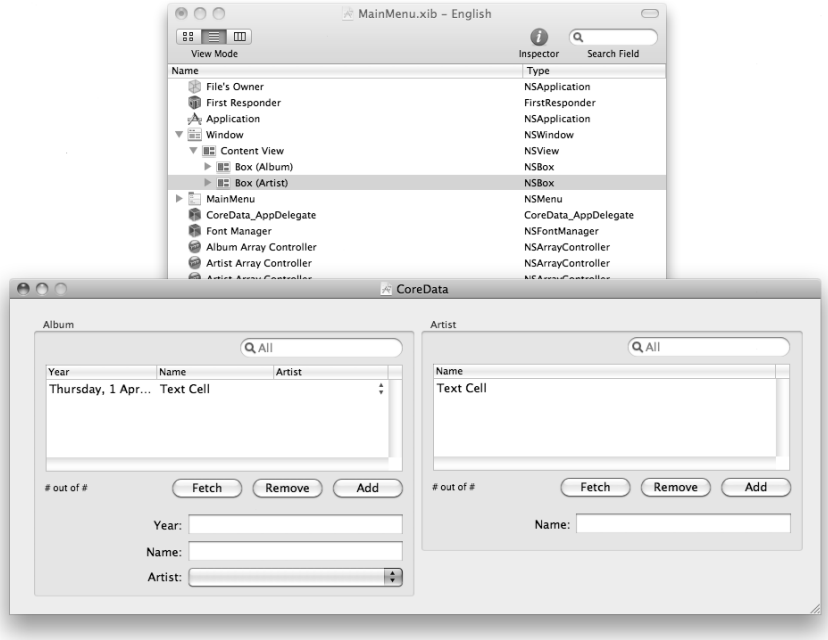
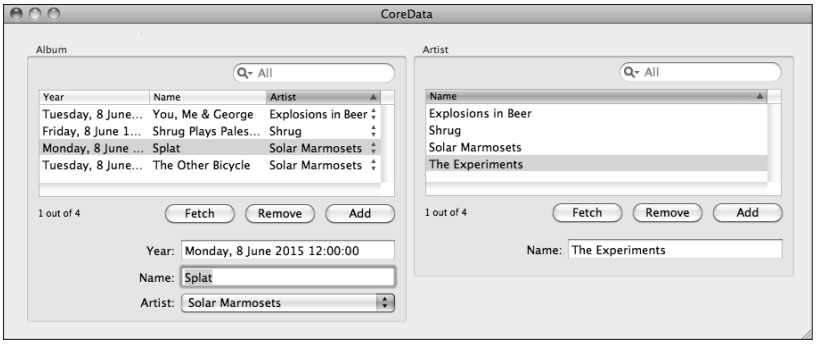


Figure 14.12

The finished application with some sample data



The design is rough around the edges, but it supports the following features:

- **Input checking.**
- **Editing.** You can change any entry.
- **Sorting.** Click the column headers.
- **Searching.** Type into the search bar.
- **Save and reload.** Choose File ➪ Save to save the data. Data is reloaded automatically.
- **OS X standard undo.**

This is impressive without code; but it does have some limitations:

- **The date field** shows the day, date, and time, when it should only show the year.
- **Links are unidirectional.** It would be useful to filter the albums by artist.

CAUTION

Core Data can't reload previously saved data if you change the model; if you try this, the application will crash. A feature called *migration* can repair entities programmatically, but migration is an advanced topic and is beyond the scope of this book.

Exploring and Extending a Core Data Application

You can improve on this application in two ways: by modifying the nib and by adding code to implement custom features. Take a closer look at the application nib, shown in Figure 14.13.

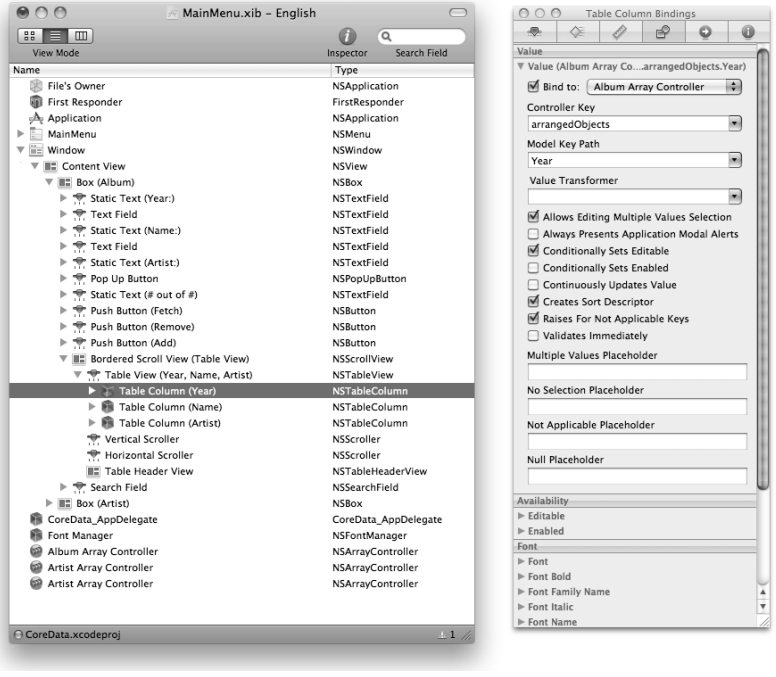
The format is similar to the table view example from the previous chapter. The Album box includes a table view with three column objects. Each is bound to a separate key in the Album Array Controller object. This generates the three column views.

Each array controller is bound to the `managedObjectContext` in the App Delegate, as shown in Figure 14.14. This is Core Data's main data store. The managed object context binding option includes features that search the store automatically, returning arrays of values and keys that can be displayed by an array controller.

Expanding the Year column reveals that the Year field already includes a date formatter object. A date formatter is similar to the `NSNumberFormatter` object introduced in the previous chapter. You can use it to control how dates are displayed.

Figure 14.13

Exploring the auto-UI, and discovering some of its bindings



Open the date formatter's attributes, as shown in Figure 14.15, delete the existing format, and replace it with the year formatter from the collection under the Format window, dragging and dropping it.

This solves the format problem. The Year column should now display a single year correctly. Modify the date formatter used to display the year in the year-entry text field in the same way. It's above the scroll view, inside the text field under the Static Text (Year:) label.

To solve the second problem, you'll need to access the model data with code. While Core Data supports generic filtering for data display, there isn't a simple way to implement it here. It would be possible to add a custom data transformer that checked an Artist string and filtered results accordingly. But you'll create the same result by searching the data directly and packing a list of results into an array that can be displayed in a separate table. To do this, you'll need to look at Core Data's classes and programming model.

Figure 14.14

Exploring how the array controllers are bound to a single common data store, called a *managed object context*

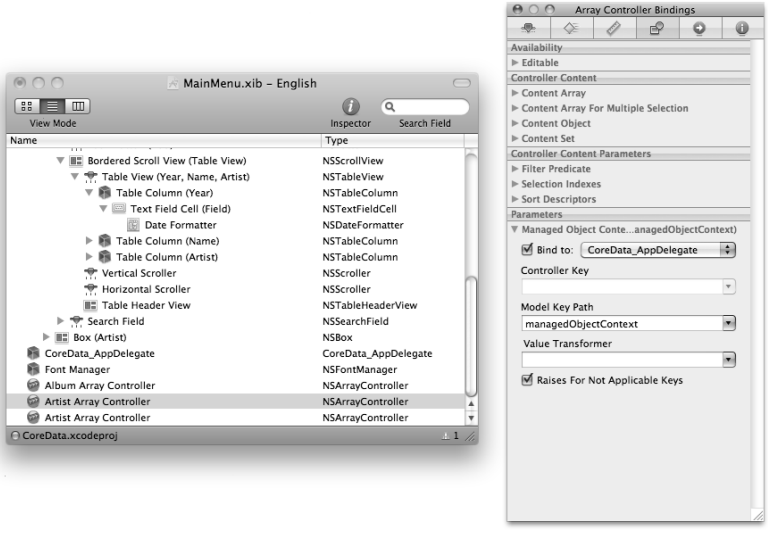
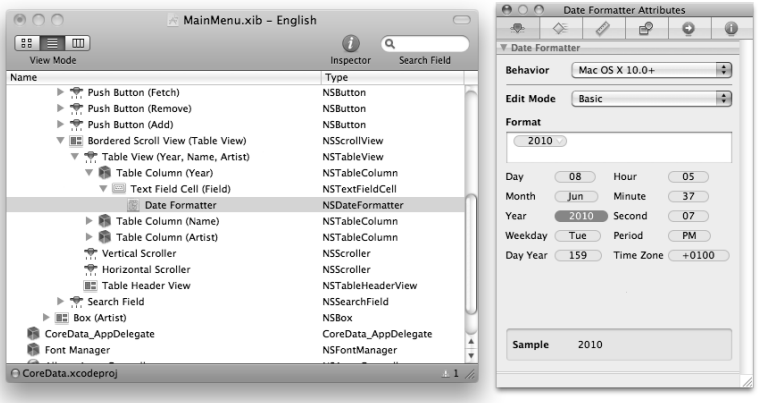


Figure 14.15

Modifying the date format to eliminate time, day, and date strings



Understanding Core Data's objects and programming model

Core Data uses three objects to define its working environment, which are listed in Table 14.1. If you use the Core Data template and Xcode editor for your project, the persistent store coordinator and the managed object model "just work." You do, however, need to access the managed object context to read and write data.

Table 14.1 Core Data Environment Objects

Object	*Application*
`NSPersistentStoreCoordinator`	Archives and unarchives data, and keeps saved and live data synchronized. You can ignore this object in the Core Data template. The boilerplate code initializes it for you and it works automatically.
`NSManagedObjectModel`	The collection of entities, properties, relationships, and other features used in the data model. The managed object model defines how the data is organized, but doesn't store the data — it's a template for the model. Don't change the model dynamically. It's best to leave model management to the model editor in Xcode.
`NSManagedObjectContext`	The complete database, with data. The template creates a context automatically. An application can use more than one context — but usually one is enough.

The Core Data classes used to read and write data are shown in Table 14.2.

Table 14.2 Core Data Objects used for Editing and Searching

Object	*Application*
`NSManagedObject`	A wrapper for entity objects that plugs them into the archiving and editing features built into Core Data. Sometimes you need to access a managed object directly. At other times you access its associated entity description object. Supports Key-Value Coding (KVC).
`NSEntityDescription`	Defines the name, property list, and relationships of a named entity. A managed object holds data; its entity description describes how it's organized.
`NSFetchRequest`	A wrapper for a search. The search request is sent to a context, which returns an array of objects that match the search criteria.
`NSPredicate`	A predicate object that defines the details of a search request. It holds the low-level details of search strings or patterns, matching criteria, case sensitivity, and so on.
`NSSortDescriptor`	An object that specifies how search results should be sorted. Optionally, it can be passed to a predicate object before a search request.

Creating fetch requests

To read data, create a fetch request and pass it to the managed context. Fetch requests can be complex, and the code interface isn't intuitive, but in outline, there isn't any more to searching than creating a request, setting its parameters, passing it to the context, and reading the results.

Sample code is shown below:

```
NSEntityDescription *albumEntity
= [NSEntityDescription entityForName:@"Album"
inManagedObjectContext:managedObjectContext];
NSFetchRequest *theRequest = [[NSFetchRequest alloc] init];
[theRequest setEntity:albumEntity];
returnArray = [managedObjectContext
   executeFetchRequest:theRequest error:&error];
```

In the context of your model, this code runs the simplest possible search. `returnArray` returns a list of all Album entities in the context; that is, a list of all albums. You can then process the array further to extract values from it.

Using predicates

To narrow the search, you can add a predicate to the search request:

```
NSPredicate *thePredicate =
[NSPredicate predicateWithFormat:@"Name == 'An Album Name'"];

[theRequest setPredicate:thePredicate];
//Run the search...
```

The predicate syntax is complex, but recognizably similar to regex matching, which it's based on. `predicateWithFormat:` allows substitutions with the usual `withFormat:` options. Use this to insert search strings and other values dynamically.

```
NSPredicate *thePredicate =
[NSPredicate predicateWithFormat:@"Name == %@", aString];
```

Table 14.3 shows some other examples. For more information about predicate programming, see the Predicates Programming Guide in the Cocoa documentation.

Table 14.3 Sample Predicate Search Strings

Object	*Application*
`like 'string'`	Searches for similar strings. A literal string must include single quotation marks.
`contains[cd] 'ey'`	Finds all records with `"ey,"` ignoring case and diacritical characters; that is, non-English characters with accent marks.
`beginsWith 'A'`	Finds all records starting with "A."
`ANY salary < 20000`	Returns the first found salary < 20,000. No quotation marks are needed for numbers.
`ALL salary > 1000000`	Returns all salaries over a million currency units. (The units depend on the data and the context.)

Using sort descriptors

Sort descriptors can sort the results into ascending or descending order with respect to named key. Descriptors are passed in an array, so it's possible — but not always useful — to specify multiple descriptors.

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
   initWithKey: @"Artist" ascending: YES];
[theRequest setSortDescriptors:
[NSArray arrayWithObject: sortDescriptor]];
```

A fetch request *must* name an entity, and it *may* specify a predicate and a sort descriptor.

TIP

One of the biggest challenges for newcomers is the absence of a simple query mechanism. There are no simple "Find me an object with the following properties" methods. Instead, queries are handled with query objects, which have to be assembled and packed for each query. You can make Core Data easier to work with by writing query wrappers to simplify the code interface.

Creating to-many fetch requests

Often, you can use a simple predicate search to return an array of results. No more code is needed.

Unfortunately, Core Data doesn't support searches where the target is a to-many relationship — that is, where one item can be linked to many others.

For example, if you want to filter albums by artist, it would be natural to assume that you can do something like:

```
NSEntityDescription *albumEntity
= [NSEntityDescription entityForName:@"Album"
inManagedObjectContext:managedObjectContext];
NSFetchRequest *theRequest = [[NSFetchRequest alloc] init];
[theRequest setEntity:albumEntity];
NSPredicate *thePredicate =
[NSPredicate predicateWithFormat:@"Artist == %@", aString];

[theRequest setPredicate:thePredicate];
```

But this doesn't work. The `Artist` key is a relationship, and the regex search code can't follow to-many relationship links.

A workaround is to create a list of album keys and values in a separate array, and search the array for the `Artist` key. Instead of specifying a predicate, you request a list of all albums, and then search the returned array.

Counterintuitively, this solves the problem, with only minor extra complications. The code for a full solution follows. The search is packed into a `refreshList:` method that can be called when the application should refresh the list of filtered albums.

```
- (void) refreshList: (id) sender {
    NSEntityDescription *albumEntity =
    [NSEntityDescription entityForName:@"Album"
    inManagedObjectContext:managedObjectContext];
    NSFetchRequest *theRequest = [[NSFetchRequest alloc] init];
    [theRequest setEntity:albumEntity];
    searchReturnArray = [NSMutableArray arrayWithCapacity:50];
    tempArray =
    [managedObjectContext executeFetchRequest:theRequest
   error:nil];
    [tempArray enumerateObjectsUsingBlock:^(id obj,
    NSUInteger idx, BOOL *stop) {
        NSString *albumName = [obj valueForKey:@"Name"];
        NSString *artistName =
       [[obj valueForKey:@"Artist"] valueForKey:@"Name"];
        if ([artistName isEqual: selectedName.stringValue])
            [self.searchReturnArray insertObject:
            [NSDictionary dictionaryWithObject: albumName
            forKey:@"FilterName"] atIndex: 0];
    }] ;

    self.searchReturnArray = searchReturnArray;
    NSLog(@" %@", searchReturnArray);
}
```

Key features of the code include:

- **A fetch request returns the full list of albums.**
- **The** enumerateObjectsUsingBlock**: method processes each object in turn.**
- **The processing block extracts an album name and an artist name for each album.**
- **The artist name is retrieved with** valueForKey: **to return a reference to an artist object, and then extracts the name from the object with the** Name **key using nested code.**
- **The name is compared with a string read from a text field in the UI.** (Assume an outlet and link to the UI have been created.)
- **Matching results are packed into an array of dictionary objects, each containing a name linked to a key —** `FilterName`**.** This makes it possible for an array controller to create a list of items with that key and bind it to a column in a table view.
- **The search array is assigned with** `self` **to make it visible to a binding.**

Displaying search results

There are various ways to display the results of the search. Figure 14.16 shows one solution. The default UI has been rearranged, and an extra table view and associated array controller have been added. The controller is bound to the `searchReturnArray` as its data source, which has been added to the App Delegate header.

The single column in the new table view is bound to the `FilterName` key in the Array Controller, via arrangedObjects, as shown in Figure 14.17.

A few extra niceties are needed to finish the code. It would be possible to add a separate search button to create a filtered search, but it's more intuitive for the user to see an updated list whenever he or she selects a new artist. Key-Value Observing (KVO) could be used for this, but the table view's selector has been linked to the `refreshLink:` method, as shown in the list of links to the App Delegate in Figure 14.18. An outlet to the text field that displays the artist name has also been created.

When the user selects a new artist name in the list in the Artist panel, the following happens:

1. **The table view updates the name box under the list, via a binding that was built into the original UI.**
2. **The table view's selector action triggers the** refreshList: **method.**
3. **The method reads a list of all albums from the Core Data store and copies them to a temporary array.**

4. **With some minor coding trickery, it reads the artist key from each album.** If the artist name matches the name box, it inserts the album name into the `searchReturnArray`.
5. **It assigns the array to itself to trigger a KVO update.**
6. **The array controller in the nib notices the updates, and sends them to a column in the Albums table view to display them.**
7. **The list appears.**

Figure 14.16

Creating an improved interface and adding a table view to display a summary list of albums for each artist, with a new array controller to bind the search results to the table

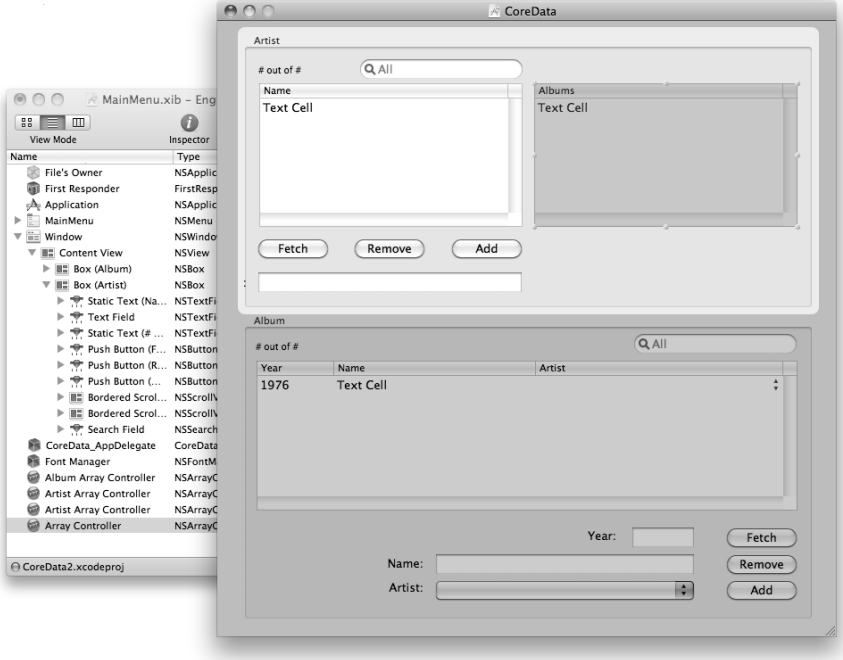


Figure 14.17

Binding the new table to the custom FilterName property, which is used to tag dictionary entries in the filtered search code

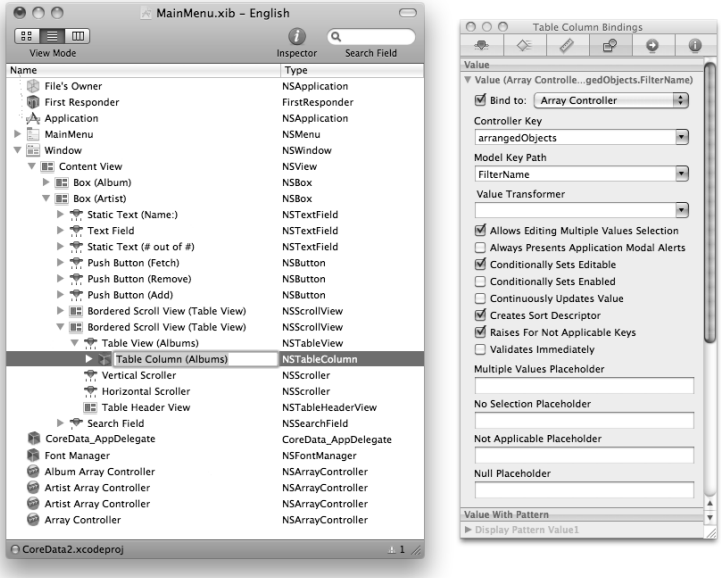
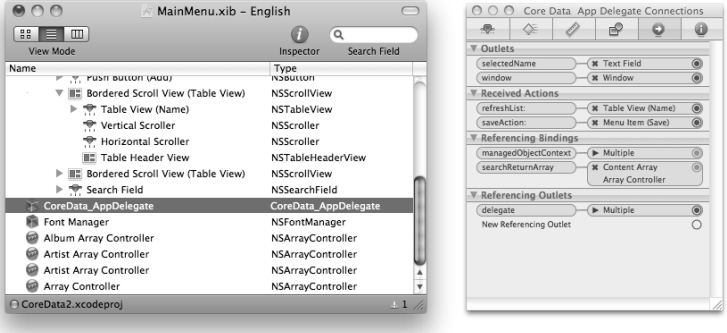


Figure 14.18

Links and bindings to the App Delegate for this application. The two key links are the link to the outlet to the `selectedName` text box, which displays the current selected artist name, and the `refreshList:` method, which is triggered by a new selection in the Artist table.



While this isn't the most elegant solution, it solves the problem. The finished application is shown in Figure 14.19.

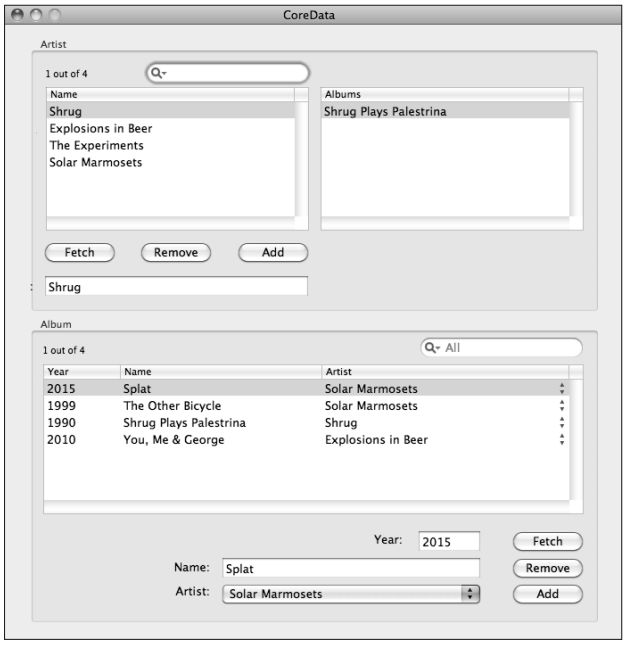
As an exercise, try to extend the application so that the Album box displays the year as well as the artist name.

TIP

One final nicety is to add an auto-refresh feature on load. The finished application implements `applicationDidFinishLaunching:`. A `performSelector:withObject:afterDelay:` method calls `refreshList:` one second after start-up. This updates the album list after giving the application a short time to settle.

Figure 14.19

Displaying albums automatically. Selecting a new artist lists that artist's albums in the adjacent text field.



Summary

In this chapter, you learned about Core Data. You created a Core Data project using a template built into Xcode, and you designed a data model using Xcode's model editor.

You discovered how to convert the model into a simple application, using the conversion features built into Interface Builder, and how to fine-tune elements of the interface to improve the look and feel of the application.

Next, you learned about Core Data's key objects, including the managed context, entities, and fetch requests. You created a simple fetch request and used it to retrieve a list of entities. You also explored the limitations of Core Data and looked at one possible way to work around them.

Finally, you created an extended version of the original application, with added search features and an improved interface.

15 Working with Text and Documents

Cocoa includes classes that implement powerful multi-language text features and drastically simplify the design of documents that support styled and formatted text. An associated Xcode template makes it easy to create an application that supports multiple editing windows. Each window is an instance of a customizable `MyDocument` class, which is a subclass of `NSDocument` — Cocoa's document object. You can define the look and feel of this class, by customizing its associated nib, and add custom code to implement saving and loading features for customized and existing file formats.

Using NSString

`NSString` is Cocoa's string-handling class. It supports a vast library of methods that can initialize, convert, compare, split, combine, and process string data.

`NSString` objects are read-only. You can create edited strings by copying the output of one of the string-processing methods to a new string. You can also use `NSMutableString`, which implements insert, delete, and replace methods.

Table 15.1 lists a small selection of useful `NSString` methods. The full list in the `NSString` Class Reference is *much* longer.

15 In This Chapter

Using `NSString`

Using `NSAttributedString`

Creating multi-document applications

Implementing undo and redo

Supporting other languages

Table 15.1 Some Useful NSString Methods

Method	*Description*
`stringWithFormat:`	Takes a formatting string and a content string; applies standard C formatting from the former to the latter.
`stringWithContentsOfFile:`	Loads the string with the contents of a file from a filepath or a URL.
`stringWithContentsOfURL:`	Loads the string with the contents of a URL, which can be a local file or an online source. With a standard Web URL, this method downloads the HTML from the specified Web page.
`writeToFile/URL:`	Writes the string to a filepath or URL, with encoding options and error trapping.
`length:`	The string length in characters.
`componentsSeparatedByString:`	Searches a string for a separator character or string, and splits it into substrings. Can split filepaths ("/") or extract words from text (" ").
`getLineStart:` `getParagraphStart:`	Finds the start and end of lines and paragraphs.
`isEqualToString:`	Compares two strings for identity. Faster than `isEqual:`.
`caseInsensitiveCompare:`	Compares two strings, ignoring case.
`capitalizedString:`	Capitalizes the first letter of each word.
`lowercaseString:` `uppercaseString:`	Creates lower- and uppercase copies of a string.
`<type>Value:`	Converts text into a number or Boolean. Supports `int`, `float`, `double`, `integer`, `longlong`, and `Bool` types.
`stringByStandardizingPath:`	Filters a path string to remove invalid elements.

Using NSRange

Many string operations use a *range* — a `struct` with two integers that define a `location` in the string and a `length`. Ranges are used to extract and edit strings. Some Cocoa classes, such as `NSTextView`, return a range automatically when the user highlights a block of text.

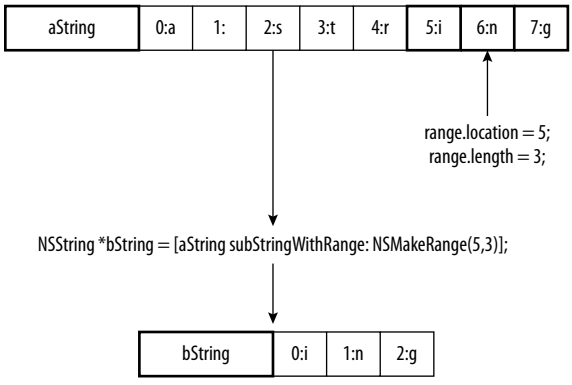
You can create a custom range with

```
NSRange myRange = NSMakeRange(location, length);
```

Figure 15.1 illustrates how you can use a range and the `subStringWithRange:` method to extract a substring from a string.

Figure 15.1

Extracting a substring from the characters in `aString` to a new `bString` with `NSMakeRange();`



CAUTION

A range is really just a pair of integers. It's a structure, not an object. It doesn't need a pointer, isn't associated with any one string, and isn't part of a string's properties.

Working with encodings

`NSString` supports a range of *encodings*, which make it possible to work with languages that do not use Latin characters. The preferred encoding is UTF-8 (Unicode Transformation Format-8), which is a byte-packed implementation of Unicode. Single-byte characters are backward-compatible with ASCII. Multi-byte characters define the rest of the world's written languages. In theory, characters may use up to 6 bytes, but in practice, no language uses than more 4 bytes per character.

Selecting a default encoding

It would be helpful if you could assume UTF-8 is the default encoding. Typically, your application should save text using the `NSUTF8StringEncoding` constant.

```
[aString writeToFile: aPath
encoding: NSUTF8StringEncoding: error: &error];
```

But some Web sources, such as RSS (Really Simple Syndication) feeds, use other encodings such as `NSISOLatin1StringEncoding`. Cocoa's UTF-8 decoder is designed to be pedantic rather than robust, and returns a null string if there is any ambiguity about the encoding, making it incompatible with RSS feeds and other Web sources, including those that generate plain ASCII.

Various solutions are possible. You can write code that cycles through a list of possible encodings, trying each in turn until it returns a possible match. A trickier solution is to download the Web information into `NSXMLDocument`, and ask it to report the encoding with the `characterEncoding` method.

Whichever solution you choose, be aware that creating robust support for the encodings used in Web text may not be a trivial problem, even without multi-language support.

Creating buffers for UTF-8 text

Each character in a UTF-8 string can have a variable length. If you need to convert a string into a byte representation, don't use the `length` property to set the buffer length because it assumes all characters are a single byte.

The `lengthOfBytesUsingEncoding:` method returns the byte count needed for a given string.

The `maximumLengthOfBytesUsingEncoding:` method returns the maximum possible byte count, assuming every character uses the maximum number of bytes. This is rarely useful.

If you need an estimate for a string with a given length, irrespective of content, multiply `length` by four and add 1 for the `NULL` terminator character.

Using NSAttributedString

By default, strings contain plain text with no styling or layout information. You can use `NSAttributedString` to manage styled text. This class includes a standard `NSString` object but adds a list of attribute records. Each record holds an `NSRange` and the attribute data for that range, which can include the font, text size, justification, and so on.

`NSMutableAttributeString` supports attribute editing. For example, to change the font style and size of the first three characters of a string, use

```
NSMutableAttributedString *theString =
[[NSMutableAttributedString alloc] initWithString: @"The
   String"];
[theString addAttribute: NSFontAttributeName
value: [NSFont fontWithName: @"Myriad Pro" size: 40.]
range: NSMakeRange (0, 3)];
```

Repeat with other attributes and values to style the string. To remove attributes, use

```
[theString removeAttribute: NS<AttributeName>
range: NSMakeRange (location, length)];
```

There's no direct way to remove all attributes in a range, but you can use `setAttributes:` with an empty dictionary to delete all existing attributes.

CAUTION

There is a full list of supported attributes in the Documentation. It's not in the Class Reference for either `NSAttributedString` or `NSMutableAttributedString`. You'll find it in the NSAttributedString Application Kit Additions Reference Class Reference.

Understanding attributes

Attributes implement advanced editing and styling features that are a superset of those found in standard text editors. Text attributes support the following:

- Font and size
- Underline in various weights and line styles
- Ligatures to support fonts that support joined characters such as œ
- Background, stroke, and strikethrough color
- Shadow effects
- Kerning control
- Hypertext linking
- Individual specialized character glyphs

Selected document attributes include:

- Author name
- Company name
- Document type
- Zoom and view mode
- Paper size and margins
- Copyright

Document types include plain text; RTF (Rich Text Format); RTFD, which combines an RTF text file with a folder of associated image attachments; Mac legacy; HTML; Word Doc; Open Office; and OpenDocumentText.

The type attribute is set when you save and load documents using one of the specialized methods listed next, but is otherwise ignored.

Saving and loading attributed strings

`NSAttributedString` supports `initWith<type>` methods for loading the following document types:

- Plain text
- RTF
- RTFD

- HTML
- MS Word

The MS Word importer isn't guaranteed to support the latest MS Word format. You can pass an optional dictionary pointer to an `initWith<type>` method to receive a list of document attributes extracted from the file.

To save a file, convert it to an `NSData` object using one of the conversion methods listed below, and then save the data object to disk. All formats support an optional dictionary of attributes that defines the attributes included in the file:

- `dataFromRange:` is a generic converter that supports all valid attributes and is ideal for Mac-only document saving.
- `docFormatFromRange:` creates an MS Word file, usually of a legacy type.
- `RTFFromRange:` creates an RTF file. For sample code, see the Nanopad example later in this chapter.
- `RTFDFromRange:` creates a folder that contains an RTF file and associated image attachments. This is a legacy format and isn't widely used.

Drawing and using attributed strings

`NSAttributedString` supports direct drawing. You can draw a string at a point or inside a `RECT` with the `drawAtPoint:` and `drawInRect:` methods. When your application has more than one window, the string is drawn into the current top view.

```
//Draw a string at the bottom left
[attrString drawAtPoint: NSZeroPoint];
```

Potentially, you can use this feature to animate strings and create Flash-like effects.

Drawing text on a path

There is no easy way to draw text on a path. You must draw each glyph on a path separately, applying size, rotation, and position transforms as needed.

The Documentation includes sample code for a project called CircleView that draws text on a circular path. It uses Cocoa's text storage, text layout, and text container classes, which are typically used to implement page design tools for desktop publishing and advanced PDF creation.

This approach isn't any simpler or more efficient than a character-by-character solution; it simply wraps the character-by-character code inside container objects that your application may not need.

Using attributed text in controls

Many controls support a `setAttributedTitle:` method. Use this method to label buttons and other controls with attributed text; for example, to set the label text color or add a text drop shadow.

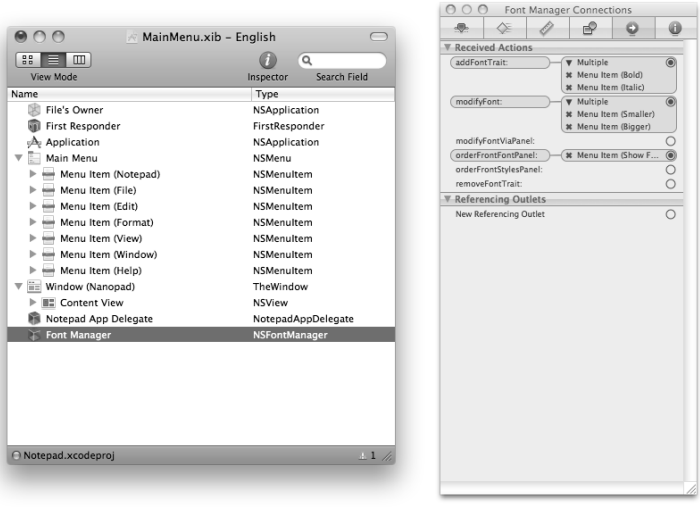
Creating Nanopad: A Rich Text Editor

Writing a text editor that supports attributes could be a significant challenge. Fortunately, you don't need to reinvent the wheel; the work has already been done for you in `NSTextView`. The basic default application template not only allows rich text editing, but also includes support for fonts, letter sizing, underlines and other attributes, and justification control.

To create an editor, drop an instance of `NSTextView` into the standard template's nib file. You already did this with the Picopad editor introduced in Chapter 10. If you reload the project and open its nib file, you'll see that the nib file includes a font manager, as shown in Figure 15.2. The font manager provides attributed text support.

Figure 15.2

The default application template includes an instance of NSFontManager. It's already linked to menu items that display the manager's font panel and generate attribute information for text editing.



Using NSFontManager

In the default template, the font manager is triggered by menu events. The default menu includes prelinked entries that generate the required events. For example, the Show Fonts item triggers the `orderFrontFontPanel:` method. When you run the application and send this message to the font manager, the font panel appears. You can drag+select a range of text and apply fonts and attributes from the manager.

The font manager isn't linked to the text view, so it's not obvious how the text view knows which fonts have been selected. When the user selects an attribute, the font manager sends a message to the responder chain. In this example, the window automatically passes the message down to the text view, which is the first object in the window's hierarchy to respond to attribute messages.

This means updates happen automatically, without linking, bindings, or Key-Value Observing (KVO). If you place a text-aware object in the responder chain, font management "just works," as long as the object is first in line for attribute messages.

If you experiment with the editor, you'll discover that undo and spell-checking are also built into the text view, as shown in Figure 15.3. The application even supports the Mac speech synthesizer and printing.

Figure 15.3

Applying fonts and attributes from the font manager. Spell check has noted that this isn't how you spell "Haettenschweiler." Or "has."

Saving and loading rich text

If you save the text with this version of the editor, you'll see that the styling information disappears. When you reload the text, the application applies the last selected styling to the entire file.

The original Picopad save/open code writes standard unattributed strings. To support attributes, save and load the file as rich text as shown below.

The open and save methods use the block structure introduced in Picopad, but the `RTFFromRange:` method is used to generate an RTF stream in an `NSData` object, which is saved to a file. The Open code is simpler. It uses the `readRTFDFromFile:` method to read the RTF data back into the text view.

CAUTION

Note that both methods run on the text view, not on a separate attributed string property of the text view. They're the same as methods used with `NSAttributedString`, but they're part of the `NSText` class, which is Cocoa's default text container object. `NSTextView` is a subclass of `NSText` and inherits its methods.

```
@implementation TheWindow
@synthesize theTextView;
-(void) awakeFromNib {
//Nothing happens here.
//You could add auto-load of the last saved file…
}
-(void) saveTheDocument: (id) sender {
   NSSavePanel *savePanel = [NSSavePanel savePanel];
   savePanel.allowedFileTypes = [NSArray arrayWithObject:@"rtf"];
   [savePanel beginSheetModalForWindow:self
   completionHandler:^(NSInteger result) {
      if (result == NSFileHandlingPanelOKButton) {
        NSData *theData = [theTextView RTFFromRange:
        NSMakeRange(0, [theTextView.string length])];
         [theData writeToURL: savePanel.URL atomically: YES];
         }
      }];
}
- (void) openADocument: (id) sender {
   NSOpenPanel *openPanel = [NSOpenPanel openPanel];
   openPanel.allowedFileTypes = [NSArray arrayWithObject:@"rtf"];
   [openPanel beginSheetModalForWindow:self
   completionHandler:^(NSInteger result) {
      if (result == NSFileHandlingPanelOKButton) {
        [theTextView readRTFDFromFile: openPanel.URL.path];
      }
   }];
}
@end
```

These few lines of code and the font manager create the core of a complete solution for editing, saving, and loading attributed text in `TheWindow` subclass of `NSWindow`, which was originally created in the Picopad example from Chapter 10.

NOTE

Printing is implemented by calling the `print:` method in `NSWindow`, which automatically opens a page setup pane, creates a print job, and manages pagination.

Three features are missing:

- File operations don't trap errors.
- There's no Save feature that re-saves the current file without showing the save pane.
- The Open Recent menu isn't implemented.

The first two are relatively easy to add, so they'll be left as reader challenges.

Implementing the Open Recent menu

To implement the Open Recent menu in an application with a single window, add the following boilerplate to the file open code, positioned so that it runs after the open operation successfully loads a file:

```
[[NSDocumentController sharedDocumentController]
noteNewRecentDocumentURL: openPanel.URL];
```

You'll see that the application is now maintaining the Open Recent list, but it isn't yet responding when the user clicks a recent file to reload it.

There are various ways to implement reloading, but the simplest is to add the following method to the App Delegate:

```
- (BOOL)application:(NSApplication *)theApplication
   openFile:(NSString *)filepath {
//Open code here
}
```

This method runs when a user selects an item in the Open Recent menu, passing the filepath as a string. You can duplicate the detailed open code here, or — more efficiently — extract the detailed open code from the open pane block and place it in a separate method. The active open code can then be called by the block, or from this method in the App Delegate. Note that this method is only called from the Open Recent menu. It doesn't run when the user opens a file by choosing File➪Open.

CAUTION

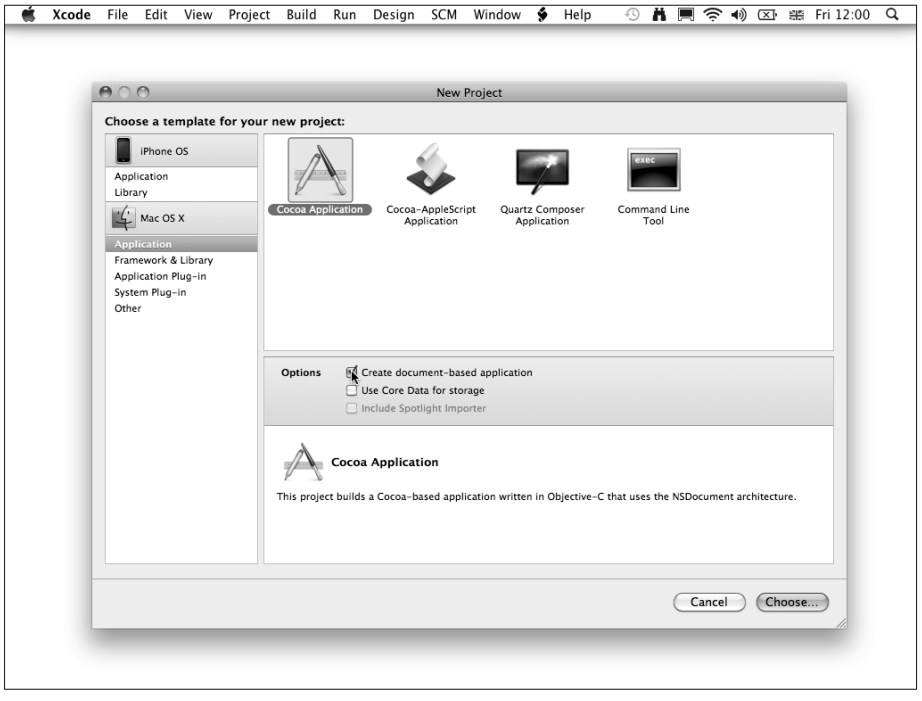
This method receives a string filepath. If your open code requires a URL, convert it with `fileURLWithPath:`.

Creating, Saving, and Loading Documents

Xcode includes an application template that outlines a multi-application document. You can select it by selecting the Create document-based application when you create a new project, as shown in Figure 15.4. Do this in Xcode and save the project as MultiDocument.

Figure 15.4

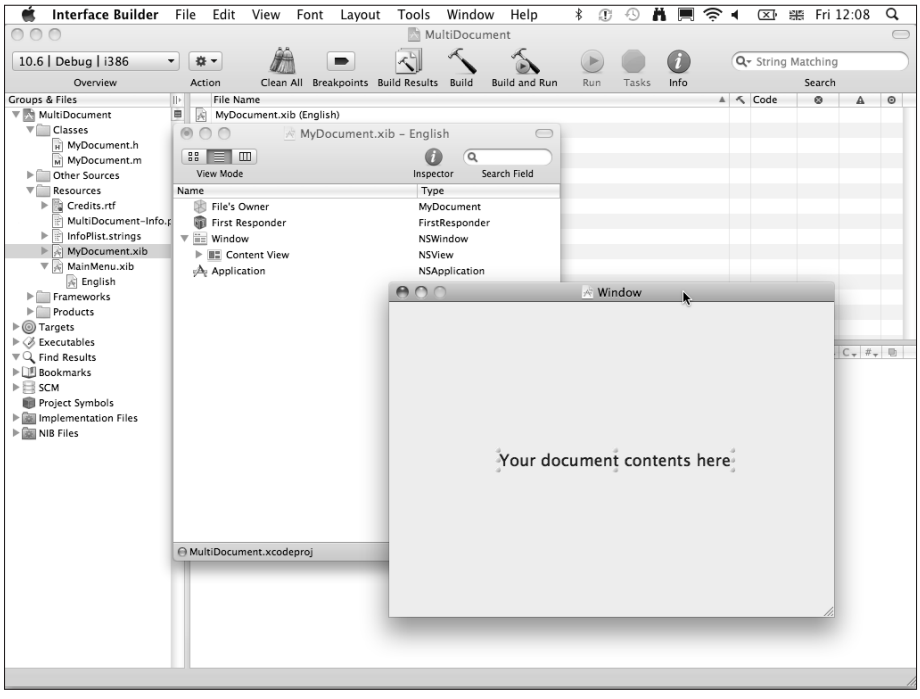
Creating a document-based application. The document template generates a project that uses the `NSDocument` class as a content container. Each document contains an `NSWindow`, with a content view.



The project structure is shown in Figure 15.5. The `MyDocument` class defines the code used to implement the document. Each document is a separate but — usually — identical instance. It runs the same code and its UI is defined in the MyDocument nib.

Figure 15.5

The structure of a document. Choosing File➪New automatically creates a new instance of the `MyDocument` class, which loads the associated nib.



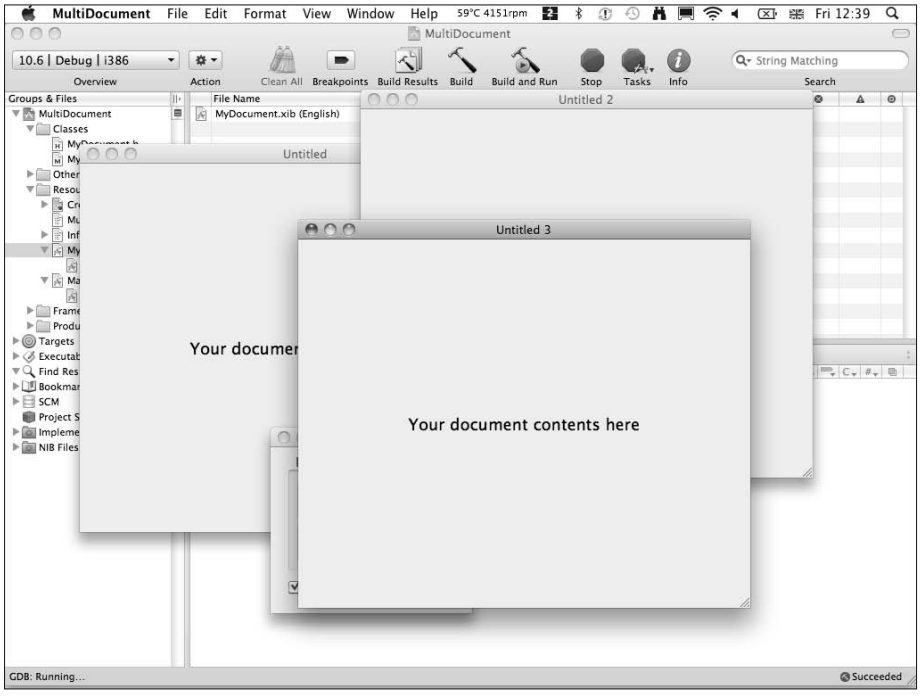
NOTE

There's no App Delegate. If your application needs one, create an App Delegate subclass of `NSObject` for the code, and add a corresponding subclassed `NSObject` to the MainMenu nib to create an instance. Link it to the delegate outlet in File's Owner or Application.

Build and run the application and choose File➪New a few times. You'll see something like Figure 15.6, with multiple identical document windows. Each window shows the MyDocument nib.

Figure 15.6

The document template is already set up to create a new window whenever you choose File➪New.



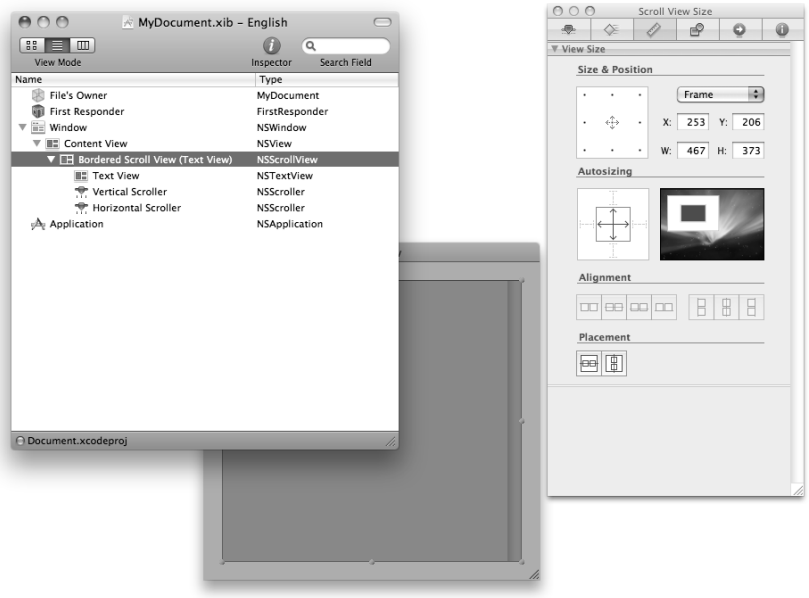
Some features are already implemented. The Window menu supports zooming, minimizing, restoring, and Bring All To Front. The Format menu displays the font manager panel. Choosing File➪Page Setup displays a printer options sheet. File options are active, but not yet supported by code.

Creating a default nib file

You can change the UI of the documents by editing the MyDocument nib file. For example, to create a simple multi-document version of Nanopad, add a scrolling text view, as shown in Figure 15.7. Set Autosizing to track the window frame.

Figure 15.7

Modifying the document nib. The nib defines the UI of every document created by the application.



Save the modified nib, then build and run the application again. It now loads multiple text views. Add some text to a view, choose Format➪Font➪Show Font to display the font manager pane, shown in Figure 15.8, and style the text.

Choose File➪New to create another document. Add some text and style it. You'll see that the font manager automatically applies its styling commands to the active document. You don't need to add code or links to make this happen — it happens automatically.

Setting document types

Saving and loading aren't yet implemented. If you try to save a file, you'll see a save pane, but the file extension is set to question marks.

Before you can save and load a document, you must define at least one supported file type. In the Groups & Files pane, click the reveal triangle next to the Targets icon and right-click the MultiDocument item inside it. You'll see the dialog shown in Figure 15.9. Select the Properties tab. In the Document Types subpanel, you'll see a single default entry called documentType (not shown here). Click on each column in turn to edit the entry until it matches the figure.

The most important change is the UTI (Universal Type Identifier) column, which must say **public.rtf**. This tells the document that it is an editor for the standard system rtf file type. A UTI holds information about each file extension used by the system. Apple predefines UTIs for standard file types, and you can also create your own for files with a customized extension. For details, see the Introduction to Uniform Type Identifiers Overview in the Cocoa Documentation.

Figure 15.8

The font manager sends styling information to the current active document automatically.

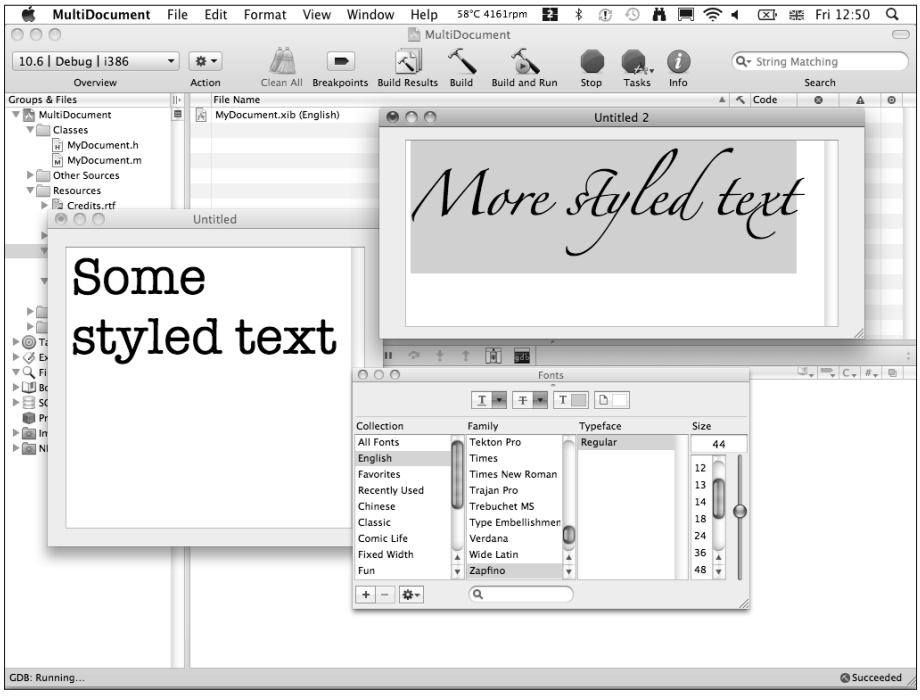
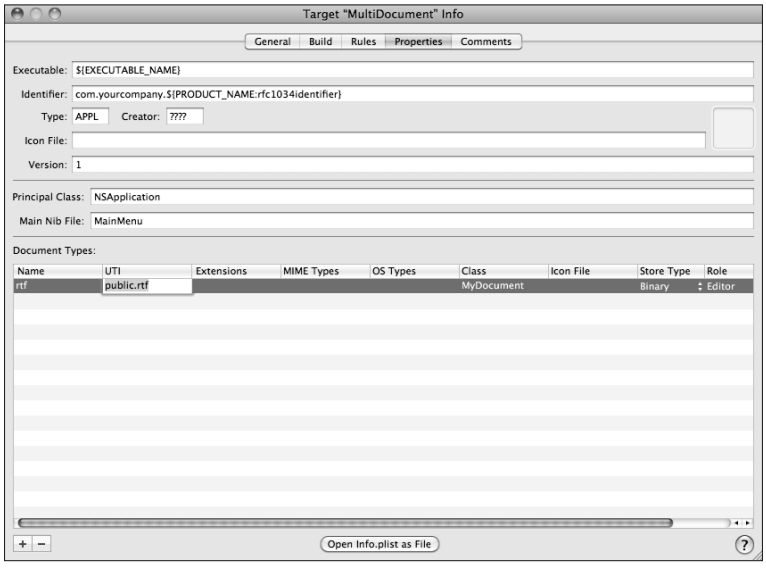


Figure 15.9

Assigning a standard system file type to the document



Adding a custom UTI is a labor-intensive process. You must define it in your application's `info.plist` file, which is an XML document with a variety of application settings. Xcode includes a plist editor, but it's easier to navigate to your project's folder in Finder, open the plist file in a text editor, copy the UTI boilerplate XML from Apple's documentation, paste it into the plist file, and make minor edits by hand. Sample XML to create a custom UTI that implements an `abcd` file extension follows:

```
<key>UTExportedTypeDeclarations</key>
        <array>
            <dict>
                <key>UTTypeIdentifier</key>
                <string>com.myURL.applicationname</string>
                <key>UTTypeDescription</key>
                <string>My custom document format</string>
                <key>UTTypeConformsTo</key>
                <array>
                    <string>public.data</string>
                </array>
                <key>UTTypeTagSpecification</key>
                <dict>
                    <key>com.apple.ostype</key>
                    <string>abcd</string>
```

```
                <key>public.filename-extension</key>
                <array>
                    <string>abcd</string>
                </array>
            </dict>
        </dict>
    </array>
```

Implementing save and open code

Code that creates a save and open sheet is already built into the template. When you choose File➪Save, you'll see a save sheet with your newly adopted file type.

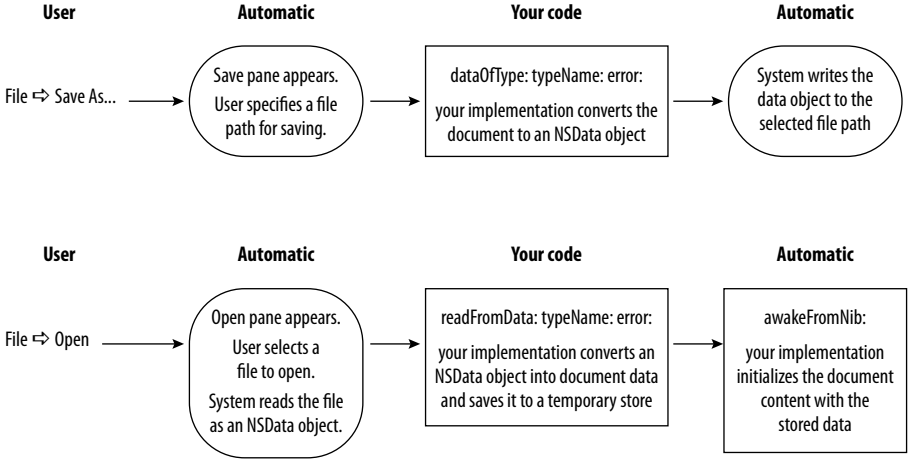
To save or open a file, do *not* access a file directly. File access is implemented for you. Inside `MyDocument.m` you'll see two methods: `dataOfType:` and `readFromData:`. These methods are left empty. Your code must create an `NSData` object from the document data, and recover document data from an `NSData` object.

Saving is straightforward. Your code is responsible for conversion, but the rest of the process is automatic.

Opening is less straightforward. When `readFromData:` runs, the new document hasn't yet been created, so you can't initialize its contents, because there's nothing to initialize. The solution is to save the document data to a temporary store in `readFromData:`. Then in the `windowControllerDidLoadNib:` method, add code to copy data from the store to the new document. This guarantees that the document is initialized correctly after it's created, as illustrated in Figure 15.10.

Figure 15.10

Understanding how the default save and open operations work, and how to add supporting code



Sample code for the `MyDocument` class follows. If your application supports a single data type, such as the rtf type used here, you can ignore the `ofType` parameter. For multiple data types, add conditionals to read the type and select different conversion code for each.

```
@implementation MyDocument
@synthesize theTextView;
NSMutableAttributedString *theString;
NSMutableDictionary *aDictionary;
- (id)init
{
    self = [super init];
    if (self) {
    }
    return self;
}
- (void)windowControllerDidLoadNib:(NSWindowController *)
   aController
{
   [super windowControllerDidLoadNib:aController];

   //Copy the loaded data to the new text view
   // text storage is the text view's container object
   if (theString)
      [[theTextView textStorage] setAttributedString: theString];
}
- (NSData *)dataOfType:(NSString *)typeName
                 error:(NSError **)outError
{
    if ( outError != NULL ) {
      *outError = [NSError errorWithDomain:NSOSStatusErrorDomain
   code:unimpErr userInfo:NULL];
   }
   //Return the converted data as an NSData object
   return [theTextView
   RTFFromRange: NSMakeRange(0, [theTextView.string length])];
}
- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName
   error:(NSError **)outError
{
   //The dictionary is optional. Use it check or list attributes.
   aDictionary = [[NSMutableDictionary alloc] init];
   //Convert the data into a temporary string variable
   theString =
   [[NSMutableAttributedString alloc] initWithRTF:data
   documentAttributes: &aDictionary];
    if ( outError != NULL ) {
```

```
        *outError = [NSError errorWithDomain:NSOSStatusErrorDomain
        code:unimpErr userInfo:NULL];
}
    return YES; //Or NO if there was a problem
}
@end
```

When an application uses the document-based template, the Open Recent feature is implemented automatically. When you choose File➪Open Recent, the application calls the same open code defined previously.

Printing documents

The easy way to print a document is to design your document with a container view, and then override the `printDocument:` method in `NSDocument` with a custom call to `print:`.

```
-(void) printDocument: (id) sender {
 [theContentView print: nil];
}
```

NOTE

`print:` takes a nominal `(id) sender` parameter but ignores it.

This launches the default print dialog, manages automatic pagination and margins, and runs the print job for you with very little code. You can also render a document to a single off-screen view and run `print:` on the view. The default printer is the standard OS X printer, and it implements a print-to-PDF feature.

Occasionally you may need to print a document the hard way. Customized printing gives you control over pagination, margins, and page scaling, and supports very long documents with unlimited pages.

A complete customized printing solution can be moderately complex. The core print operation code looks like this:

```
NSPrintInfo *printInfo;
NSPrintOperation *printOp;
printOp = [NSPrintOperation printOperationWithView: viewToPrint
                                         printInfo: [self printInfo]];
[printOp setShowPanels: NO]; //Optional
[printOp runOperation];
```

The `printInfo` object stores information about margins, paper size, orientation, and pagination. For example, you can set the pagination to clip, scale/fit in either dimension, or automatically create a column of pages.

Selecting `[self printInfo]` displays a panel that enables the user to define the print options. You can disable this feature by setting `showPanels` to `NO`.

To create manual pagination, implement two methods in your printable view:

```
-(BOOL) knowsPageRange :(NSRange *) theRange {
   theRange -> location = 1;
   theRange -> length = <estimate number of pages here>
   return YES;
}
-(NSRect) rectForPage: (int) page {
return <page rect calculated for a given page number>
}
```

`theRange` is a pointer to a range `struct`, which is passed when the method is called and contains returned values when the method completes.

Calculating the `rect` to return may require information about the paper size, required margins, scaling and optional borders, crop marks, page numbers, header and footer strings, and so on. There is no general solution; each application has different requirements. For very long documents, you can use a single `rect` as a buffer, and render the content for each page with `drawRect:`.

Using NSUndoManager

The text view in the MultiDocument project implements undo automatically. This is a feature of the text view, and not the document. To implement a full undo solution, you need to add extra code to every editing operation.

`NSDocument` supports an `NSUndoManager` class that can maintain a separate undo/redo stack for each document. If you initialize the undo manager correctly before each edit, this stack is managed automatically for multiple levels of undo and redo.

To get a pointer to a document's undo manager, use

```
NSUndoManager *theManager = [self undoManager];
```

There are two ways to implement undo. For simple operations, you can set the target/action that the undo will perform, and the object that the undo will perform it on.

```
[undoManager
registerUndoWithTarget: targetObject
              selector: @selector(methodToImplementTheUndo:)
                object: objectThatWillBeRestored];
```

`targetObject` is usually `self`. When the user selects Undo, the result is

```
[targetObject methodToImplementUndo: objectThatWillBeRestored];
```

Your code must implement the method and manage the object in a way that creates the undo. It's possible, with some effort, to create edit methods that embed this code and store the old pre-edit value in the object. This is a good solution for simple objects, but it can become unwieldy for complex multiple edits.

If `object` is a complex composite object such as a video sequence, it's impractical to save a backup copy of the complete object for each undo step. An alternative undo technique uses a class called `NSInvocation`, which stores a complete message including the target object, the message sent, and all parameters.

```
[[undoManager prepareWithInvocationTarget: targetObject
                                  aMethod: undoParameters];
```

Again, `targetObject` is usually `self`. When the user selects Undo, the result is

```
[targetObject aMethod: undoParameters];
```

Typically before each edit, you create an invocation that restores the pre-edit values of the parameter or property that is about to be changed. Alternatively, you can create an invocation that does the opposite of some edits; for example, decreasing a value that has just been increased.

For a complex edit, the undo parameters are likely to include arrays and dictionaries of previous values. Implementing a complete undo solution for every possible edit in an application requires significant extra code.

Localizing Applications

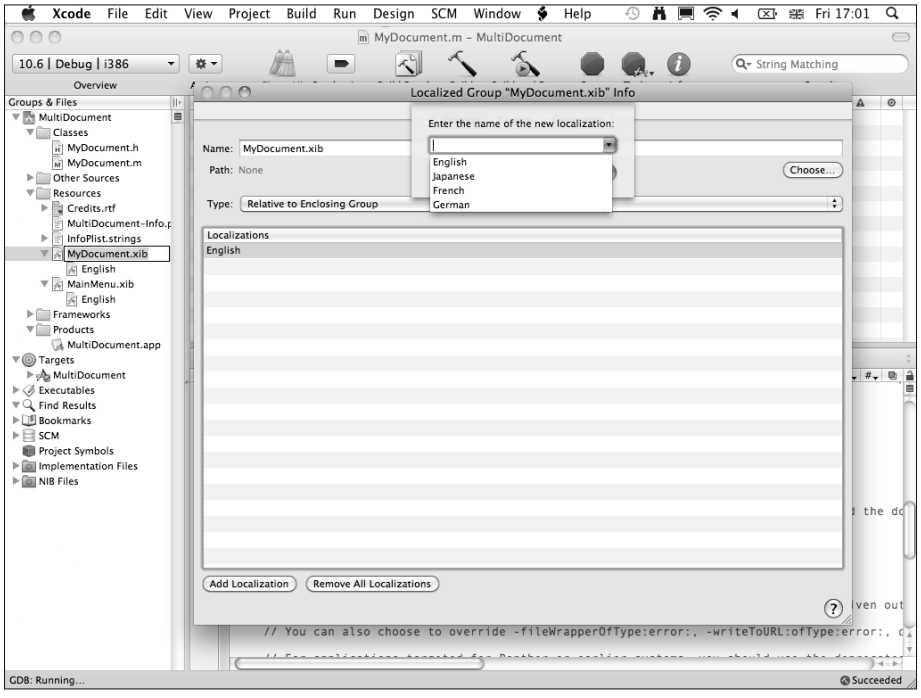
For commercial projects, you can improve sales by adding multi-language support. Supporting multiple languages is known as *localization* and is implemented in two complementary ways: localized nib files and localized strings.

Creating a localized nib file

Right-click a nib file and select Get Info. Select the General tab if it isn't already selected. Select Add Localization at the bottom left of the window, and pick a language from the drop-down list in the sheet that appears at the top of the window, as shown in Figure 15.11.

Figure 15.11

Creating a localized nib file. The default selection of languages is limited.



This creates a new French nib, which joins the existing English nib. To localize the nib, replace all static labels and titles with French text, as shown in Figure 15.12. When you build and run the application and the user selects Français as her preferred language, the French nib is loaded automatically. Repeat the process for the MainMenu nib.

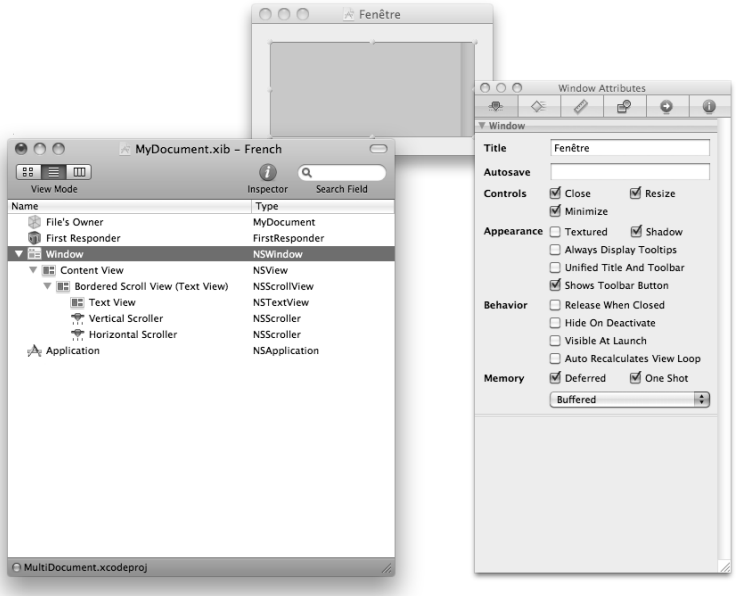
To enter accented text, select the characters and insert them using the Character Viewer sheet. This is a standard Mac feature and should appear under your usual language selection, as shown in Figure 15.13.

Creating localized strings

Rarely, a localized nib may solve the localization problem. More typically, you must also add *localized strings* to localize prompts and error messages. Localized strings are saved in the application bundle. Each language has a separate file.

Figure 15.12

Replacing static text



To create a file, right-click Resources and choose Add➪New File. Select Mac OS X Resource and Strings File, as shown in Figure 15.14.

The file is saved as **Localizable.strings,** as shown in Figure 15.15. The filepath should be **English.lproj** for an English file, **French.lproj** for a French file, and so on. This creates a `Localizable.strings` item in Xcode with separate files for each language.

Before you edit a file with non-English text, convert it to the UTF-8 encoding. Right-click it, select Get Info, and select the encoding from drop-down menu, as shown in Figure 15.16. Select the Convert option at the alert.

You can edit each file in Xcode. Strings are stored in key-text pairs.

```
"key" = "localized text";
```

Figure 15.13

The Character Viewer is an optional feature built into most language selections.

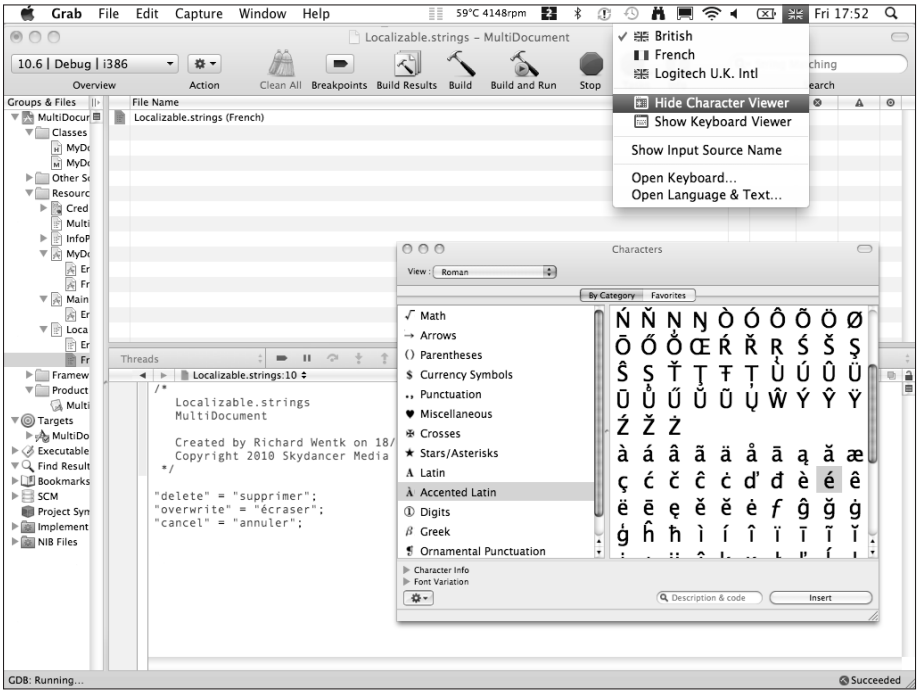


Figure 15.13 showed part of a French `Localizable.strings` file. The key appears in the code in your application. When the application runs, it selects the appropriate `Localizable.strings` file for the user's location, looks up the text for each key dynamically, and displays it. Optionally you can make the key more recognizable by writing it in caps, prefixing a special character, suffixing the word "key," and so on.

To use the localized text, replace the usual string reference with the following:

```
NSLocalizedString(@"key", @"default text")
```

If the application finds a localized string that matches Key, it displays it. If not, it displays the default text.

NOTE

`NSLocalizedString` is a macro that runs `localizedStringForKey:` on `[NSBundle main Bundle]`. You can ignore the macro and write your own implementation if you need to add special features to the lookup, but typically, the macro solves the problem.

Figure 15.14

Creating a Localizable Strings file

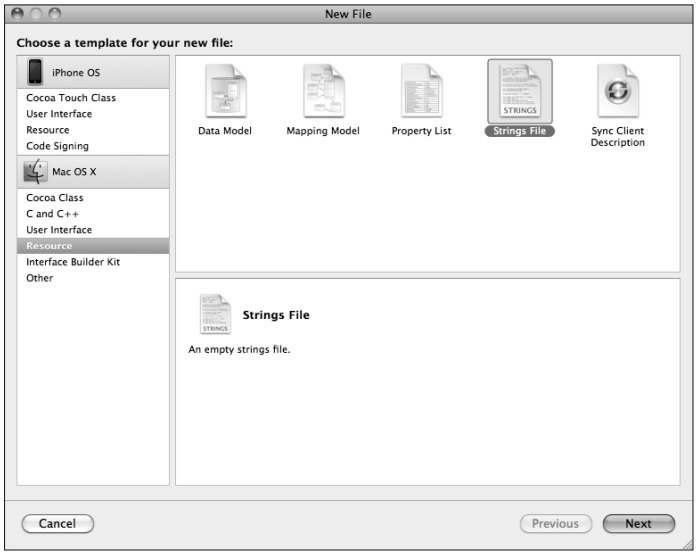


Figure 15.15

Saving the file for a specific language

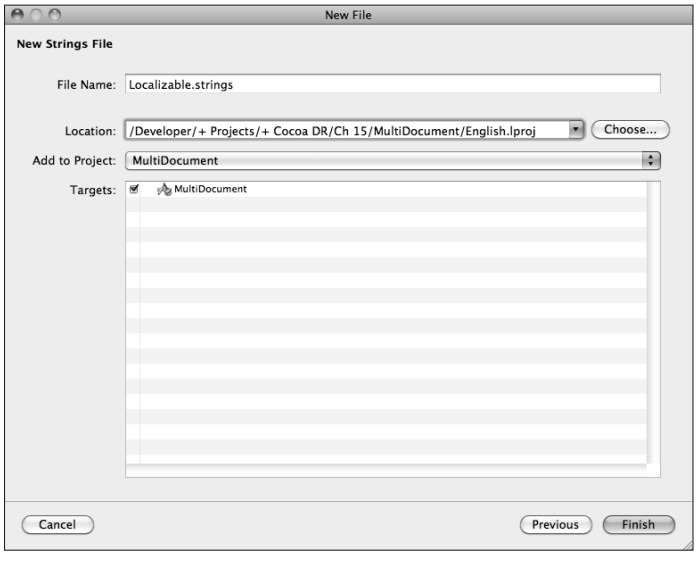
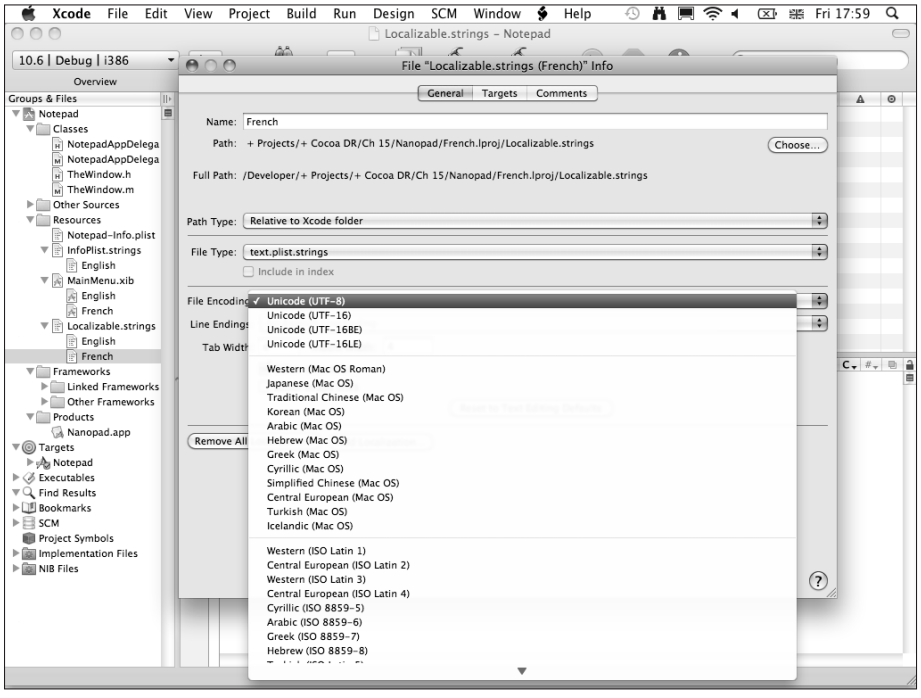


Figure 15.16

Selecting UTF-8 encoding to support accented and non-Latin text



Summary

In this chapter, you learned about strings, attributed strings, and documents. You were introduced to some of the key methods supported by `NSString` and `NSMutableString`, and you learned how to create and edit rich text by using `NSAttributedString` and `NSMutableAttributedString`.

You explored Xcode's multi-document application template and discovered how to add powerful multi-document support to your application with very little code. You explored the `NSUndoManager` and learned about the essential steps required to implement a full undo solution.

Finally, you learned about localized strings and language-specific nib files, and you discovered how to create an application that supports multiple languages and selects language string and nib files automatically.

Expanding the Possibilities

In This Part

Chapter 16
Managing Views and Creating 2D Graphics

Chapter 17
Creating Animations and 3D Graphics

Chapter 18
Debugging, Optimizing, and Managing Code

Chapter 19
Developing for the iPhone and iPad

16 Managing Views and Creating 2D Graphics

Cocoa and Cocoa Touch have impressive graphics capabilities, but from a developer's point of view, the graphics frameworks are a maze of semicompatible technologies with inconsistent interfaces. The designers of each framework have reinvented fundamental concepts, such as size, position, and orientation, and packed them into incompatible data structures.

One of the most frustrating challenges in graphics programming is the almost constant need to move data between objects and data structures that should be "toll-free bridged," but aren't.

For example, Cocoa's `NSRect` and Core Graphics' `CGRect` structures define a rectangle with identical components: an origin, a width, and a height. In spite of the similarities, you can only move data between them by calling a pair of conversion functions: `NSRectToCGRect` and `NSRectFromCGRect`.

Similarly, `NSPoint` and `CGPoint` — two data structures that define an x, y coordinate — are incompatible, even though the only difference between them is their name.

The function that packs two floats into a `CGPoint` is called `CGPointMake`. The equivalent function that creates an `NSPoint` is called `NSMakePoint`.

Image support is also inconsistent. Cocoa defines an `NSImage` type for handling image data. The Core Graphics framework uses a different `CGImage` type. `CoreImage` filters use yet another `CIImage` type. Functionally, these data types are used in equivalent ways. You can load them with data from a file or other data source and draw the data into a view, but their contents aren't compatible or interchangeable.

One final source of confusion is that Cocoa's graphics system combines data structures and objects. `NSRect` is a data structure. `NSBezierPath`, used to define paths and shapes, is an object. You don't need to `alloc/init` an `NSRect`, but you do need to allocate and initialize an instance of `NSBezierPath`. Inevitably, this creates confusion.

16 In This Chapter

Understanding the view hierarchy

Handling mouse events

Understanding geometry in Cocoa graphics

Creating and drawing paths

Using CoreImage filters

Unfortunately these inconsistencies are unavoidable and there is no way to work around them. The graphics frameworks do not support low-level access to useful values; for example, there is no equivalent of a point, line, or rectangle plot with literal `float` or `int` values. To use the frameworks, you must pack values into data structures and objects, and then pass the packed data to a function or method.

While it's possible — barely — to solve many problems exclusively with Cocoa's own graphics features, more typically you'll use a combination of frameworks. To do this successfully, you must understand the different data structures and objects that are used, and you must be familiar with the many data packing, conversion, and translation helper functions that are available. You must also understand how the frameworks use the basic graphics canvas: the view object.

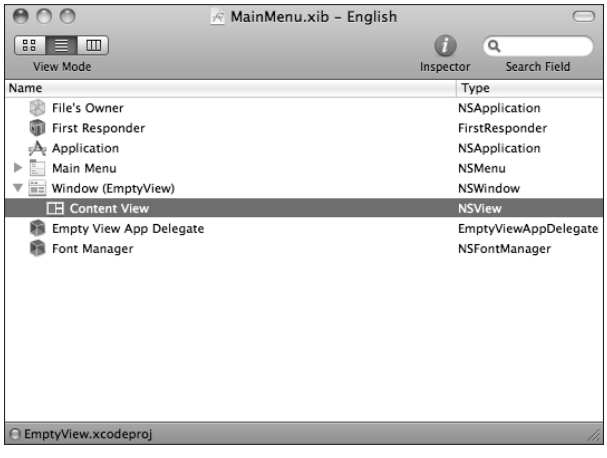
Understanding Windows and Views

The typical structure of a Cocoa UI is shown in Figure 16.1. The window is the master container. It handles window-specific messages, such as window drag events and minimize, open, and close operations.

You can design an application by adding visible objects to a window. More typically, you use a *content* or *container* view, and place the objects inside the container. This is optional for very simple applications with a single window and a very minimal feature set. In more complex applications, it simplifies event handling and makes it possible to fill the view with custom graphics and animation effects, drawn with methods and functions that aren't supported in `NSWindow`.

Figure 16.1

A typical UI structure, with a content view inside a window. This structure is created automatically when you create a new project from the blank template.



The container view is usually subclassed so that you can add code to it to handle mouse and other events. Optionally, you can also create custom graphics and animation effects. Sometimes it's useful to subclass the main window object as well, but this is done less frequently.

The container view holds a tree structure of subviews, as shown in Figure 16.2. It may also include other custom graphics. Subviews typically manage the UI and allow user interaction. Custom graphics can include image files loaded from disk or from the Internet and dynamically generated shapes with colors, textures, and gradients.

Figure 16.2

An example of the view structure from a real application: the Twirl Filter project described later in this chapter. The main view has been subclassed and it contains a selection of interface objects that create a UI. It also contains another subclassed view that generates custom graphics: a filter effect applied to an image.

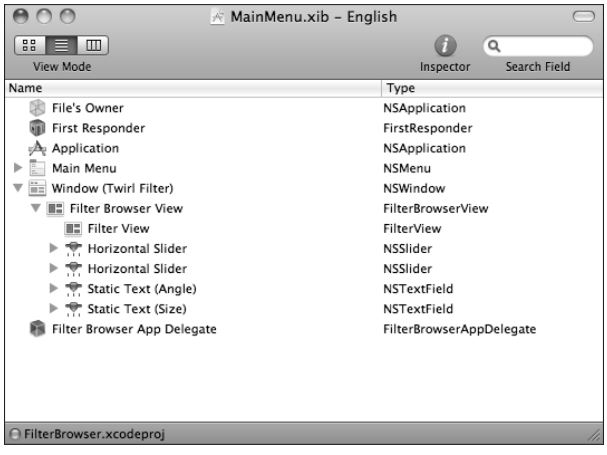


Table 16.1 summarizes the different objects associated with views.

Table 16.1 Window and View Objects and Their Applications

Object	*Application*
`NSWindow`	Sets the application title, handles open/close/minimize/maximize events. Can handle mouse events, but usually doesn't, except in very simple applications.
`NSPanel`	A special simplified floating auxiliary window for application panes and panels. Can be used for pop-up alerts.
`NSView`	The main view objects. Usually subclassed. Handles mouse events and other UI events. Includes a `drawRect:` method that can draw custom graphics. Can include other view objects to create a view hierarchy that can be modified dynamically.

continued

Table 16.1 Continued

Object	*Application*
`NSControl`	A superclass of objects that create user controls, including buttons, sliders, and other UI objects. Controls are placed "inside" the view hierarchy. They generate user events that are handled by the view.
`NSViewController`	An optional invisible container object for views. Used to switch views, often by loading them from a nib on demand. Setting a controller's `view` property makes a view visible. Also used to implement advanced page and print effects in document-based applications.
`NSResponder`	An abstract class that defines the messages and events that any view can handle. Implements message handling by adding `NSResponder` method signatures to a view and filling them out with custom code.

Understanding the view hierarchy

The nib structure illustrates how the view hierarchy is assembled. When you place an object "inside" a view, it's added to a tree structure. The main view object is the root of the tree. Other objects, called subviews, are placed lower down the tree.

When you load a nib, you load this structure and it appears in the application's window. But it can be modified dynamically to add and remove objects. Views can also be repositioned and resized dynamically. To add code that can implement these features, you must subclass the root view.

Subclassing the root view

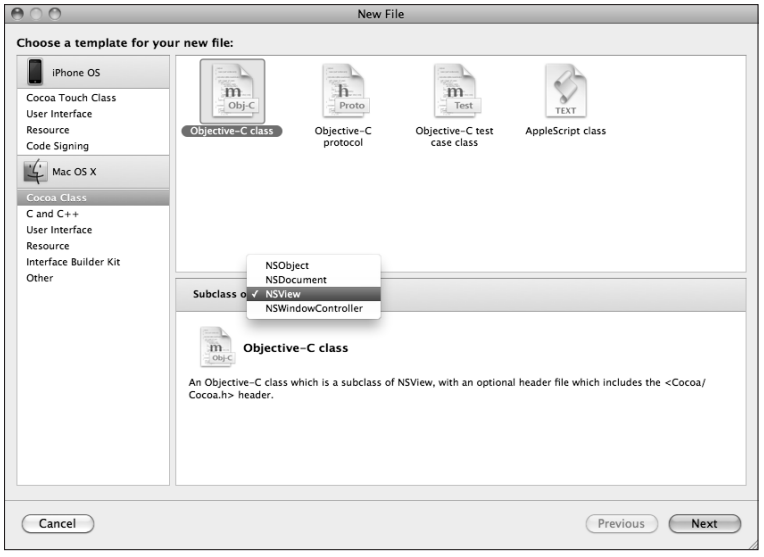
To subclass the root view, begin by creating a new project. Save it as **EmptyView.** By default, it has the nib structure shown in Figure 16.1. Ignore the MainMenu item — in this chapter you'll concentrate on the contents of the window.

To subclass it, right-click the Classes group and choose Add➪New File. Select the Mac OS X Cocoa Class option, and then select NSView in the pop-up menu in the middle of the window, as shown in Figure 16.3. Save it as **NewView.**

Figure 16.4 shows the result. The `NSView` template includes a couple of extra methods that aren't included in the file when you subclass `NSObject`. The `initWithFrame:` method is called when a new instance of the view is allocated.

Figure 16.3

Creating a subclass of `NSView`. Selecting the `NSView` template generates a file with extra features that aren't included in the standard blank `NSObject` template.



NOTE

Because you never — or hardly ever — generate more than one instance of a root view subclass, the `initWithFrame:` method is never called. This code block isn't notable for its usefulness, although it sometimes holds initialization code in a multi-document application.

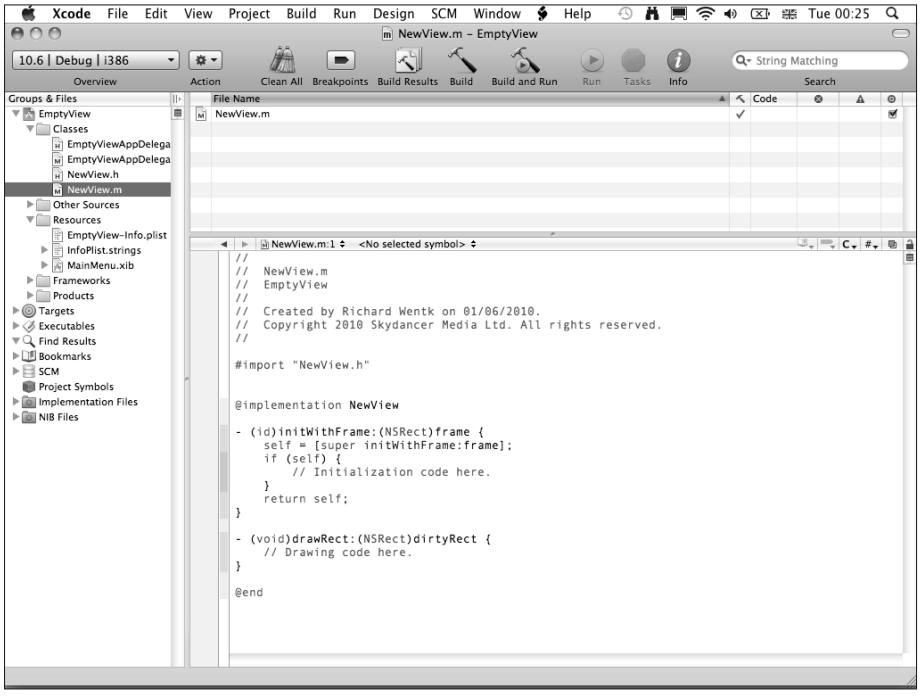
The `drawRect:` method is a fundamental feature of `NSView`, and it is used as a wrapper for custom graphics code. The method is triggered automatically to refresh the view when the surrounding window is moved, opened, or resized. You can also trigger it on demand when you need to refresh the graphics for some other reason, such as a user event.

TIP

You can create animations by triggering `drawRect:` repeatedly using a timer. For more on this, see the next chapter.

Figure 16.4

The result



The template *doesn't* include two other methods that are used in almost every application. You must add these methods by hand. Other methods are optional, but almost all projects use `awakeFromNib:` and `mouseDown:`.

`mouseDown:` is triggered by mouse events. The complete signature is

```
- (void) mouseDown: (NSEvent *) theEvent;
```

`theEvent` is an instance of `NSEvent` and includes information about mouse position and button status. The equivalent for mouse drag events is `mouseDragged:`. The signature is similar. Information on extracting position information from mouse events appears a little later in this chapter.

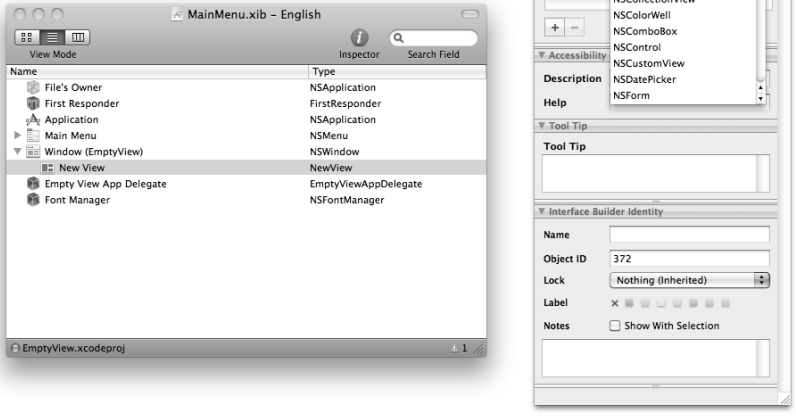
`awakeFromNib` is triggered when the view loads. Use it to run initialization code when the application starts. There are no parameters and the signature is

```
- (void) awakeFromNib;
```

To finish subclassing a view, open Interface Builder (IB), select the view object, select the Identity tab in the Inspector window, and choose your new class name from the pop-up menu, as shown in Figure 16.5. This tells the application to run the code in the new files. Your new view can now respond to events, and you can also add code that manages the view hierarchy or generates custom graphics.

Figure 16.5

Assigning the subclass in IB. This step was introduced in Chapter 7, but it's worth repeating it here; it's easy to forget, but your subclass won't work without it.

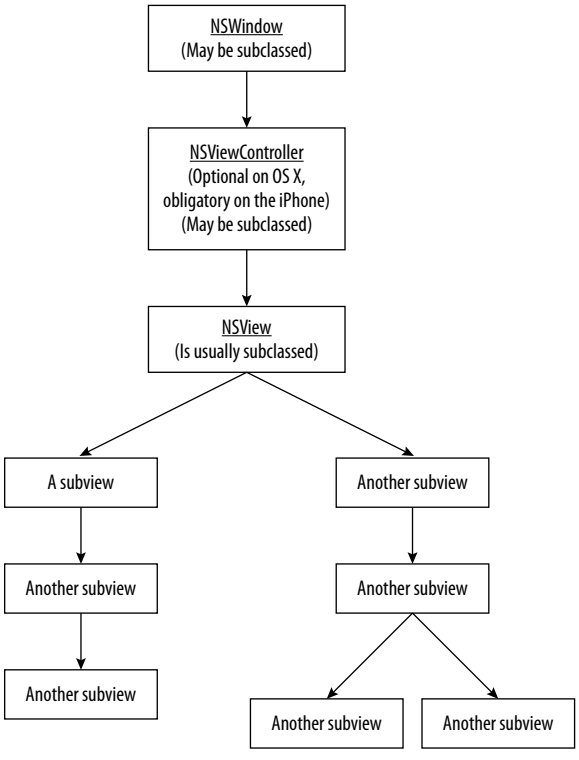


Adding and removing views from the view hierarchy

The view hierarchy is immensely powerful. Figure 16.6 shows an alternative view of one possible view hierarchy, illustrating the tree structure. You can load a nib and leave the hierarchy unchanged. But you can also redefine the hierarchy at runtime, creating completely dynamic context-dependent interfaces.

Figure 16.6

Exploring the tree structure of a view hierarchy. You can create a structure like this in IB and save it in a nib, or you can generate it dynamically to populate an empty view. Any item can be subclassed as needed. An item above another on a given branch is called a superview. An item below another on a branch is called a subview.



Most applications take the first approach, and the UI is loaded as is from a nib. The second approach is more powerful and can create some unique effects, such as dynamic, floating, animated menu items, reconfigurable button panels, and grids with a variable cell count.

Manipulating the hierarchy is simple. To create a new subview, use `alloc/init` to create a new view object. Add code to set a selection of default properties.

Alternatively, you can create a separate nib file for each object, saving it with default settings. Using a nib makes it possible to load an existing hierarchy as a composite object and insert it into the root view in a single operation.

TIP

You can think of each level in the hierarchy as a new root object. It's possible to add and remove items at every level.

To insert a subview and make it appear inside the container view in the window, use

```
[rootView addSubview: newSubview];
```

To remove a subview, use

```
[subview removeFromSuperview];
```

To remove a subview and replace it with a new view, use

```
[rootView replaceSubview: oldView with: newView];
```

To traverse the hierarchy, use a subview's `superview` property to return the object one level up, and use the `subviews` method to return an array of the objects one level down, or `nil` if the subview is at the end of a branch. Looking at Figure 16.6, you can see that these methods move you up and down the tree.

CAUTION

When using `superview`, keep a separate record of the root view to make sure you don't try to find the superview of the root. If you do, you'll `superview` yourself right out of the hierarchy into unprotected memory.

For a demonstration, see the MultiButtons project available on this book's Web site at `www.wiley.com/go/cocoadevref`. The project loads an empty content view from a nib and populates it dynamically with a selection of randomly placed buttons. Each button is numbered, and the button style is set to a useful default. The buttons are stored in an array. Randomizing the buttons removes the old button array and creates and displays a new one.

The code for the subclassed view is:

```
#import "ButtonsView.h"
NSButton *newButton;
NSButton *thisButton;
NSRect targetRect;
float buttonWidth = 60;
float buttonHeight = 30;
int buttonCount = 10;
@implementation ButtonsView
-(void) awakeFromNib {
//Seed the random number gen
 srand([[NSDate date] timeIntervalSince1970]);
//Find the target rectangle, allowing for the button anchor
//which is at the lower left
```

```
targetRect = NSMakeRect(0, 0, self.bounds.size.width-buttonWidth,
   self.bounds.size.height-buttonHeight);
[self drawSomeButtons];
}
-(void)  mouseDown: (NSEvent *) theEvent {
[self removeTheButtons];
[self drawSomeButtons];
}
-(void) drawSomeButtons {
 for (int i = 0; i < buttonCount; i++) {
 //Create a button at a random point
 CGPoint thisPoint = CGPointMake(arc4random() %(int)targetRect.
   size.width, arc4random() % (int)targetRect.size.height);
 newButton = [[NSButton alloc]
   initWithFrame:NSMakeRect(thisPoint.x, thisPoint.y,
   buttonWidth,buttonHeight)];
  //Define the button style and set the title string
  [newButton setButtonType:NSMomentaryPushInButton];
  [newButton setBezelStyle: NSRoundedBezelStyle];
  [newButton setTitle: [NSString stringWithFormat:@"%i", i+1]];
  //Set a target/action method that
  //is triggered when the button is clicked
   [newButton setAction: @selector(buttonWasPressed:)];
   [newButton setTarget:self];
   //Add the button to the container view
   [self addSubview:newButton];
   }
}
-(void) removeTheButtons {
//Get an array of subviews
  NSArray *theSubviews = [NSArray arrayWithArray:[self
   subviews]];
//Remove them all with a single line of code
[theSubviews
  makeObjectsPerformSelector:@selector(removeFromSuperview)];
}
-(void) buttonWasPressed: (id) sender {
 //When a button is pressed, log the title
 thisButton = (NSButton *) sender;
 NSLog(@"Pressed: %@", thisButton.title);
}
- (void)drawRect:(NSRect)dirtyRect {
 // Not used in this example - we're not drawing custom graphics
}
@end
```

Figure 16.7 shows the modified view. When you click a button, a single common method logs its title string — which also happens to be its number — to the console. When you click elsewhere in the view, the button positions are randomized.

Figure 16.7

A view with random buttons. Click in the view to change the arrangement. The console logs the button number when a button is clicked.

Key points in the code include:

- Calling the `removeFromSuperview` method on any object removes it from the view hierarchy. It also releases its memory and removes it from the display.
- Calling the `addSubview:` method on the root view passing an object as a parameter adds the object to the hierarchy and displays it.
- Buttons are created with `alloc/init` followed by extra initialization code. They're added to the view by calling `addSubview:` on them.
- The buttons are live. Setting a target and an action for every button links it to a method called `buttonWasPressed:` that is triggered when the user clicks a button. It's possible to read the button number from the button title text.

- Running the `makeObjectsPerformSelector:` method on the buttons in the array removes them all with a single line of code. No enumeration is needed.
- The geometry structures `NSRect` and `CGPoint` represent a rectangle and a point, respectively. These structures are explained in more detail next.

TIP

This application doesn't try to avoid button collisions; sometimes buttons overlap. Try to modify the project so that buttons can never appear on top of each other. Various solutions are possible. Some enumeration may be needed.

Handling mouse events in views

It's often useful to find the mouse position, either during a click or a drag event, or by following the cursor within a view. `NSView` supports a variety of mouse-tracking methods. Most are defined in `NSResponder`. To add them to a project, implement them with custom handlers as shown here. You can also download the mouse handler project from the Web site (`www.wiley.com/go/cocoadevref`) to see worked examples.

`mouseDown:` events are captured automatically. You can retrieve the coordinates within the window with

```
- (void) mouseDown: (NSEvent *) theEvent {
NSPoint mouseLocation = [theEvent locationInWindow];
NSLog (@"Mouse position: %.0f, %.0f", mouseLocation.x,
   mouseLocation.y);
}
```

It's often useful to translate the coordinates to those of a subview and to limit clicks so that the application ignores them when they're outside the subview. To translate coordinates, use

```
NSPoint mousePositionInPanel =
[self convertPoint: mouseLocation toView: aTargetSubview];
```

After the translation, clicking at the bottom left of `aTargetSubview` returns 0,0.

To hit test the subview, use the `mouse:inRect:` method:

```
NSRect targetBounds = [aTargetSubview bounds];
if ([self mouse: mousePositionInPanel inRect: targetBounds]) {…
```

The code in the conditional runs if the mouse is clicked inside the subview. Clicks outside the subview are ignored.

Use `mouseDragged:` to receive left-button drag events. You can read the mouse position using the same code as shown previously.

CAUTION

`mouseDragged:` events continue to arrive even if the mouse is dragged out of the window frame. Hit testing becomes more critical for drag events; otherwise, you can drag values and objects all over the screen, even when you don't want to.

Use `mouseMoved:` events to capture the mouse cursor position when the left button isn't pressed. `mouseMoved:` tracking is *not* enabled by default. To enable it, add the following code to the view's `awakeFromNib:` method:

```
- (void) awakeFromNib {
[[self window] setInitialFirstResponder: self];
[[self window] makeFirstResponder: self];
[[self window] setAcceptsMouseEvents: YES];
}
```

This tells the root window object to pass mouse moved messages to the view for processing. `mouseMoved:` tracking is active across the entire screen, and you should hit test the window bounds to limit the mouse action.

This is a small selection of the available mouse methods. The full list is defined in the NSResponder and NSView Class References. Other useful methods include:

- `mouseUp:` Sent when the left button is released.
- `mouseEntered:` Sent when the cursor enters a tracking area.
- `mouseExited:` Sent when the cursor exits a tracking area.

Use the `NSTrackingArea` class to define tracking area objects.

Understanding the Cocoa Graphics System

To create Cocoa graphics, you must understand:

- How points, rectangles, and other basic data are packed into data structures, and how to set and read data from them
- How points and other elements are combined to create shape objects
- How shape objects are drawn into a view
- How colors and gradients are created and managed
- How images are loaded and drawn into a view

The rest of this section introduces these basic techniques and concepts.

Understanding and defining basic geometry

In Cocoa, geometric data is formal and structured. There's no equivalent of `plot (x,y)`, `line (x1, y1, x2, y2)` or other simple graphics primitives found in simpler languages. Point and rectangle — "rect" — values are wrapped in data structures, and the structures are used in methods and functions. All components are floats. Cocoa coordinates use a bottom-left origin. Table 16.2 introduces Cocoa's key graphics data structures. Note that these are *not* classes or objects — they are standard C structs or object properties.

Table 16.2 Useful data structures

Data structure	*Description*
`NSPoint`	An x,y coordinate pair. Use `point.x` and `point.y` to access the individual coordinates.
`NSSize`	Defines a rectangle's size. Use `size.width` and `size.height` to access the components.
`NSRect`	A rectangle with an `NSPoint` origin and an `NSSize`. Use `rect.origin.x/y` to access the origin, and use `rect.size.width/height` to access the size components.
`Frame`	An `NSRect` with the origin defined with respect to the object's superview. Changing `origin` moves the object's bottom-left anchor in the superview.
`Bounds`	An `NSRect` with the origin set to 0,0. Use `bounds` to access a subview's internal coordinates. `frame` and `bounds` rects usually have the same size. Changing the `frame` stretches the content.

NOTE

There is no `NSLine` structure, because there is no direct way to draw lines in Cocoa. Lines are defined using the more complex `NSBezierPath` object, which can build arbitrary shapes from line segments, arcs, and complex curves.

Using geometric data structures

A standard problem is getting the `NSRect` of a view. To return the size, use

```
NSRect boundsRect = view.bounds;
```

You can then access the components of `boundsRect` to find the width and height.

You can also access components directly.

```
float thisWidth = view.bounds.size.width;
```

To return the position in the superview with respect to a rect's lower-left point, use

```
NSPoint thisOrigin = view.frame.origin;
```

Note that because `NSPoint`, `NSSize`, and `NSRect` are data structures and not objects, no pointer star is required.

Using Foundation Constants

A number of constants are predefined:

- `NSZeroPoint` Equal to 0,0
- `NSZeroSize` Width and height are both zero
- `NSZeroRect` A rect of zero size at 0, 0

You can use these wherever you need the equivalent `NSPoint`, `NSSize`, or `NSRect`.

Using Foundation Functions

A collection of helper functions is included in Cocoa's Foundation Functions. For a complete list, see the Foundation Functions Reference. You *must* review this reference before you start working with Cocoa graphics. These functions are essential timesavers. It can be helpful to print them out and pin them somewhere close and visible.

Table 16.3 lists a selection of the more commonly used functions. Equivalent functions for `NSSize` are listed in the Foundation Functions Reference.

Table 16.3 Selected Foundation Functions

Function	*Description*
`NSMakePoint`	Packs two floats into an `NSPoint`
`NSMakeRect`	Packs four floats into an `NSRect`
`NSEqualPoints`	Returns `YES` if two points are identical
`NSEqualRect`	Returns `YES` if two rects are identical
`NSIntersectionRect`	Returns the rect where two input rects overlap
`NSUnionRect`	Returns the smallest rect that encloses two input rects
`NSMidX` and `NSMidY`	Returns the X or Y center of a rect
`NSPointInRect`	Returns `YES` if a point is inside a rect

CAUTION

Because the basic data element is a float, be careful when comparing points, rects, and sizes. Rounding errors may mean that similar values aren't identical, and comparisons may fail because of a difference in the nth decimal place. This can become a problem when you multiply points and rects to scale them. You can use `NSIntegralRect` to round float values up to integers in a rect.

Creating shapes and colors in drawRect:

The Cocoa graphics system takes points, sizes, and rects and assembles them into shapes. Typically, you add code to a view's `drawRect:` method to call functions and methods that define shapes with `NSBezierPath` objects. Optionally, you can manipulate them to stretch them, move them, and rotate them.

Path objects are invisible. You can draw them into the view by *stroking* them — drawing their outlines in a color or line style — or *filling* them, which fills the outline with a solid or variable color. Table 16.4 introduces the key concepts and objects.

Table 16.4 Key Graphics Features

Feature	*Description*
`NSGraphicsContext`	A canvas for graphics. Can be used online for instant display, offline for off-screen rendering, or print-ready for PDF creation and paper print. Optional when drawing into a view.
`NSBezierPath`	A collection of points that define a shape, connected by arbitrary curves. Curvature can be controlled to create straight lines, circles, or arbitrary combined shapes.
`[path stroke];`	Paths are invisible until painted. Stroking a path traces its outline with visible colors and lines.
`[path fill];`	Filling a path fills its area with color.
`NSAffineTransform`	An object that shrinks/stretches, rotates, or moves a path or image. Transforms use matrix math. Helper methods simplify the implementation; matrix expertise isn't essential.
`NSColor`	A color object. Encapsulates color information with respect to a color space. Use the `set` method to set fill and stroke colors.
`NSColorSpace`	Implements color calibration. Adjusts color values to allow for device variations *or* allows device-independent absolute color values.
`NSGradient`	A composite color objects that supports color gradient fills — colors that blend into each other.
`NSImage`	An image object, loaded from a file or generated dynamically.

Creating path objects

You can use `NSBezierPath` in two ways. For simple shapes, including rectangles, rounded rectangles, and ovals or circles, you can use convenience methods to create a path object:

```
NSBezierPath *myRectPath =
[NSBezierPath bezierPathWithRect: aRect];
NSBezierPath *myOvalPath =
[NSBezierPath bezierPathWithOvalInRect: aRect];
NSBezierPath *myRoundRect =
[NSBezierPath bezierPathWithRoundedRect: aRect
xRadius: aFloat yRadius: anotherFloat];
```

The oval path returns an oval or circle that touches the edges of the surrounding rectangle. If the height and width are equal, the path is circular.

For more complex shapes, you can define the path in sections. The number and order of sections are arbitrary, and they do not have to be contiguous. Useful methods include

- `moveToPoint:` Moves the drawing position to an arbitrary point without creating a mark.
- `lineToPoint:` Moves the drawing position to a point, creating a line.
- `closePath:` Moves the drawing position to the start of the path, creating a closed shape.

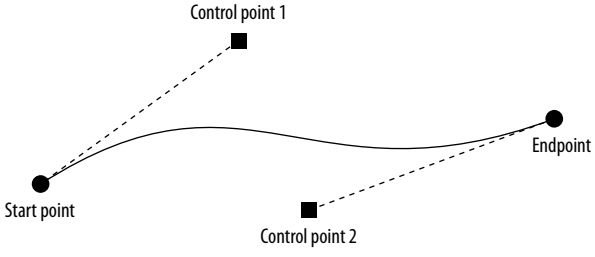
`relativeMoveToPoint:` and `relativeLineToPoint:` convenience methods are available. They take relative rather than absolute coordinates to simplify certain path calculations.

Creating Bezier paths with control points

`curveToPoint:controlPoint1:controlPoint2:` draws a curve from the current position to a point using two control points to define a Bezier curve, as shown in Figure 16.8. The control points are invisible, but they define the shape of the curve. Bezier curves are used to define glyph shapes in fonts, but you can use them in any application that requires arbitrary line shapes.

Figure 16.8

A Bezier curve defined by two control points. The points are invisible "handles" that define the curvature of the path. They can be placed anywhere to create an almost infinite range of curves.



Creating composite Bezier paths

You can join paths to create complex composite paths. Use `appendBezierPath:` to join one path to another. Variations include

- `appendBezierPathWithPoints:` Takes an array and interprets the elements as points joined by line segments.
- `appendBezierPathWithArcFromPoint:` to Point: Creates an arc between two points.
- `appendBezierPathWithGlyph:` InFont: Adds a font glyph from a selected font.

NOTE

You must use the `appendBezierPath:` methods to create arcs. You can only append arcs to a path; you can't initialize or create a path with an arc.

Stroking and filling paths

Once you have a path object, you can draw it by calling the `stroke` and `fill` methods on it. Optionally, you can set a color, line type, and line width for these methods. The default color is black, the default line type is unbroken, and the default width is 1 pixel.

Additionally, you can define line end caps to create arrow heads or lines with curved terminations. You can also specify line mitres to create butt-end or sharpened line joins. See the NSNBezierPath Class Reference for details. Color is global, but line features are properties of a path object.

As a very simple example, you'll create a rectangle offset from the edges of a container view. Rect calculations can become complex because the bottom-left origin complicates size and position calculation. It's often useful to center objects, but this is only possible by calculating the size and position manually.

Start by creating a new blank project. Create a new `NSView` subclass called **RectsView.** Open IB and assign the subclass to the container view. Add the following code to the `drawRect:` method in the new view:

```
- (void)drawRect:(NSRect)dirtyRect {
float rectOffset = 30;
 NSRect boundsRect = self.bounds;
 NSRect newRect = NSMakeRect(boundsRect.origin.x+rectOffset,
                  boundsRect.origin.y+rectOffset,
                  boundsRect.size.width-2*rectOffset,
                  boundsRect.size.height-2*rectOffset);
 NSBezierPath *newPath =
  [NSBezierPath bezierPathWithRect:newRect];
  [newPath stroke];  //Draw the path
}
```

Save the file and nib. Build and Run the project. Figure 16.9 shows the result.

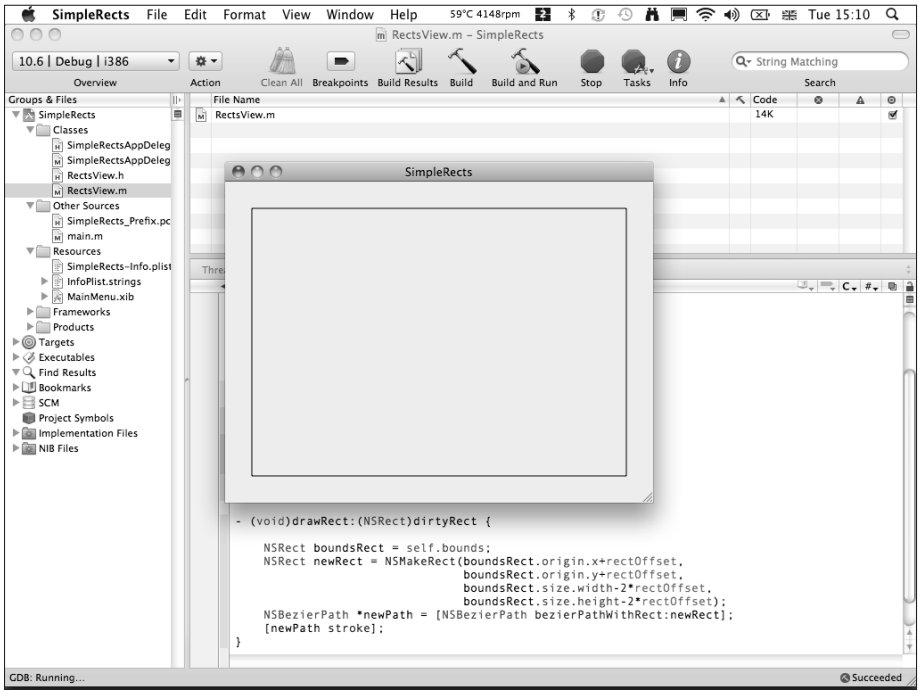
Change `[newPath stroke]` to `[newPath fill]`. Build and Run the project. You'll see that the rectangle is now filled with solid black, as shown in Figure 16.10.

Using colors

In Cocoa, `NSColor` defines a color object. Call the set method on the color object to set fill and stroke colors. Convenience methods are available for various predefined colors. See the NSColor Class Reference for a list.

Figure 16.9

Drawing a centered rect inside a view, with the edges offset by a constant. Try resizing the window. The `drawRect:` method is called automatically, and the rectangle follows the window dimensions.



For example, to use the predefined blue color to fill or stroke a path, use

```
[[NSColor blueColor] set];
```

To define a color with specific components, use

```
[[NSColor colorWithDeviceRed: rFloat green:
 gFloat blue: bFloat alpha: aFloat] set];
```

or

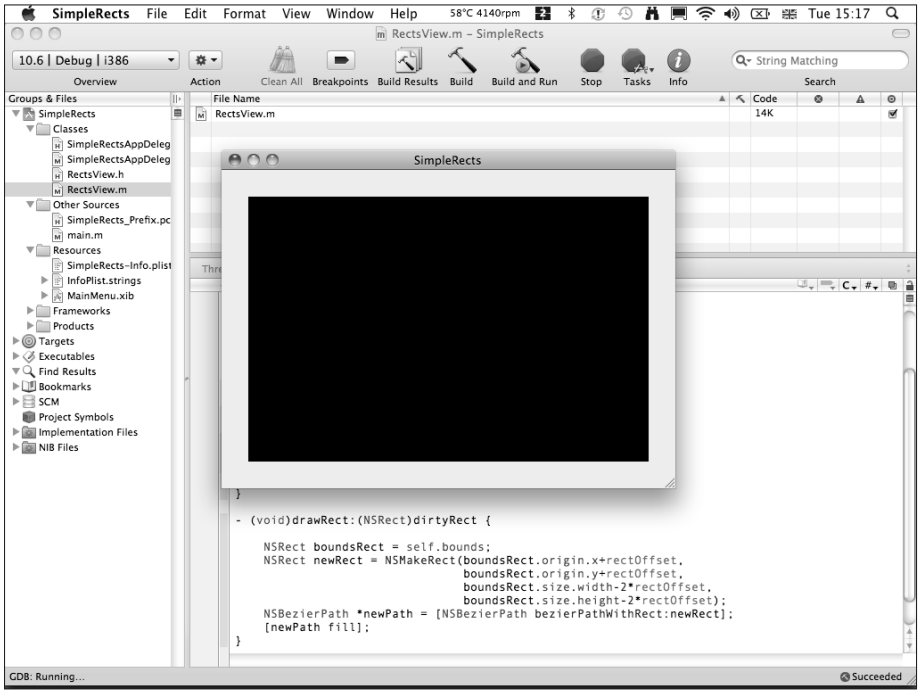
```
[[NSColor colorWithDeviceHue: hFloat saturation: bFloat
brightness: brFloat alpha: aFloat] set];
```

Each component is a float in the range 0 to 1. The `alpha` parameter sets transparency. 0 is invisible, 1 is solid.

The color is written to the graphics context and applied globally, until `set` is called again. Paths don't have a separate color property.

Figure 16.10

Replacing `stroke` with `fill` fills the rectangle with a solid color.



Using calibrated colors

Color is an unexpectedly complex topic. Cocoa color is designed to support *calibration* — a mapping process that eliminates imperfections in color hardware. Monitors and printers often have poor calibration, so there is no guarantee that a given shade of blue or red will appear the same on two different devices.

A full discussion of calibration is outside the scope of this book, but you can use the `colorWithCalibratedHue:` and `colorWithCalibratedRed:green:blue:alpha:` methods to create calibrated color objects. These methods use the calibration features built into OS X to create colors that take into account existing calibration maps.

Creating and drawing gradients

Gradients are *not* defined and used in the same way as color objects. An `NSGradient` object takes two or more colors and defines a smooth blend between them, as shown in Figure 16.11.

```
NSGradient *thisGradient = [[NSGradient alloc]
initWithStartingColor: aColor
endingColor: anotherColor];
```

Gradients are then drawn separately, either filling a rect with

```
[thisGradient drawInRect: aRect withAngle: angleInDegrees];
```

or filing a path with

```
[thisGradient drawInBezierPath: aPath angle: angleInDegrees];
```

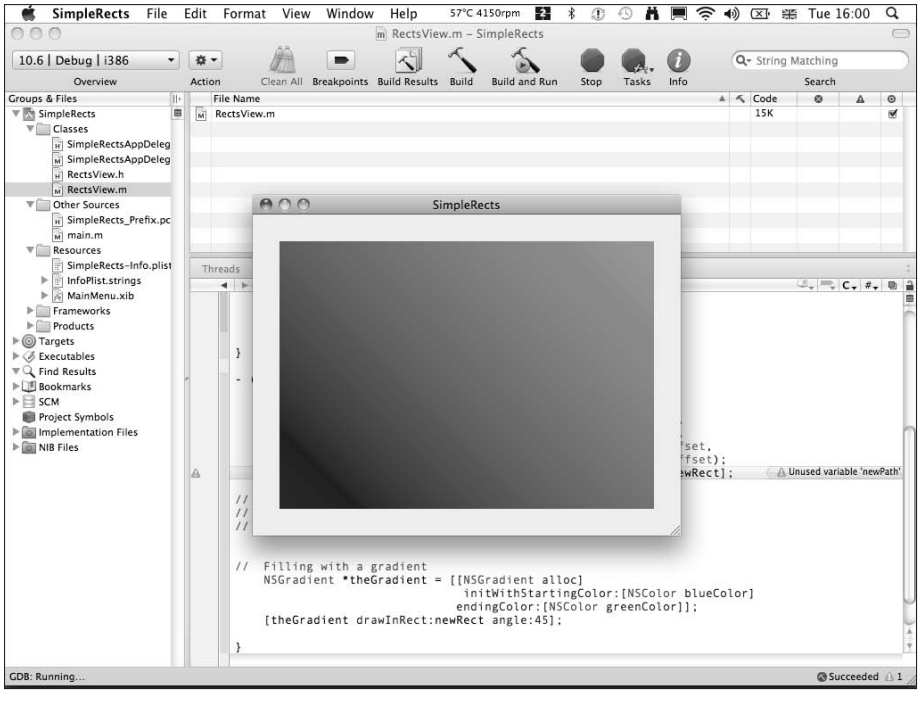
The `angle` parameter rotates the gradient. There is no `set` method for a gradient.

CAUTION

Most graphics methods and functions take parameters in radians. The gradient drawing functions take an angle parameter in degrees.

Figure 16.11

Filling a rect with a gradient. The angle parameter rotates the gradient. Note that the rotation is around the bottom-left corner of the rect.



Transforming paths

Unless you're a math expert, the matrix calculations used in affine transforms aren't trivial. Fortunately it's easy to use affine transforms without understanding how they do what they do. To transform a path, create a transform object:

```
NSAffineTransform *aTransform = [NSAffineTransform transform];
```

This slightly unusual syntax creates a default null transform that does nothing when it's applied to an object. You can then use scale methods to create a transform that resizes a path, rotate methods to rotate a path, and translate methods to move a path.

For example

```
[aTransform rotateByDegrees: 45];
```

creates a transform that rotates an object by 45 degrees. Similarly,

```
[aTransform translateXBy: xFloat yBy: yFloat];
```

creates a transform that moves an object; and

```
[aTransform scaledXBy: xFloat yBy: yFloat];
```

creates a transform that scales an object.

Transforms are cumulative, so you can scale, rotate, and then translate. The order is important: scaling and rotation happen around the bottom-left origin. Keeping track of the relative origin can be challenging. Typically you translate objects to the origin to scale and rotate them, and then translate them back to their original position, or to some other point in the view. A complex path may not have a well-defined center.

To apply the transform to a path, use

```
NSBezierPath *newPath =
[aTransform transformBezierPath: anOldPath];
```

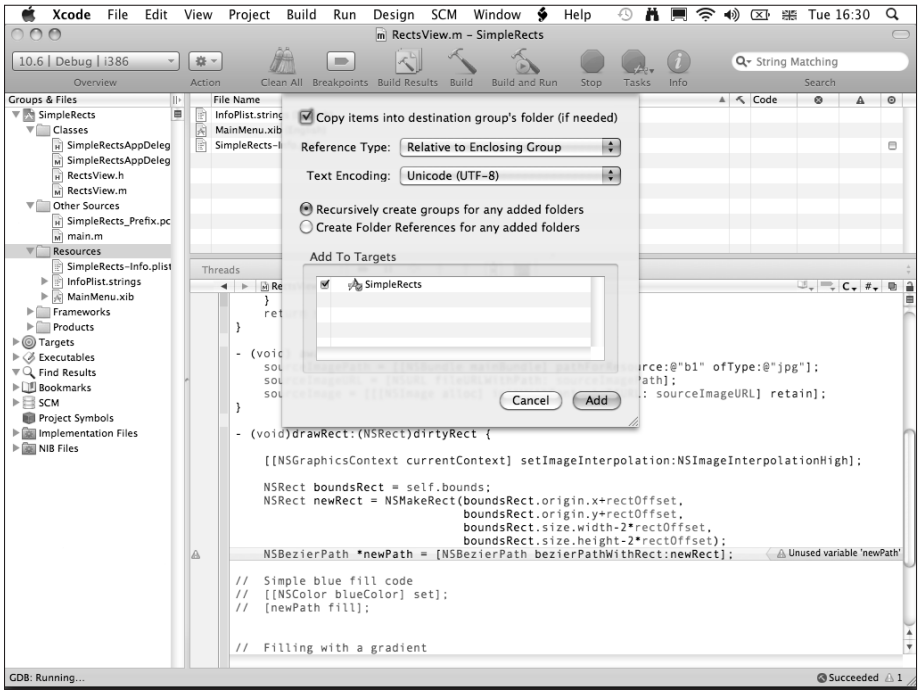
When you stroke or fill `newPath`, it appears in its translated location, scaled and rotated.

Loading images

To load an image, create an `NSImage` object and initialize it with image data from a file. You can specify an arbitrary filepath, or you can include the image in the application bundle and load it directly from the bundle. Use the former approach for image editing, and the latter for defining images that are loaded and displayed automatically. For example, to include an image in the bundle, right-click the Resources group and choose Add➪Existing Files. Navigate to the file, select it, and copy it into the project, selecting the Relative to Enclosing Group reference type, as shown in Figure 16.12.

Figure 16.12

Bundling a file with a project. Don't forget to select the Relative to Enclosing Group Reference Type; otherwise, your bundled file won't be bundled correctly.



You can load an image each time it's drawn, but this is usually very inefficient. It's more efficient to load it once in `awakeFromNib` and reference it as needed. For example:

```
NSImage *sourceImage;
- (void) awakeFromNib {
 NSString *sourceImagePath =
 [[NSBundle mainBundle] pathForResource:@"b1" ofType:@"jpg"];

 NSURL *sourceImageURL =
 [NSURL fileURLWithPath: sourceImagePath];
 sourceImage =
 [[[NSImage alloc] initWithContentsOfURL: sourceImageURL]
   retain];
}
```

This creates a URL to a named file from a path that accesses the application bundle. The `initWithContentsOfURL:` method loads the image data into an instance of `NSImage`.

NOTE

You can access image files elsewhere on disk via a conventional string filepath. Files in the application bundle must be accessed via a URL. This isn't very consistent, but it's the law.

Drawing images

You can draw an image object at any point in a view. Technically the image isn't drawn; it's *composited,* combining the image data with the existing pixel values in the view.

If you've used an image-editing package, you'll already be familiar with compositing. It's sometimes known as *layer blending*. Cocoa supports various compositing options. You can composite the image as is, you can extract maximum and minimum pixel values after composition, or you can XOR the image to create excessively garish psychedelic effects. Opacity — transparency —- is controllable with a separate parameter, also known as *alpha*.

To draw an image, call the `drawInRect:` method on it. The full method signature specifies a destination rect, a source rect, an *operation* parameter that takes one of Cocoa's predefined compositing modes, and a *fraction* parameter between 0 and 1 that defines opacity/alpha.

TIP

Specifying `NSZeroRect` for the source rect is a quick way to specify the entire image. For example:

```
[sourceImage drawInRect:newRect fromRect:NSZeroRect
   operation:NSCompositeSourceOver fraction:0.5];
```

The compositing mode constants are defined in the NSImage Class Reference.

Figure 16.13 shows the result of combining an image with an underlying gradient and drawing it with an opacity of 50 percent.

TIP

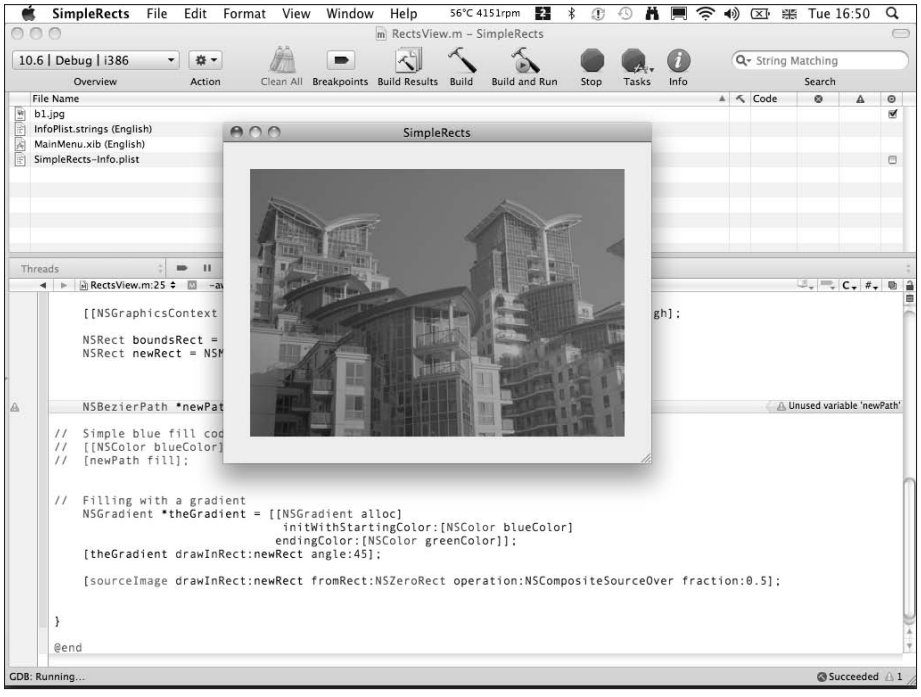
When an image is resized, it's automatically interpolated — that is, the original pixel information is expanded or made smaller to fit the new size. You can set the interpolation quality by including

```
[[NSGraphicsContext currentContext]
setImageInterpolation:NSInterpolationHigh];
```

at the start of `drawRect:`.

Figure 16.13

Loading and compositing an image. Selecting the `newRect` rect as a destination automatically resizes the image as the window is resized. Setting the opacity to 50 percent allows some of the underlying gradient to show through.



Combining Quartz, Core Graphics, and Cocoa graphics

The Cocoa graphics system is a wrapper for an underlying C-based framework called Quartz 2D. The functions and data structures used in Cocoa have Quartz equivalents, prefixed with `CG` instead of `NS`. Operationally, the frameworks are recognizably similar. Quartz uses paths, colors, points, gradients, and rects in almost identical ways, with the difference being that Quartz features are accessed via function calls rather than methods. Quartz provides lower-level support for graphics, and also makes it possible to draw images off-screen and to repeat them as textures. It may also be significantly faster than Cocoa drawing code, although this can be very application dependent.

Quartz is used on the iPhone because Cocoa graphics aren't available. For more details, see Chapter 19.

It's possible to combine Quartz and Cocoa features and to interconvert data between them. For example, you can convert an `NSPoint` to and from the equivalent `CGPoint` with `NSPointToCGPoint` and `NSPointFromCGPoint`.

For more information about Quartz 2D, see the Quartz 2D Programming Guide in the Documentation.

Creating a simple project: MultiBezier

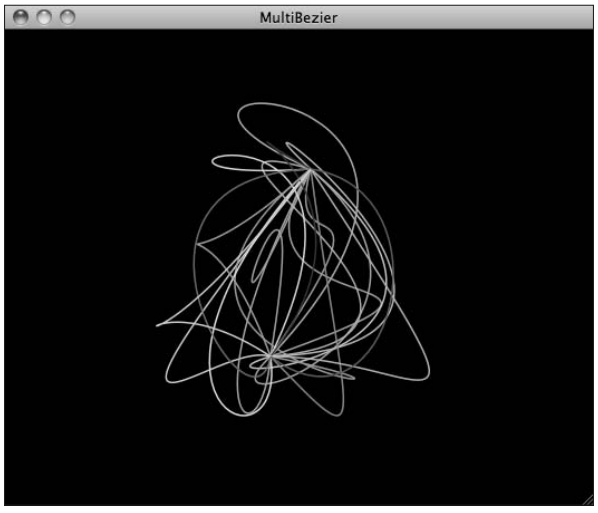
A simple supporting project that combines all the key features of views and graphics, including colors, Bezier paths, mouse events, and `draw-Rect:`, is available on the Web site (`www.wiley.com/go/cocoadevref`). The `drawRect:` method is listed here. (The Web site version includes optional code that demonstrates transforms.)

```
- (void)drawRect:(NSRect)dirtyRect {
 NSGraphicsContext *thisContext =
 [NSGraphicsContext currentContext];
 for (int i=0; i < maxLines; i++) {
 float thisHue = (float)0.001*(arc4random() % 1000);
 [[NSColor colorWithDeviceHue:
 0.1+thisHue*0.55 saturation:1.0
 brightness:1.0 alpha:1.0] set];
 c1Point = NSMakePoint(arc4random() %
   (int)self.bounds.size.width, arc4random() %
   (int)self.bounds.size.height);
 c2Point = NSMakePoint(arc4random() %
   (int)self.bounds.size.width, arc4random() %
   (int)self.bounds.size.height);
 NSBezierPath *newPath = [NSBezierPath bezierPath];
[newPath moveToPoint:startPoint];
[newPath curveToPoint:endPoint
 controlPoint1: c1Point controlPoint2: c2Point];
[newPath setLineWidth:1.5];
[newPath stroke];
}
```

This code creates `maxLine` Bezier paths with random control points and random colors, joining two points defined with mouse clicks. Clicking again creates a new set of curves. The result is shown in Figure 16.14.

Figure 16.14

A simple application that creates random multiple Bezier paths between two points, in a range of attractive colors



NOTE

The window background has been set to black in the application delegate.

Using CoreImage Filters

CoreImage filters are a collection of prewritten image filtering plug-ins that you can apply to the content or background of any view. The filters are listed by name in the Core Image Filter Reference in the Documentation, which also includes preview images. If you have Adobe Photoshop experience, many of the effects will be familiar, but the full list of filters includes novel effects that haven't — yet — been made available in industry-standard image editors.

There are three ways to use filters:

- You can add them to any view by clicking the view's Effects tab in the Inspector window in Interface Builder.
- You can trigger them from code in a subclassed implementation of `drawRect:`.
- You can write Objective-C code that combines existing filters or defines completely new effects.

Creating and combining filters are advanced topics and are outside the scope of this book. They won't be covered here. The first option is very accessible and you can use it to add special effects to any UI with very little effort. The second is more challenging, but can be implemented with relatively simple boilerplate code.

NOTE

CoreImage effects are processor-intensive and aren't yet available on the iPhone or iPad.

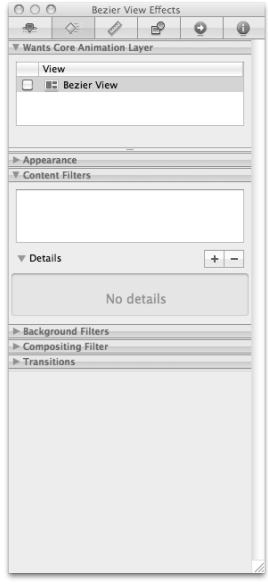
Adding CoreImage effects in Interface Builder

As a demonstration of view-based CoreImage effects, you'll extend the MultiBezier project to include them. Adding effects is a trivial process.

Select the BezierView in the Doc window. If the Inspector window isn't visible, open it by choosing Tools➪Inspector. Select the Effects tab, which is second from the left. Find the Content Filters pane, as shown in Figure 16.15.

Figure 16.15

Finding the Content Filters pane, which applies one or more filters to the contents of a view



Click the + button. You'll see the pane shown in Figure 16.16. Select Blur and then Gaussian Blur. Click OK to add the filter to the view.

Figure 16.16

Selecting and adding a filter from the list. The filters are grouped by function. To see a complete list, select the All group.

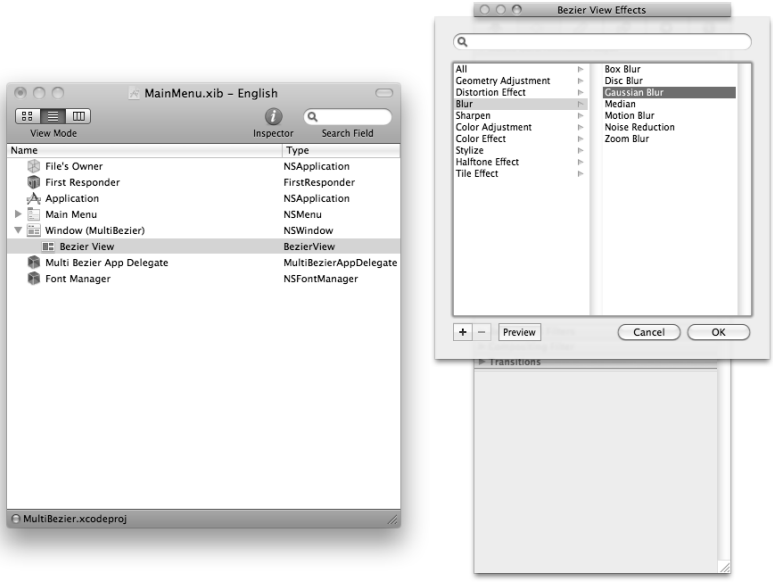


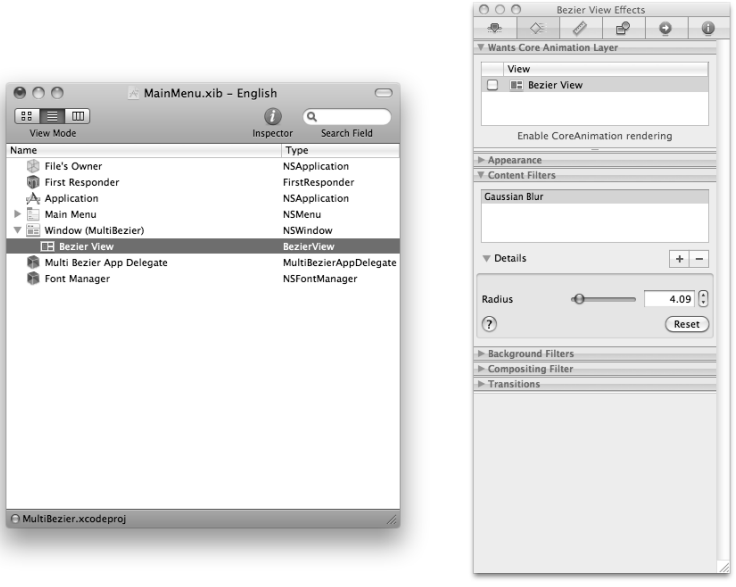
Figure 16.17 shows the result. Select the check box next to Bezier View to enable the effects. In the Details tab, you can change the filter's settings. Change the Radius to a value around 4. The slider isn't calibrated to precise values, but any value between 3 and 5 is acceptable.

NOTE

Technically CoreImage is part of the CoreAnimation system, which is why the prompt at the top of the window asks you to Enable CoreAnimation rendering. Click the check box next to the top view to enable CoreAnimation.

Figure 16.17

Setting the filter details — the specific values that control each filter property. The Gaussian Blur filter has a single Radius property. Some filters have no properties at all. Others have six or more.



Save the nib file. Build and run the application. Figure 16.18 shows how the content is processed through the blur filter. You can now experiment with the other filters, or you can create combination effects by adding multiple filters. Use the – button to remove a filter. You can change the order of the filter effects by dragging and dropping them.

Follow analogous steps to set up filter effects for a view's background. Aqua's menu system uses this feature to blur the content behind a menu, creating a floating translucent effect.

Setting up filters for processing

The Twirl Filter project supplied with this book demonstrates how to apply a filter to the contents of a context. In this example, the code loads an image from the application's bundle, processes it, and draws it into a view. The result is the same as applying a filter to the view in IB. But you can use a similar technique to apply filters to an off-screen context that isn't visible; for example, to implement batch processing of images.

Figure 16.18

With the filter running, the sharp lines of the Bezier paths are replaced with wispy blurs. The Radius parameter controls the amount of blur. This is the output of a single filter, but you can combine filters almost indefinitely to create sophisticated multiple effects.

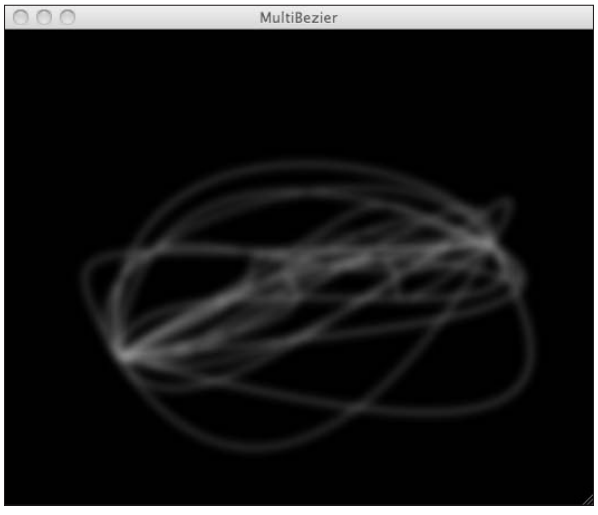


Figure 16.19 shows the application's nib and UI. The Filter Browser View and the Filter View are both subclasses of NSView. The Browser View manages the interface. It responds to `mouseDown:` events within the application window and implements two sliders that are linked to two filter parameters. This view exists to manage the UI, and its `drawRect:` method isn't used.

The Filter View receives its settings from the UI, and implements the filter. A filter object is created with a call to `CIFilter` using the `filterWithName:` method:

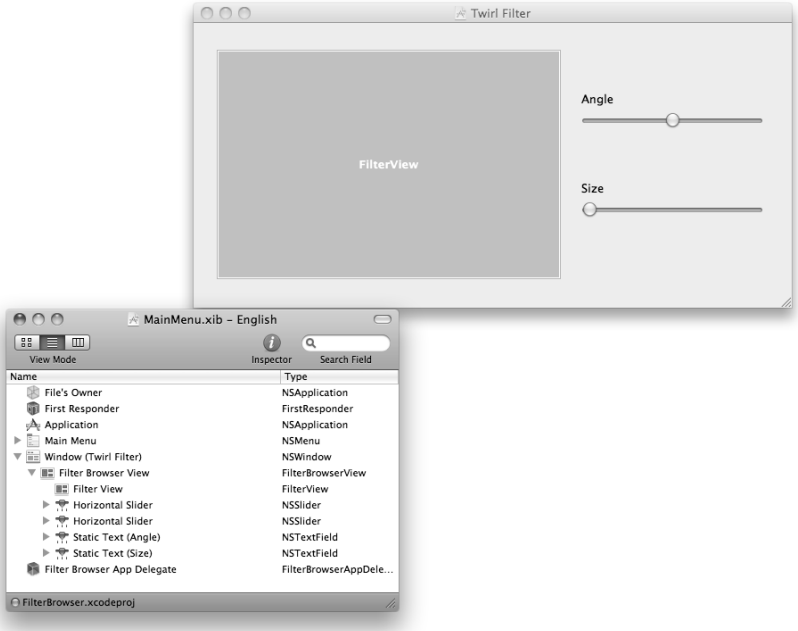
```
CIFilter *thisFilter =
[CIFilter filterWithName: @"CITwirlDistortion"];
```

`CIFilter` parameters are accessed through key value coding:

```
[thisFilter setValue: [NSNumber numberWithFloat: 50]
forKey: @"inputRadius"];
```

Figure 16.19

The nib and view structure of the Twirl Filter application. It's a standard content view with an extra filter view and a pair of sliders.



Finding filter keys

Each filter has a different set of keys listed in the Core Image Filter Reference Guide. In the Twirl Filter project, the keys are hardwired into the code and are controlled from another view. You can also read the keys for any filter in the code with the `inputKeys` method. It generates an array of key names as strings:

```
NSArray *keyNames = [thisFilter inputKeys];
```

This list isn't enough to create a useful UI, because it doesn't tell you the maximum or minimum range of each key value. To retrieve that information, use the `attributes` method to return a dictionary:

```
NSDictionary *filterAttributes = [thisFilter attributes];
```

The dictionary lists:

- The categories the filter belongs to. Categories are functional groups of filters; for example, distortion, video, still image, and so on. Category names are defined in the Reference Guide.
- The filter name.
- A list of key names.
- A list of attributes for each key, including the maximum and minimum acceptable values and the value type.

Creating filter controller interfaces

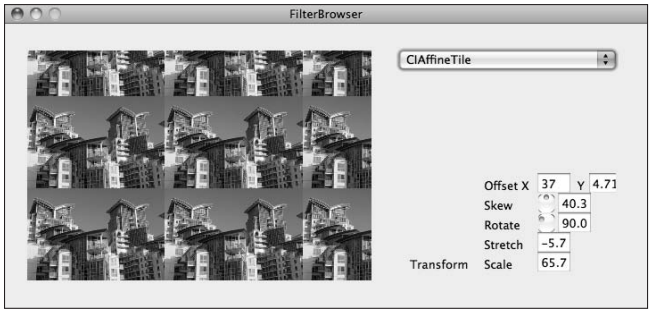
In theory, you can use this information to create a completely general filter wrapper. In practice, the dictionary format is complex, and creating a general automated UI is a challenging project.

A simpler solution is available. Running the `viewForUIConfiguration:` method on any filter generates a complete controller view automatically, with the appropriate sliders, color pickers, and other UI features. You can specify various sizes of controller view, and you can also exclude specific parameters; for example, you may choose to implement an image source picker separately. Slot the controller view into a container view with `addSubview:` or `replaceSubview:`.

Figure 16.20 shows a simple prototype example. The UI controls appear at the bottom right. The UI design is more functional than stylish, and you must use bindings to link the UI elements to filter parameters. But you can use this method to auto-generate filter controls with relatively little coding effort.

Figure 16.20

Using `viewForUIConfiguration:` to create a filter interface.
The auto-generated view appears at the bottom right of the window.



Applying filters to an image

The `drawRect:` method in the Twirl Filter example demonstrates how to set the filter's input and output values and how to apply the filter. The input image is set with the `inputImage` key.

The image object must be a `CIImage` — a special image type unique to CoreImage. In this example, the image data is loaded from a file via a URL path.

For a more general solution, convert the source image into a bitmap or `NSData` object and load the `CIImage` object with the `imageWithBitmap:` or `imageWithData:` method.

The simplest way to process data from a view — without adding the filter to the view in IB — is to read it from a `CGLayer` object.

These conversion options can be messy in practice. In some applications, you will need to convert images through various intermediate data types. But they make it possible to filter images generated by the other graphics frameworks.

The `drawRect:` method that runs the filter follows:

```
- (void)drawRect:(NSRect)viewRect {
    thisContext = [[NSGraphicsContext currentContext] CIContext];
    preFilterImage =
    [CIImage imageWithContentsOfURL:sourceImageURL];
    [thisFilter setValue: preFilterImage forKey: @"inputImage"];
    thisCenter =
    [CIVector vectorWithX: filterCenter.x Y: filterCenter.y];
    [thisFilter setValue:thisCenter forKey:@"inputCenter"];
    [thisFilter setValue:
    [NSNumber numberWithFloat: filterSize]
   forKey:@"inputRadius"];
    [thisFilter setValue:
    [NSNumber numberWithFloat: filterAngle]
   forKey:@"inputAngle"];
    postFilterImage = [thisFilter valueForKey:@"outputImage"];
    CGRect preRect = [preFilterImage extent];
    CGPoint anImageOrigin =
    CGPointMake(viewRect.size.width*.5 - preRect.size.width*.5,
    viewRect.size.height*.5 - preRect.size.height*.5);
    [thisContext drawImage: postFilterImage
                   atPoint:anImageOrigin
                  fromRect:preRect];
}
```

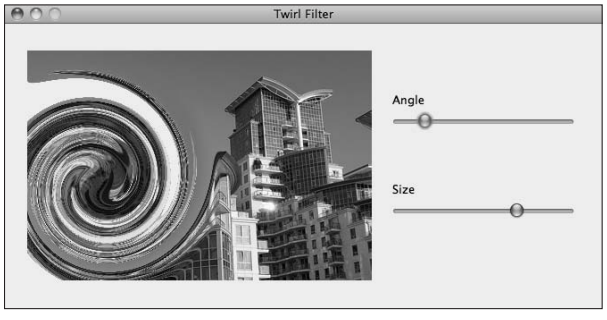
Key points of the code include:

- The `filterAngle`, `filterCenter`, and `filterSize` values are supplied from the controller view.
- Filter output is managed by a special `CIContext` object. This works like `CGContext` or `NSContext`, but includes a special `drawImage:` method that draws the output into the context and, from there, into the surrounding view.
- A `CIVector` data type is used to define a point. Filters cannot use `NSPoint` or `CGPoint`.
- The input and output image data is stored in two `CIImage` objects, called `preFilterImage` and `postFilterImage`.
- *The filter is run by triggering the* `drawImage:` *method in the last line.* No filtering is done until this method is called. This method runs the filter and then writes the data into the context at the specified point, which, with almost complete inconsistency, is specified as a `CGPoint` and not as a `CIVector`.

The application is shown in Figure 16.21. Clicking in the filter view sets the filter center, and the sliders control the filter parameters. You can use similar code to implement any other CoreImage filter, setting different keys and values and adding or removing parameter sliders from the controller as needed.

Figure 16.21

The finished Twirl Filter. Creating this simple filter effect isn't an entirely trivial challenge.



CAUTION

CoreImage isn't robust. If a filter crashes because you supply it with invalid values or keys, the entire framework stops working and all filter effects stop with it. The only way to restart it is to reboot your application. You *must* debug code carefully to avoid crashing CoreImage.

Summary

In this chapter you were introduced to the view hierarchy and discovered how to add and remove objects dynamically. You explored the data structures used in the Cocoa graphics frameworks and learned how to define, stroke, and fill paths. You experimented with colors and gradients and discovered how to draw images in a view. Finally, you were introduced to the key points of the CoreImage framework and learned how to add CoreImage to views and to custom drawing code.

Creating Animations and 3D Graphics

Animations are often associated with games, but Cocoa makes it easy to animate almost any element in a user interface. Objects can move smoothly, pulse with color or brightness, jiggle, or change their size with one click or mouse over. Aqua includes standard animated effects, such as slide-out panes and pulsing buttons. In part, these are automatic. When you create a file open/save pane, the animation is embedded in the class and runs without further code.

You can add further animations to improve the look and feel of your application, make it more professional, and add value by subtly highlighting important features.

It's also possible to create animated transitions when replacing one object with another. View swapping of this type is a key skill in iPhone app design, where swapped views replace the windows, panels, and panes of a desktop application.

CROSS-REF

For a practical example of view swapping, see Chapter 19.

In This Chapter

Creating direct property animations

Using animators

Creating custom animation objects

Creating a simple OpenGL animation in Cocoa

To use animations effectively, you must become familiar with Cocoa's different animation classes and features. Many applications underutilize animations or add them in an unimaginative way. A key design goal is to use animations to create a genuine user benefit. A successful animation should do more than look impressive; it should also offer visual hints to the user to help him or her understand the application and use it in a more intuitive way.

For example, on the iPhone, views are swapped by finger movements accompanied by matching animations. The animation suggests a metaphor, such as a page curl, pop-up, swipe, or twirl, that implies something about the organization of the app. An app that swipes into a view but returns with a page curl feels inconsistent.

Cocoa includes a selection of classes and techniques that support animation. In outline, there are three options: direct property access; *Core Animation,* which is Cocoa's main animation framework; and *OpenGL*, which is a complex low-level language that supports 3D scene rendering.

A key feature of OS X animation is that you have a lot of choices about how you use it. There are easy ways to add limited predefined Core Animation effects with hardly any code, and more complex options that offer full control over customized animations assembled from groups of animation objects. There's a natural progression from simple to complex, and part of the challenge of animation is in implementing the simplest possible solution.

This chapter introduces a selection of some of the animation features and techniques built into Cocoa. It doesn't demonstrate every feature or every code interface, but it does summarize the essential features you can use to create effective animations.

TIP

Apple's documentation and other sources of information about Core Animation sometimes imply that Cocoa animation always requires customized animation objects, complicated keypaths, animatable layers, and delegate methods. This is misleading and untrue. Sometimes, you may need these features to build complex animations, but you can create useful and impressive Core Animation effects without them.

Table 17.1 outlines the animation options in Cocoa.

Table 17.1 Available Animation Types

Animation type	*Description*
Direct property animation	Can animate any settable property, but is limited to simple timed animations managed with explicit code.
Default `animator` animation	Creates simple smooth automated transitions between property values. Can be implemented with barely any code, but offers very limited control.
Customized `animator` animation	Can set the duration, timing, and keyframed path of any property animation.
Customized Core Animation effect	A more advanced option that supports abstracted and linked animations. Animation delegation, which triggers messages when an animation begins and when it ends, is also available.
OpenGL	A 3D scene-rendering engine, with hardware acceleration. Often used in games; occasionally used to create 3D interfaces. OpenGL is a challenging and complex environment with a steep learning curve.

Using Direct Property Animation

Direct property animation is crude, simple, and often underappreciated. You create animation effects by initializing an animation timer and recalculating properties and values on each timer tick. In Cocoa, this means placing the animation code inside a timer method.

Core Animation relies on Key-Value Coding (KVC). If a parameter or property doesn't support KVC, Core Animation can't work with it. Direct property animation is unrestricted. You can use it to animate almost any feature of an application.

Direct property animation is also slow and inefficient. This makes it ideal for simple and unusual animation effects that may not be possible in Core Animation. But re-creating some of the advanced animation built into Core Animation "for free" is a challenge and may not be worth the development effort.

Creating a timer for animation

An animation timer is simply a timer that runs a dedicated animation method. There are no special animation timers in Cocoa. To create a timer, use the standard `NSTimer` object. The target is often `self`, but can be a different object.

```
NSTimer *theTimer = [NSTimer scheduledTimerWithTimeInterval:
   float
                      target: anObject
                    selector: @selector(theAnimationMethod:)
                    userInfo: anOptionalObject
                     repeats: YES];
```

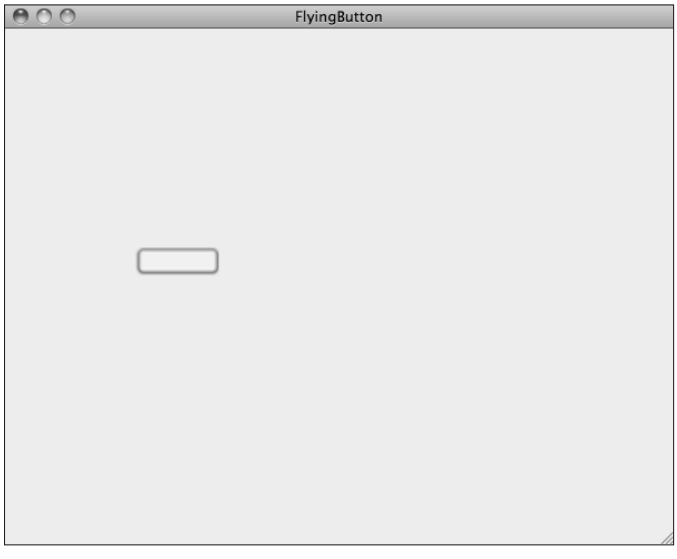
Implement the animation timer method in the target object:

```
-(void) theAnimationMethod: (NSTimer *) theTimer {
    //Do something to a property here
}
```

Figure 17.1 illustrates a simple example. The view includes a single button. The code moves the button in a stately circle around the center of the view and modifies its size dynamically. The button is always active: if you can catch it, you can click it. It could be connected to a selector action in the usual way.

Figure 17.1

The flying button, an unusual UI design you're unlikely to see in a finished application



Creating property animation code

With direct property animation, you have complete freedom over how you implement the animation features. The key section of code for this example follows:

```
@synthesize window, theButton;
float bOffX, bOffY, currX, currY, centerX, centerY, timeCount,
   radius, theta;
NSPoint currPoint;
- (void)applicationDidFinishLaunching:
(NSNotification *)aNotification {
   NSTimer *aTimer =
   [NSTimer scheduledTimerWithTimeInterval:0.017
      target: self selector: @selector(timerMethod)
    userInfo: nil repeats: YES];
}
-(void) timerMethod {
   timeCount +=0.1;
   theta = timeCount/6.283;
   bOffX = NSMidX(theButton.bounds);
   bOffY = NSMidY(theButton.bounds);
   centerX = window.frame.size.width/2;
   centerY = window.frame.size.height/2;
   radius = centerY*0.6;
   currX = centerX+radius*cos(theta)-bOffX;
   currY = centerY+radius*sin(theta)-bOffY;
   currPoint = NSMakePoint(currX, currY);
   [theButton setFrameOrigin:currPoint];
   [theButton setFrameSize:
   NSMakeSize(50+25*cos(2*theta), 50+25*cos(3*theta))];
}
```

The timer method updates the position counter, calculates the center of the view, and then uses simple trigonometry to create a circular path for the button. The button is repositioned by updating its frame origin with `setFrameOrigin:`, and it is resized by updating the frame size with `setFrameSize:`. An offset term corrects for the lower-left origin of the button's frame and moves the draw point to the center of the button.

This is moderately complex code for a simple effect. It's also heavily load dependent. If your Mac is busy, the button will stutter. This approach also makes it difficult to manage multiple simultaneous animations. It's a good solution when you need to animate a small number of related properties, but it's difficult to work with if you have more than one independent animation happening at once. You can create a separate timer for each animation, but this approach soon starts to become unwieldy.

TIP

When animating position and size, set an object's `frame` property, not its `bounds`. This also applies to rotation. A full rotation solution requires a backing layer, which is described later in this chapter.

Using drawRect:

You can use a similar technique to create customized animated 2D graphics, using the paths and drawing objects described in the previous chapter. Use the timer method to initialize relevant path and object values for each update, and end the timer method with

```
[theView setNeedsDisplay];
```

NOTE

If the view is refreshing itself, replace `theView` with `self`.

This generates an indirect call to the view's `drawRect:` method and refreshes the view. The `drawRect:` code can read the updated values to create a refreshed view for each timer update. You can use this technique to create complex effects. It's fast enough for simple games, but can stutter when the drawing code is dense and is updated at typical game refresh rates.

For more complex effects, you can get smoother results using one of Cocoa's other animation options.

Using Animators

You can create very simple animation effects by adding a *proxy* — a simple drop-in animation controller that automatically adds an animation object, initializes it with useful default values, and runs the animation. Core Animation's proxy option gives you a lot of animation power for very little code. You can customize the animations with extra code, but a minimal implementation adds almost no code at all.

For example, to animate the position update for a button, replace

```
[theButton setFrameOrigin: newPosition];
```

with

```
[[theButton animator] setFrameOrigin: newPosition];
```

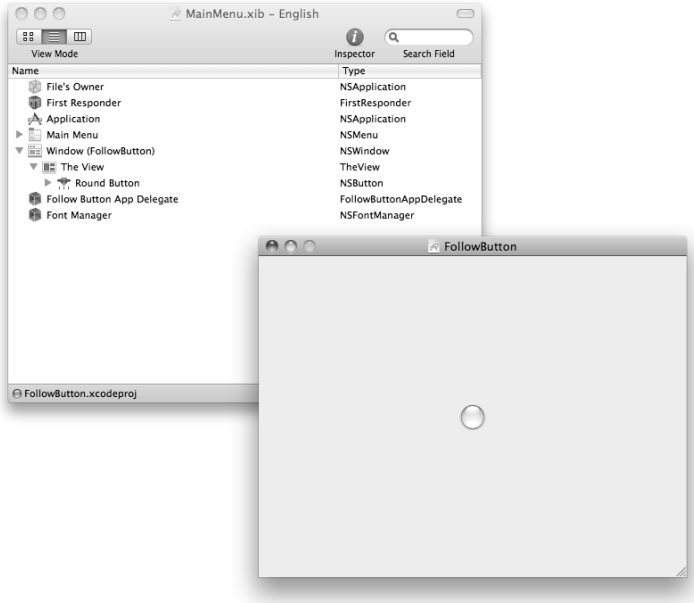
The `animator` creates a smooth change from the current position to the new position, with a default duration of 0.25 seconds.

Creating a simple proxy animation

Figure 17.2 shows the nib from a very simple application that demonstrates proxy animation, using a button as a convenient animatable object.

Figure 17.2

The nib file for a simple animator demonstration application, with a subclassed view containing a button



`TheView` is a subclass of `NSView` and includes the code shown below. When the user clicks the mouse, the `mouseDown:` handler reads the position and moves the button to the click position. Without an `animator`, the position of the button changes instantly. With an `animator`, the button glides between the old and new positions.

The code is straightforward. As in the previous example, an offset term corrects for the lower-left frame origin of the button; otherwise, the code reads the window coordinates of the mouse click, corrects for the offset, and sets the button's position with the `animator`.

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

@implementation TheView
@synthesize theButton;
NSPoint oldPosition;
float bOffX, bOffY;
```

```
-(void) mouseDown: (NSEvent *)theEvent {
   NSPoint mousePosition = [theEvent locationInWindow];
   bOffX = NSMidX(theButton.bounds);
   bOffY = NSMidY(theButton.bounds);
   mousePosition.x = mousePosition.x-bOffX;
   mousePosition.y = mousePosition.y-bOffY;
   [[theButton animator] setFrameOrigin: mousePosition];
}
```

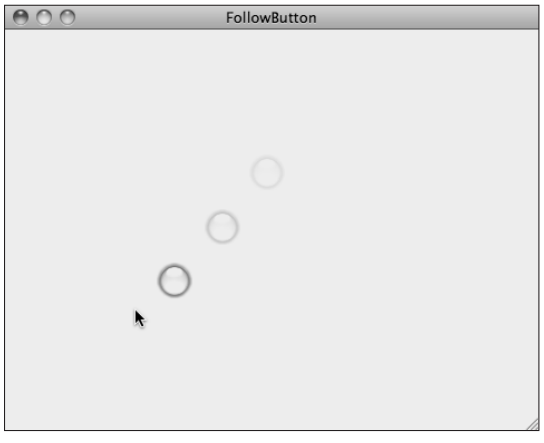
CAUTION

Animation effects require the QuartzCore framework. To add this framework to your project, right-click the Frameworks folder, choose Add ➪ Existing Frameworks, scroll down the list to find the QuartzCore.framework item, select it, and click the Add button.

The animation time is fixed, so the button velocity depends on the distance it has to cover. Figure 17.3 gives a hint of the result.

Figure 17.3

Click the mouse anywhere, and the button follows eventually.



Note that you don't use `animator` as a class or an object. It's a wrapper for a more complex animation object, and you can't access that object's properties directly. It's best to think of `animator` as a unique modifier. There's nothing quite like it anywhere else in Cocoa. Functionally, it means you can create animations without object creation and setup code. You have the option to bypass `animator` and add that code when you need it, but for many effects `animator` is simpler and can be almost as flexible.

Setting the animation duration

To change the default animator duration, wrap it in `NSAnimationContext` calls.

```
[NSAnimationContext beginGrouping];
[[NSAnimationContext currentContext] setDuration 2.0];
[[theButton animator] setFrameOrigin: mousePosition];
[NSAnimationContext endGrouping];
```

Technically, this creates a wrapper object with a modified duration value that is fed through to the underlying animation object. In practice, you can use this code as boilerplate. You can also use it to group multiple `animator` effects together, giving them a common duration and triggering them at the same moment.

CROSS-REF

The second FollowButton project on the Web site for this book (`www.wiley.com/go/cocoadevref`) demonstrates a simple modification that animates both button size and target position.

Customizing the animation object

By default, `animator` works in an opaque way. It contains an animation object, but you can't modify its properties. However, you can create a new custom animation object, set its properties as needed, and plug it into an animator, replacing the default animation. To do this, create an instance of one of Core Animation's animation manager classes.

Core Animation is a powerful but somewhat disorganized framework. Instead of one animation object, there are many subclasses of a class called `CAAnimation`. Table 17.2 summarizes the most useful subclasses and their typical applications.

Table 17.2 Key Core Animation Classes

Class	*Description*
`CAAnimation`	Root class, rarely used directly. Creates animation objects, supports `animationDidStart:` and `animationDidEnd:` delegate methods. Also implements `removedOnCompletion:` to auto-delete animations after they run, and the `timingFunction:` property to control the development of the animation over time.
`CAPropertyAnimation`	`keyPath` property defines the keypath of the animated property, and `cumulative` defines whether the animation adds to or replaces the current value of that property. Rarely used.
`CABasicAnimation`	The most useful object. Implements all the previous properties, and adds `fromValue:` and `toValue:` properties that define the animated value numerically. Includes an optional `byValue:` property that the animation sequence will pass through, if it's defined.

Class	Description
`CAKeyframeAnimation`	An alternative to `CABasicAnimation` that animates the target value with a sequence of discrete steps called *keyframes*, defined by an array of points or a path. Includes an option to pace animations and force a fixed duration on the sequence of steps.
`CAGroupAnimation`	A wrapper for an array of animation objects that can be used together.
`CAMediaTiming`	A protocol implemented by all Core Animation objects that makes it possible to repeat animations, delay them, and apply autoreverse so that they run forward and then backward. Also defines how the final property value is defined after the animation.
`CAMediaTimingFunction`	*Not* a subclass of `CAMediaTiming`. This is a separate class that defines the animation timing curve with either one of four presets or a custom timing Bezier curve. The curve is an object that plugs into the `timingFunction:` property.

You can simplify this nest of classes with the following hints:

- You'll rarely, if ever, use `CAAnimation` and `CAPropertyAnimation` directly.
- For a simple from-to animation, create a `CABasicAnimation`.
- For a complex path animation that follows a list of defined points, create a `CAKeyframeAnimation`.
- Optionally, define an instance of `CAMediaTimingFunction` to control time development. For most animations, the preset timing functions create a good result. Use a custom function for special effects.
- Optionally, set up repeat and autoreverse effects with an instance of `CAMediaTiming`.
- Plug your chosen timing function and timing settings back into your animation object.
- Optionally define a delegate and implement the `animationDidStart:` and `animationDidEnd:` methods. For example, you can use `animationDidEnd:` to run another animation object, creating complex chained animation sequences; or it can just log a message for debugging.
- To replace the default animator, bundle the new animation object into a dictionary and assign the dictionary to the target object's `animations` property.

This looks like a lot of work, but most of it is optional. Some sample code that adds a timing function to the follow-button example follows:

```
-(void) awakeFromNib {
    oldPosition = theButton.frame.origin;
    //Create a basic animation object
    moveAnimation = [CABasicAnimation animation];
    moveAnimation.duration = 2.0;
    //Preset ease-in/out curve
```

```
    moveAnimation.timingFunction = [CAMediaTimingFunction
    functionWithName: kCAMediaTimingFunctionEaseInEaseOut];
    //Extreme custom timing function - pauses before moving
    //moveAnimation.timingFunction = [CAMediaTimingFunction
    functionWithControlPoints:0.99 :-1.0 :0.01 :5.0];
    theButton.animations = [NSDictionary dictionaryWithObject:
    moveAnimation forKey: @"frameOrigin"];
}
-(void) mouseDown: (NSEvent *)theEvent {
    NSPoint mousePosition = [theEvent locationInWindow];
    bOffX = NSMidX(theButton.bounds);
    bOffY = NSMidY(theButton.bounds);
    mousePosition.x = mousePosition.x-bOffX;
    mousePosition.y = mousePosition.y-bOffY;
    [[theButton animator] setFrameOrigin: mousePosition];
}
```

This code creates the custom animation object in `awakeFromNib:` and plugs it into the `animations` property after wrapping it inside a dictionary. The dictionary makes it possible to set different animation objects for different properties.

In this context, the animation key name *must* match the animated property. If `animator` can't find a matching animation key, it falls back to its default animation object. You can, of course, fill the dictionary with the same animation object assigned to different keys.

Using timing functions

Timing functions are objects created by the `CAMediaTimingFunction` function to return an instance of a timing function object. Table 17.3 lists the preset timing functions, with some possible applications.

Table 17.3 PRESET Core Animation Classes

Class	*Description*
`kCAMediaTiming FunctionLinear`	The default value — a linear animation at a constant speed. Can seem unpolished when animating movement, but is ideal for colors and other non-moving properties.
`kCAMediaTiming FunctionEaseIn`	The animation starts slowly and slams to a stop when it completes. Urgent and frantic.
`kCAMediaTiming FunctionEaseOut`	The animation starts suddenly and slows down as it completes. Relaxed and unhurried.
`kCAMediaTimingFunction EaseInEaseOut`	The animation starts slowly, accelerates, and slows again before stopping. This is a sophisticated and natural-looking animation that suggests a physical object moving with inertia.

Creating custom timing functions

The Bezier curve used by the `CAMediaTimingFunction functionWithControl Points:` method can seem unintuitive and difficult to work with, but there is an easy way to use its features.

To control timing, set the two `cx` values so they total 1.0. They then approximate proportions/percentages that define the timing of the initial and final phases of the curve.

The two `cy` values are the "active" values that approximate the animated value. For example, 0.1 is equivalent to 10 percent of the interpolated value, while 0.9 is equivalent to 90 percent, and so on. For example:

```
0.9 : 0.0 : 0.1 : 1.0
```

creates a timing function that pauses for 90 percent of its duration and snaps to its final value in the remaining 10 percent of the time. Because the curve is a Bezier, the control points are interpolated smoothly, which is why this technique is approximate. But you can use it to gain an insight into a function that's difficult to visualize otherwise.

In OS X, the `cy` values are clamped to 0.0 and 1.0. In iOs, they're left unclamped, and you can specify values outside these limits to define overshoots. You can use this trick to create bounce effects. If `cy2` is greater than 1.0, the animated overshoots and then settles back to 1.0 when the animation ends.

Creating and using animation paths

When you need more sophisticated control over the time development of an animation, use `CAKeyFrameAnimation`. You can define keyframes in two ways.

Creating a values/keytimes animation

To create an abstract set of keyframes that can be applied to any animation sequence, create an array that holds a sequence of `NSNumber` objects and pass it to the `values` property. The animation steps through the array and uses the number values to control the target object, with optional interpolation.

Timing is controlled with either the `keyTimes` array or the `timingFunctions` array. `keyTimes` uses `NSNumber` objects to define the duration of each step. The `calculation Mode` property defines the interpolation between each step, which can be linear, discrete — that is, stepped — or paced, producing smooth changes throughout the animation.

`timingFunctions` takes a series of `CAMediaTimingFunction` objects, providing very fine control over the interpolation between points. This can be useful in complex media applications such as video editors, audio sequencers, and games, but it's excessively detailed for simple UI animations.

Creating a path animation

The `path` property is ideal for defining movement paths. It takes a Core Graphics path, which is very similar to a Cocoa path, defined in the previous chapter. Core Graphics is a lower-level framework and uses functions to define path segments. You can set the path's initial starting point, and then add arcs, rectangles, and other shapes. When the animation runs, the object traces the path. Optionally, you can enable rotation.

Points on the path are relative to the animated object's superview origin, which is a complicated way of saying that you can use either view or window coordinates. Depending on the application, you may need to recalculate the path whenever the object moves. In either case, path points are relative to the surrounding view and not the current object position.

Sample code for a keyframed path animation follows:

```
CAKeyframeAnimation *moveAnimation;
CGPathRef aPath;
-(void) awakeFromNib {
   oldPosition = theButton.frame.origin;
   //Create the animation object
   moveAnimation = [CAKeyframeAnimation animation];
   moveAnimation.duration = 1.0;
   //Set the pacing
   moveAnimation.calculationMode = kCAAnimationPaced;
   //Set the timing function, which is applied to the path
   moveAnimation.timingFunction = [CAMediaTimingFunction
   functionWithName: kCAMediaTimingFunctionEaseInEaseOut];
theButton.animations = [NSDictionary dictionaryWithObject:
   moveAnimation forKey: @"frameOrigin"];
}
-(void) mouseDown: (NSEvent *)theEvent {
NSPoint mousePosition = [theEvent locationInWindow];
   bOffX = NSMidX(theButton.bounds);
   bOffY = NSMidY(theButton.bounds);
   mousePosition.x = mousePosition.x-bOffX;
   mousePosition.y = mousePosition.y-bOffY;
   //Create a path
   aPath = CGPathCreateMutable();
   //Set the starting point
   CGPathMoveToPoint(aPath, NULL, oldPosition.x, oldPosition.y);
   //Creates a simple moveto path
   //CGPathAddLineToPoint(aPath, NULL, mousePosition.x,
   mousePosition.y);
   //A more complex path
   CGPathAddCurveToPoint(aPath, NULL, oldPosition.x-80,
   oldPosition.y-80, mousePosition.x+80, mousePosition.y+80,
   mousePosition.x, mousePosition.y);
   //Set the path
   moveAnimation.path = aPath;
   [[theButton animator] setFrameOrigin: mousePosition];
   oldPosition = mousePosition;
}
```

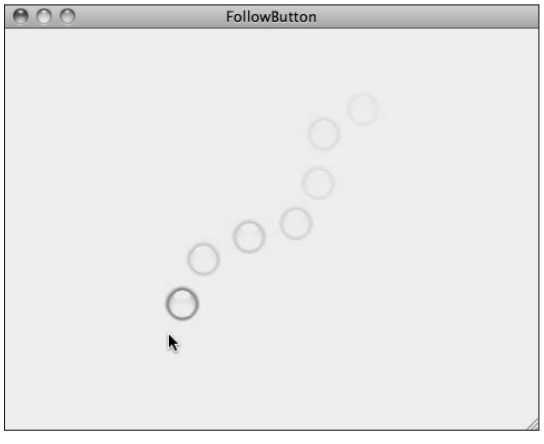
Key features of the code include:

- The code is still using an `animator`.
- The default animation object is replaced by an instance of `CAKeyFrameAnimation`.
- The `calculationMode` property is used to pace the animation, creating a more natural look.
- A timing function is applied to the path as a whole. The animation code automatically applies the timing function as it calculates each point on the path.
- The path is created with the Core Graphics `CGPathCreateMutable()` function.
- The starting point is set with `CGPathMoveToPoint()`, which takes a path pointer, an optional transformation matrix which isn't used here, and two points.
- A curve segment is added with `CGPathAddCurveToPoint()`, which takes four control points and is very similar to `NSBezierPath`.
- The path is recalculated for each mouse click, using the current and previous mouse values.
- The path is assigned to the animation object's `path` property.

The complex path adds a wiggle to the animation, illustrated in Figure 17.4.

Figure 17.4

An animation with a custom path. The animated object wiggles its way from the starting point to the destination. This simple path is created with a single curved path segment. But it could be made arbitrarily complex, with many subpaths.



The path can be made as simple or as complex as it needs to be by adding more path segments. It can also be made noncontiguous for special effects. For a full list of `CGPath` functions, see the CGPath Reference in the Documentation.

Creating Animations with CALayer

You now have most of the tools you need to create completely customized animations without using the `animator` system. An animator is a convenient way to encapsulate an animation object. You can run arbitrary animations by replacing the animator with a more flexible animation component called a *layer*.

Internally, Core Animation creates a hierarchy of animation object components that is similar to the view hierarchy. Instead of views, each item in the hierarchy is a layer, an instance of the `CALayer` class.

A layer is like a view, but it has more properties and is optimized for animation. Technically, it's also a wrapper for the contents of the view. Layers are more flexible than views, and you can use them to create impressive static effects and animations.

A view with a layer is said to be *layer-backed*. By default, views don't have layers. You can add a layer in two ways. In code, use

```
CALayer *theLayer = [theView makeBackingLayer];
```

This creates a layer and returns a pointer to it. You can also create layers in Interface Builder. In the Wants Core Animation Layer pane in the Effects tab, select the check box next to the top view, as shown in Figure 17.5. If the view is a container view with multiple objects, you don't need to select every box; it's enough to select the top one. Layers for subviews are created automatically.

CAUTION

If you try to apply animations to a view that isn't layer-backed, your application may sometimes crash. Officially, you must create a layer. Unofficially, simple `animator` animations seem to work without one. But it's not a good idea to rely on this, so remember to create a layer whenever you use animations.

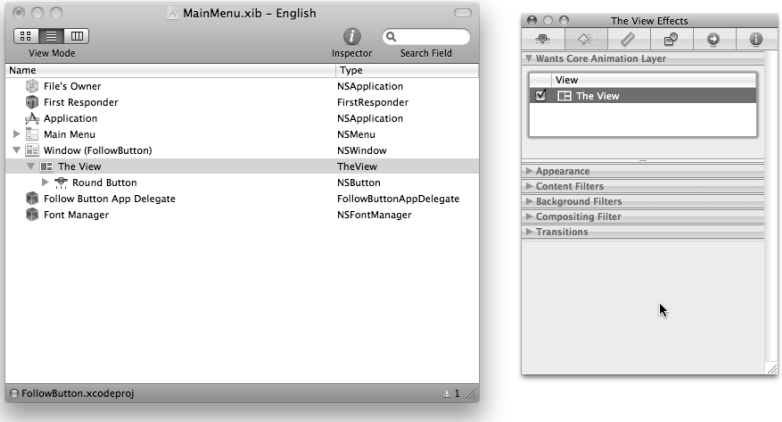
Using layers for animation

There's surprisingly little difference between the code used to create an `animator` and the code used to create and run a layer-backed animation. The main differences are as follows:

- You must use the `layer` method to return a view's layer object.
- You must use the `addAnimation:` method to run an animation on a layer.
- You can give the animation object a name and use the name as a key string.
- Instead of running the animation on a property, you can specify a keypath for the property when you create the animation.

Figure 17.5

Making a view layer-backed to enable animation and other layer features



In outline, the differences can be summarized with one word — keypath. With layer animation, the keypaths are explicit.

As an example, you'll use a modified version of the Twirl Filter project from the previous chapter, replaceing the twirl filter with a torus lens distortion filter and animating the filter so that the distortion appears to spread out from a mouse click.

Creating an animatable filter

In the last chapter, you looked at the Core Image filters that can be applied to the background or the contents of any view or that can be used to composite — blend — two views together. In this section, you'll create a different implementation and make it animatable.

You can apply static filters to a view in Interface Builder by adding them in the Effects tab, or you can implement them with customized drawing code that initializes and runs a filter directly. A third option is to create and initialize a filter and add it to the `backgroundFilters` or `contentFilters` properties of a view. As you'd expect, this is the code equivalent of adding the filter in Interface Builder.

TIP

You can use the Color Monochrome and Color Controls filters to change, and optionally to animate, the background color of a button. This breaks any number of Aqua design guidelines, but sometimes a pulsing or color-cycling button is exactly what an application needs.

This sample code creates a filter in a view's `awakeFromNib` method and updates its `inputCenter` key when the user clicks with the mouse:

```
-(void) awakeFromNib {
    //Load the demo image from the bundle
    myBundle = [NSBundle mainBundle];
    sourceImagePath =
    [myBundle pathForResource:@"b1"ofType:@"jpg"];
    sourceImageURL =
    [NSURL fileURLWithPath: sourceImagePath];
    sourceImage = [[NSImage alloc] initWithContentsOfURL:
    sourceImageURL];
    self.image = sourceImage;
    self.imageScaling = NSScaleToFit;
    //Set a default filter center in the middle of the view
    boundsRect = [self bounds];
    filterCenter = NSMakePoint(boundsRect.size.width*.5,
    boundsRect.size.height*.5);
    //Create and initialize the filter
    thisFilter =
    [CIFilter filterWithName:@"CITorusLensDistortion"];
    [thisFilter setDefaults];
    thisCenter = [CIVector vectorWithX: filterCenter.x Y:
    filterCenter.y];
    [thisFilter setValue: thisCenter forKey: @"inputCenter"];
    [thisFilter setValue:[NSNumber numberWithFloat:0]
    forKey:@"inputRadius"];
    [thisFilter setValue:[NSNumber numberWithFloat:200]
    forKey:@"inputWidth"];
    [thisFilter setValue:[NSNumber numberWithFloat:1.5]
    forKey:@"inputRefraction"];
    thisFilter.name = @"torus";
    [self setContentFilters:
    [NSArray arrayWithObjects:thisFilter, nil]];
}
-(void) mouseDown: (NSEvent *) theEvent {
    //Get the click location, using superview coordinates
    clickedPoint =
    [self.superview convertPoint:
    [theEvent locationInWindow] toView: self];
    //Change the filter center
```

```
    if ([self mouse: clickedPoint inRect: [self bounds]]) {
       filterCenter = clickedPoint;
       thisCenter =
       [CIVector vectorWithX: filterCenter.x Y: filterCenter.y];
       [self setValue: thisCenter forKeyPath:
       @"contentFilters.torus.inputCenter"];
    //[self animateFilter];
    }
}
```

This is pure setup and update code for the filter. There's no animation — yet. The code so far:

1. **Loads an image into `self`, which is a subclassed `NSImageView`.**
2. **Finds the center of the view.**
3. **Creates a filter object of the `CITorusLensDistortion` type.**
4. **Sets initial values for the value, including the default center point.**
5. **Names the filter.** The name will be used for keypath access.
6. **Creates a filter array with the filter as the only object, and assigns the array to the view's `contentFilters` property.**
7. **Uses a `mouseDown:` method to respond to mouse clicks by converting the mouse position into a `CIVector` and updating the filter's `inputCenter` property.**

Most of this code is straightforward, but a couple of lines need further explanation. The default `inputRadius` is set to 0, which means that if you run this code as is, the filter has no obvious effect. This is irrelevant because you'll be animating this property. But if you want to test this code to create a static filter effect, change the radius to a larger value, such as 200.

The line

```
[self setValue: thisCenter
forKeyPath: @"contentFilters.torus.inputCenter"];
```

uses KVC to set the filter's `inputCenter` property. The keypath selects the filter with the matching name — `torus` — and then accesses the specified property.

You *must* use KVC in this way to update filter values. If you try to change the `inputCenter` property directly with

```
[thisFilter setValue: aCenter forKey: @"inputCenter"];
```

nothing happens. You might think `setValue:` would be enough to trigger a setter method, but it isn't. Once the filter is active, you must use the view's accessors to set properties indirectly; the filter's own accessors no longer work.

CAUTION

There's some inconsistency in the filter keypaths, as you'll see below.

Although this code implements a torus refraction filter, it's a general solution you can use with any Core Image filter. It's less complex than the code in the previous chapter, and it's easy to animate. The low-level code in the previous chapter works on the image data directly, making it a better solution when you need to copy the results of the image processing and re-use them in a different context, save them to a file, or batch process images. This version is better suited to display effects.

Animating the filter

To animate the filter, follow these steps:

1. **Create an animation object that works on a filter property via a keypath.**
2. **Initialize from/to values or create keyframes.**
3. **Set the duration, timing function, and other essential parameters.**
4. **Run the animation by calling** `addAnimation:` **on the view's layer.** This method automatically runs the animation as soon as it's called.

Sample code follows:

```
-(void) animateFilter {
    animation =
    [CABasicAnimation animationWithKeyPath:
    @"filters.torus.inputRadius"];
    animation.fromValue = [NSNumber numberWithFloat:0];
    animation.toValue =[NSNumber numberWithFloat: 1000];
    animation.duration = 3.0;
    animation.delegate = self;
    animation.timingFunction = [CAMediaTimingFunction
    functionWithName: kCAMediaTimingFunctionLinear];
    [[self layer] addAnimation:animation forKey:@"torus"];
}
```

Most of this code is similar to the animations you've already seen. The critical difference is the target keypath. You must set the keypath correctly; otherwise the animation runs without modifying values. Because you're setting a content filter, you might expect the keypath to be

```
@"contentFilters.torus.inputRadius"
```

It isn't, even though you can use this keypath to access filter values without animation. The correct keypath is

```
@"filters.torus.inputRadius"
```

However, if you wanted to add the filter to the view's background filter set and animate it, the correct keypath would be

```
@"backgroundFilters.torus.inputRadius"
```

You can find the correct keypaths for other animatable properties in the Animatable Properties list in the Core Animation Programming Guide. Some of the keys have slightly random names. It's useful to double-check them before adding them to your code.

To run the animation, trigger the `animateFilter` method. In the previous code, I added it to the `mouseDown:` method, but left it commented out. Uncommenting it will run it when the user clicks the mouse. Potentially, you could trigger the effect as a result of other user actions, a slow timer, or even incoming Internet traffic. Figure 17.6 shows the finished application.

Figure 17.6

Creating an animated filter effect, triggered by mouse clicks

TIP

The example sets the animation radius to a fixed to-value of 1000. For a more sophisticated effect, try setting the to-value to a number related to the view's bounds. This guarantees that when the view is resized on a large monitor, the animation won't stop suddenly before it reaches the edges of the view.

One final nicety is the use of a delegate. The sample code for this example on the Web site for this book (`www.wiley.com/go/cocoadevref`) has minimal implementations of the `animationDidStart:` and `animationDidEnd: finished:` methods that log a message to the console. In a more complex application, you can use them to trigger associated features, including other animations, as each animation starts and ends. With expanded code, it's possible to animate many filters simultaneously, creating extremely rich visual effects.

NOTE

You can use similar code to animate any animatable property. Conventional properties have much simpler keypaths. Typically you use the property name — there's no need to create a custom filter or name it. However, you can use this more complex code to experiment with animating other Core Image filters.

Using OpenGL

OpenGL is designed for stand-alone 3D scene rendering. It's a separate animation technology and isn't directly connected to Core Animation.

NOTE

Behind the scenes, Core Animation creates its effects with OpenGL calls, but there's no API (Application Programming Interface) for the connection between the two. You can use Core Animation or OpenGL, but you can't work with the interface between them.

OpenGL is powerful, but complex and challenging, and has a very steep learning curve. A full introduction is outside the scope of this book, but the rest of this chapter is an introduction to Cocoa's `NSOpenGLView` class, which provides a simple wrapper for OpenGL code that runs in an OpenGL context.

`NSOpenGLView` is a hybrid view class with most of the features of `NSView` and three unique methods. The `reshape` method is called when the view is resized. `drawRect:` is called to render a scene into the view. Both methods support OpenGL code as well as standard Cocoa objects and features. The remaining method is `openGLContext`, which runs on the `NSOpenGLContext` class and returns a context for the drawing code.

Optionally, you can optimize an application by creating another body of code that is used to set up the key features of an OpenGL scene. Typically, you implement this with a custom method with a suitable name, such as `setUp`. `setUp` is run once, when the application loads. It sets the background color of a scene, creates and initializes lights, and so on. This method isn't obligatory. You can — and sometimes have to — re-run the same setup code for each refresh; but it's efficient to split run-once code from run-often code.

Introducing OpenGL

OpenGL code is a list of functions and parameters. Functions are grouped into categories that define shapes, manage lighting, define textures and colors, control the virtual viewport, and so on. For example,

```
glEnable(GL_LIGHTING);
```

turns on the lights in a scene, assuming that some lights have been defined. Parameters are typically passed as predefined constants such as `GL_LIGHTING`, float literals, or pre-packed C arrays.

Shapes are defined with polygons in 3D space. If the polygons are small enough, it's possible to create surfaces that appear to be smooth and curved. OpenGL offers various ways to define arbitrary shapes, but the simplest — and least flexible — is to call on a set of optional functions in a library called GLUT (OpenGL Utility Toolkit) that can add predefined shapes to a scene. In this example, you'll use the GLUT library to add a standard teapot.

NOTE

The original teapot model was created in 1975 at the University of Utah by Martin Newell. Since then it has become a standard 3D test shape. The model was digitized from a German Melitta teapot, which is currently owned by the Computer History Museum in Mountain View, California. It is no longer used for tea.

Shapes are rendered with materials, which are lit by at least two different kinds of light sources. Materials can be very simple or extremely complex. Materials have no physical properties such as mass or elasticity. In OpenGL, they're defined by how they reflect light. In this example, you'll use an extremely simple single-color material, ambient background lighting, and a single light with both specular (highlight) and diffuse light components.

Once a scene is defined and lit, it's processed through various matrix transformations to create the final view. The transformations define a virtual camera in the scene, with a variable field of view, pointing in a specific direction. Perspective and field-of-view effects are both implemented. The reshape method recalculates the matrices when the view is resized.

Objects are placed in the 3D world with reference to a virtual drawing matrix. Initially the matrix is at the origin with unity scaling and no rotation. But it can be translated (moved), scaled, or rotated as needed. In this example, you'll learn how to offset the teapot slightly and add an animated rotation effect.

Creating an OpenGL animation

To animate an OpenGL scene, use a timer that calls `[self setNeedDisplay];` to trigger a `drawRect:` redraw of the scene on each timer tick. In a complex animation, the position, orientation, size, and sometimes also the texture and lighting of the scene are recalculated at each timer tick. `drawRect:` then uses the recalculated positions when it refreshes the scene.

If the timer ticks more than 25 times a second, the scene animation appears smooth. In this example, you use the timer to update a variable that defines the rotation of the scene.

An OpenGL scene is defined by a long list of functions. Typically the code does the following:

1. Initializes lights and materials
2. Defines the viewport and camera position
3. Places objects in the scene
4. Calls `glFinish()` to render the scene

The `glFinish()` function compiles the scene code and runs it, creating the finished view. Sample code for the teapot example is as follows. The code uses the openGL and GLUT libraries, so you must add these to your project, with their headers.

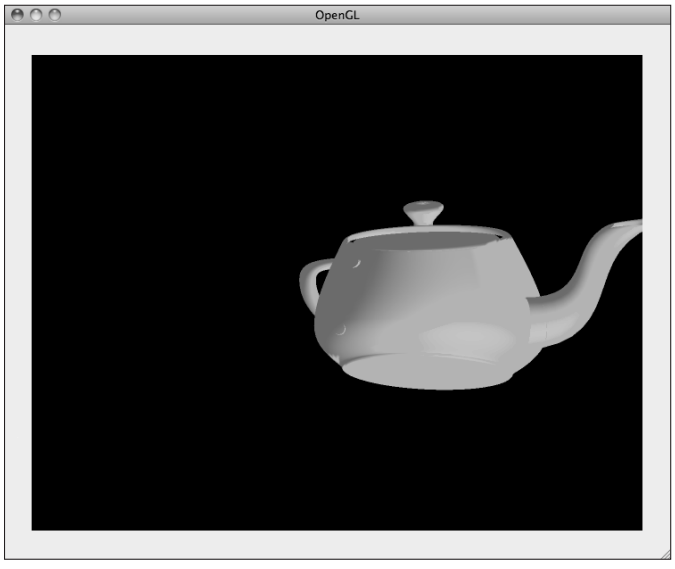
```
-(void) timerMethod {
//Increment the rotation value, and refresh the display
   rotationCount +=rotationStep;
   [self setNeedsDisplay: YES];
}
- (void)drawRect:(NSRect)dirtyRect {
   //Setup a context
   NSOpenGLContext *glContext = [self openGLContext];
   [glContext makeCurrentContext];
   //Basic setup
   glEnable(GL_LIGHTING);
   glEnable(GL_DEPTH_TEST);
   glDepthFunc(GL_LEQUAL);
   glClearDepth(1.0);
   glEnable(GL_CULL_FACE);

   //Ambient light
   GLfloat ambientLight[] = {ambientBrightness,
   ambientBrightness, ambientBrightness, 1.0};
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);
   //Add a diffuse white light
   GLfloat diffuseLight[] = {diffuseBrightness,
   diffuseBrightness, diffuseBrightness, 1.0};
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
   //Some specular highlights, same value as the diffuse light
   GLfloat specularLight[] = {diffuseBrightness,
   diffuseBrightness, diffuseBrightness, 1.0};
   glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
   //Turn on the light
   glEnable(GL_LIGHT0);
```

```
    //The material is sort of blue-ish
    GLfloat theMaterial[] = {0.25, 0.8, 1.0, 1.0};
    glMaterialfv(GL_FRONT, GL_AMBIENT, theMaterial);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, theMaterial);
    glMaterialfv(GL_FRONT, GL_SPECULAR, theMaterial);
    //Background color is black
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //Set the projection view matrix
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(fieldOfView, baseRect.size.width/baseRect.size.
    height, 0.1, 10);
    //Set the model view matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //'Eye' position - looking straight ahead
    gluLookAt   (0,0,zEye,
                 0,0,0,
                 0,1,0);
    //Position the light
    GLfloat lightPosition[] = {diffuseX, diffuseY , diffuseZ};
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
    //***Object drawing code***
    //Center the drawing position
    glTranslatef(0.0, 0.0, 0.0);
    //Include some rotation
    glRotatef(rotationCount, 0, 1, 0);
    //Move one step to the right
    glTranslatef(1.0, 0.0, 0.0);
    //Draw the teapot
    glutSolidTeapot(1.0);
    //Render the scene
    glFinish();
}
-(void) reshape {
    //Resize the viewport to support the new view rect
    baseRect = [self convertRectToBase:[self bounds]];
    glViewport(0, 0,  baseRect.size.width, baseRect.size.height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(fieldOfView, baseRect.size.width/baseRect.size.
    height, 0.1, 10);
}
```

The finished scene is shown in Figure 17.7.

Figure 17.7

Rotating a teapot. The teapot rotates about its handle, so it's offset from the center of the view.



Controlling an OpenGL animation

It would be useful to have some control over the parameters that define the scene. In this example, the code includes variables for some of the key values, such as the position of the light, the brightness of the ambient and diffuse light options, and the camera location. You can insert float variables directly into OpenGL code. In a more complex project, you can use variable arrays to modify shapes and materials in real time.

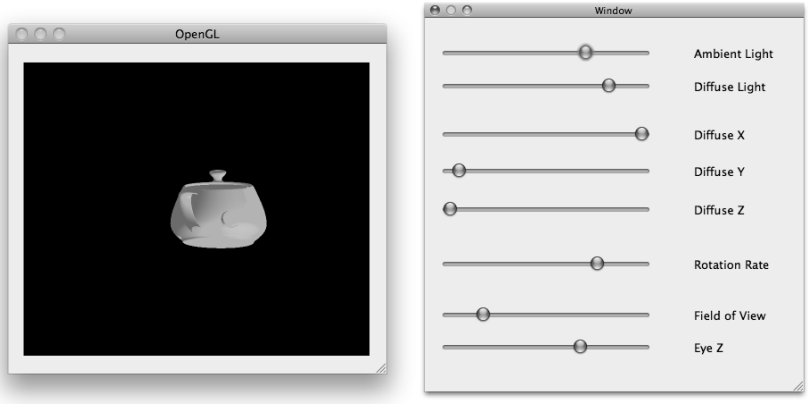
To set the variables, create a controller class that manages a collection of sliders in a separate pane. The controllers write values to the application's preferences, using the technique and code introduced in Chapter 13. The view reads values *from* the preferences using Key-Value Observing, and passes them to scene variables whenever the user moves a slider. Passing values through preferences means that the scene automatically saves the current slider state when it quits, and reloads it when it restarts.

The full code listing is long, somewhat repetitive, and almost identical to the code in Chapter 13. It isn't repeated here, but the full project is available for download from the Web site for this book (`www.wiley.com/go/cocoadevref`).

The completed application is shown in Figure 17.8.

Figure 17.8

Controlling scene parameters with a separate controller pane — one of many possible ways to control an OpenGL scene.



As an exercise, you can try to improve the project. In this version, the timer animation stops when the user moves a slider. Try to move the timer and the `drawRect:` method to a sub-thread so that they run independently of the UI. Chapter 11 has some hints about thread management.

You can also create a more sophisticated OpenGL view by adding extra lights, customizing the texture, and improving the depth culling that controls the order in which the visible surfaces are drawn.

TIP

A useful series of unofficial tutorials is available at `http://nehe.gamedev.net`. The Cocoa versions of the projects are out of date now, but you can extract the code from the Web site examples and copy it into the OpenGL project sample code. Official documentation is available at `www.opengl.org`.

Summary

In this chapter, you learned how to create simple timer-driven property animations by using a timer method to update property values. You were introduced to `animator` objects and saw how they could add simple animation effects with very little code.

Next, you discovered how to customize an `animator` by replacing its default animation object with one that you created and defined yourself. As a natural next step, you learned how to replace an animator with a layer object and discovered how to create a more complex animation effect that could be applied to a Core Image filter.

Finally, you were introduced to OpenGL and learned about the fundamentals of OpenGL scene animation. You explored the `NSOpenGLView` view class and used it to create a simple animated scene, with variable parameters set from an associated control panel.

18 Debugging, Optimizing, and Managing Code

Xcode includes a powerful suite of tools for profiling, testing, and debugging code. You can test your application in three ways:

- Logging messages to the console.
- Setting breakpoints, single-stepping through code and checking property values in a debugging window.
- Using *instruments* — a collection of test tools that can report on memory use, thread states, application performance, object allocations, and memory leaks.

By default, Xcode supports two different build configuration options for both OS X and iPhone OS projects: Release and Debug. You can select these using the build drop-down menu at the top left of the Xcode window, as shown in Figure 18.1.

The Debug option creates a supporting symbol table. The Release build doesn't create this table, producing a much smaller binary.

On OS X, you can use the Release build as a final distribution version. On the iPhone, the default Release settings don't create a build for the App Store; they simply remove the symbol table. For information about creating a valid App Store release build, see Chapter 19.

18 In This Chapter

Using the console and `NSLog`

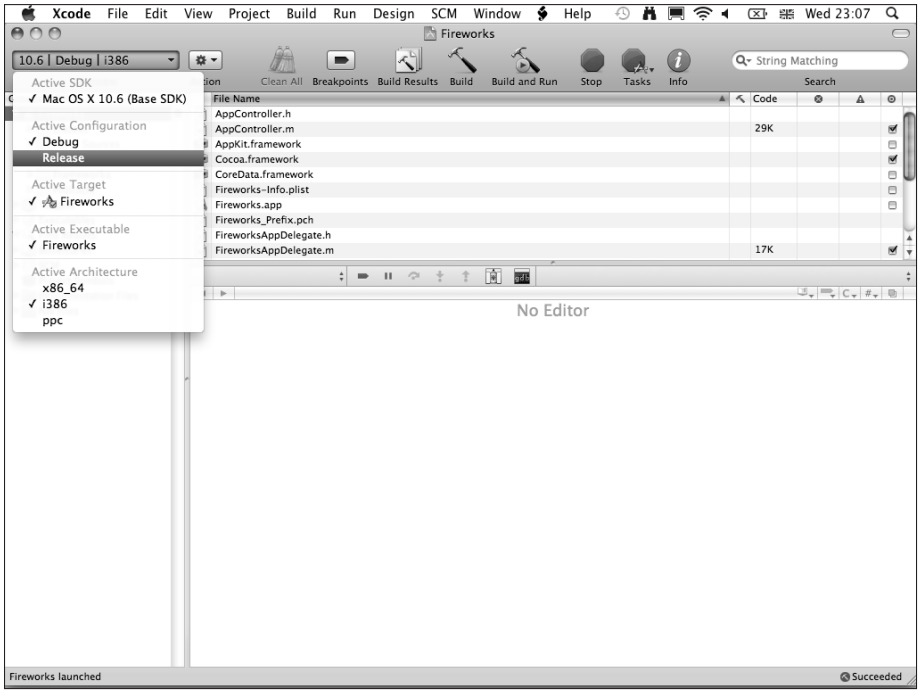
Debugging with breakpoints and the Debugger window

Using instruments

Managing code with Snapshots and Source Control

Figure 18.1

Selecting Debug and Release builds in Xcode. You can use the default Debug build settings for basic debugging. For more advanced techniques, you may need to modify the Debug configuration by hand.



Using the Console and NSLog

You'll learn about debugging using the counter application from Chapter 8: it's easy to modify but complex enough to introduce the theory and practice of profiling and debugging. Reload the application into Xcode, and click CounterAppDelegate.m. You'll use this code as a testbed for the debugging and performance reporting tools built into Xcode.

Getting started with NSLog

You've already worked with `NSLog`, but in this chapter you'll use it in a structured way as a debugging aid. `NSLog` writes strings to the application's console. The console is hidden in most Cocoa applications, but you can display it in Xcode while testing an application, and you can generate `NSLog` strings in the code to report variable values and application events.

The console is a terminal attached to `stdout`, and you can also write to it using the many variants of `printf` and their associated formatting options. `NSLog` is optimized to support Cocoa objects. `printf` is better suited for logging C-language strings and values. You can use either or both in an application.

To display the Console window, choose Run➪Console in Xcode. The window includes a subset of Xcode's standard toolbar and a Clear Log button that deletes the current log. Click the Build and Run button to build and run an application in the usual way, and click the Restart button to restart an application after a crash. You can also click the Pause button to pause the application temporarily. This displays a command line prompt, which you can use for command line control of debugging and logging.

The Console window and the Debugger window, which are described later in the chapter, are both wrappers for the underlying gdb (GNU Project Debugger) debugging environment. If you're a gdb expert, you can use gdb's command line options in the Console window to access gdb's advanced features.

gdb is a complex and powerful tool with some very sophisticated features. Xcode's debugging features barely scratch the surface of what it can do. But it's a command-line oriented environment and isn't very friendly or accessible. You can successfully debug most applications without accessing it directly.

A full discussion of gdb's features is outside the scope of this book. Apple's gdb manual is available at `http://developer.apple.com/mac/library/documentation/DeveloperTools/gdb/gdb/gdb_toc.html`.

Using NSLog to report events

Event logging is the simplest of all possible debugging options, but it can be powerful and useful, even in complex applications. `NSLog` is simple and easy to work with. With a few edits you can automatically generate a list of application events and create a record of the methods that are being triggered. Use this to test that methods are being triggered when they should be. For example, you can check that the correct method is being triggered by a user action. Typically, the log message is added to the top of the method, but it can also be embedded in conditional code for more selective reporting.

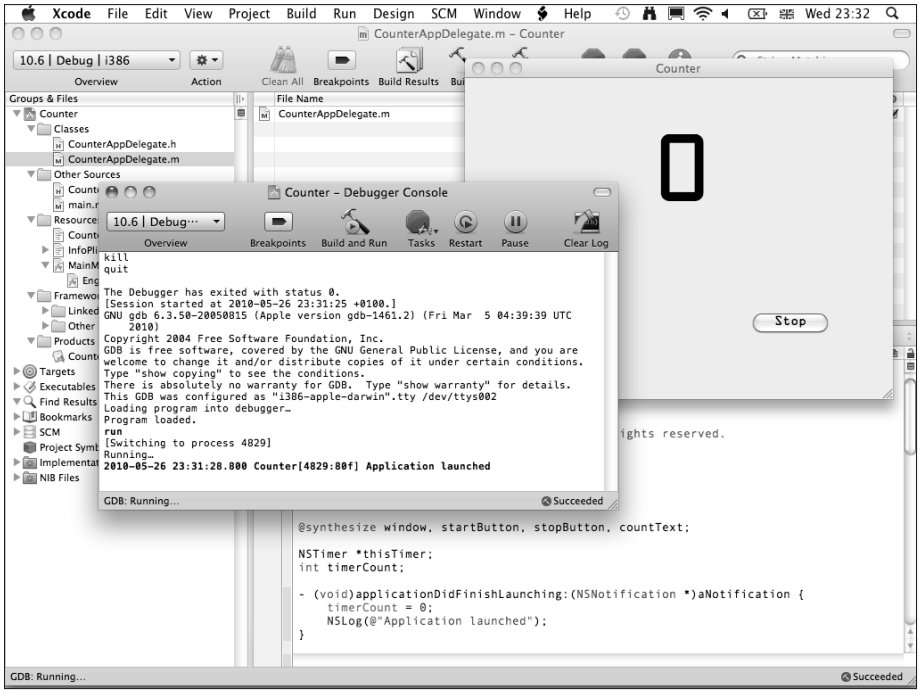
To demonstrate the first option, edit the `applicationDidFinishingLaunching:` method in the App Delegate to include an `NSLog` statement:

```
- (void)applicationDidFinishLaunching:
    (NSNotification *)aNotification {
 NSLog(@"Application launched");
 timerCount = 0;
}
```

Build and Run the application. Figure 18.2 shows the result. The code logs a message when `applicationDidFinishLaunching:` is triggered. In the sample application for this chapter, logging is added for all methods so that you can monitor when they're triggered. The date/time string and the hex addresses are generated automatically.

Figure 18.2

Using `NSLog` to report application events. Add an `NSLog` statement to the beginning of every method that you want to monitor. You can use the same technique within conditionals and switch statements to monitor program flow.



Using NSLog to report values

You can add a format string to `NSLog` to report values. The format string options are C-standard. The format string must be "objectified" with a prefixed `@` to prevent crashes. Table 18.1 shows a selection of useful format options.

Table 18.1 Useful NSLog Format Options

Option	*Used for*
`%i` or `%d`	signed int
`%u`	unsigned int
`%f`	float/double
`%x` or `%X`	int as hexadecimal

Option	Used for
%p	memory address (similar to %x, with a standard 0x prefix)
%zu	size_t
%@	object
\r	new line

For example, to log the value of an int called anInt use:

```
NSLog(@"Int value is: %i", anInt);
```

When the format string doesn't match the item to be logged, NSLog produces garbage. Technically, it attempts to read sequential memory locations and assemble them into a string, but the results are useless, unpredictable, or crash-prone, depending on the context.

Objects are treated in a special way. The standard idiom for logging any object is

```
NSLog(@"%@", anObject);
```

Because there's no standard format description for an object or its contents, different objects respond to logging requests in different ways. The default output is the class name and the object's memory address:

```
<ClassName: 0x123456>
```

It's sometimes useful to log addresses while debugging. If you need to check a selection of different object addresses and their properties, it's easier to do it with Xcode's debugging tool, which includes powerful features for viewing objects and their values.

If a class implements a method called description, this method is triggered by logging, and it returns a description string. For NSString, this generates the string value. For NSArray, NSDictionary, and NSSet, the method attempts to dump the contents of the collection into a readable string. Certain image-handling objects also support this feature. The results may be useful or completely unhelpful, depending on the application.

As a shortcut, it's possible to pass string objects to NSLog directly — no format string is needed. This idiom generates a compiler warning but usually works correctly:

```
NSString *aString = @"Some characters";
NSLog(aString);
```

This is only true for NSString objects. Other objects will cause a crash.

CAUTION

It's possible — rarely — to crash `NSLog` with strings that contain escape characters, especially those used for URL and network objects. You can sometimes fix this problem by prefixing the string with the `@"%@"` format specifier.

Using NSLog to report line numbers and function names

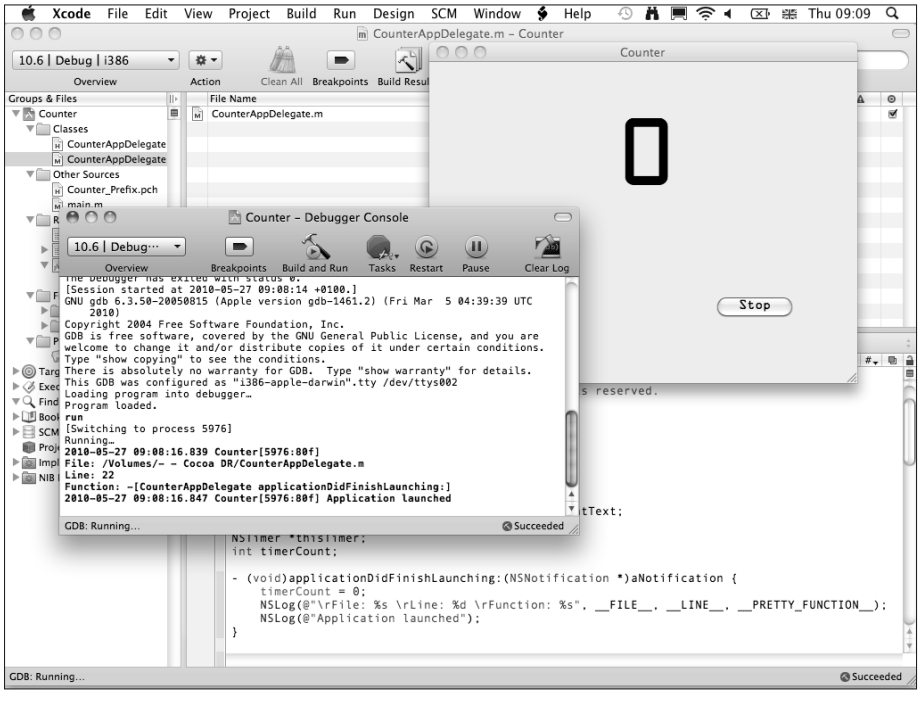
Xcode defines a number of build and debug variables. You can reference them to report debugging information. For example,

```
NSLog(@"%s %d %s", __FILE__, __LINE__, __PRETTY_FUNCTION__);
```

logs the filepath of the source file, the line number in the file, and the function name, which is a combination of an object name and the method within the object. `PRETTY_FUNCTION` is used to improve the formatting of this feature. The sample code for this chapter includes an example with added newline characters to split the result across multiple lines. Sample output is shown in Figure 18.3.

Figure 18.3

Logging line, file, and function information. The information is listed above the `Application launched` string. In this example, extra newline characters — visible in the code window — have been added to improve the formatting.



NOTE
The prefixes and suffixes for these special features are double underscores, created by typing an underscore character twice.

Selectively enabling NSLog

`NSLog` is slow and expensive, and it can destroy the performance of applications, especially if it appears in a critical loop. This is particularly obvious on the iPhone. If the loop is driven by a timer — for example, a screen refresh loop — you can sometimes simplify debugging by temporarily dropping the timer rate, giving `NSLog` time to work.

More typically you'll want to generate `NSLog` messages while debugging but remove them from release code. It's possible to do this manually, creating a special release build with selected code disabled or removed. In a typical application, you'll add `NSLog` events to multiple methods in multiple objects. Commenting them out by hand is a time-consuming and error-prone solution.

Intuitively, you might expect that it would be possible to solve the problem with a conditional definition:

```
#ifdef DEBUG
 NSLog(@"A message");
#endif
```

This is inefficient — adding extra conditional code for every log statement is wasted extra effort — and it doesn't work. Although there are separate debug and release modes, by default Xcode doesn't implement a `DEBUG` flag. You can define one of your own, but a quicker and simpler solution is to redefine `NSLog` as a blank value in the application's `_Prefix.pch` file:

```
#define NSLog
```

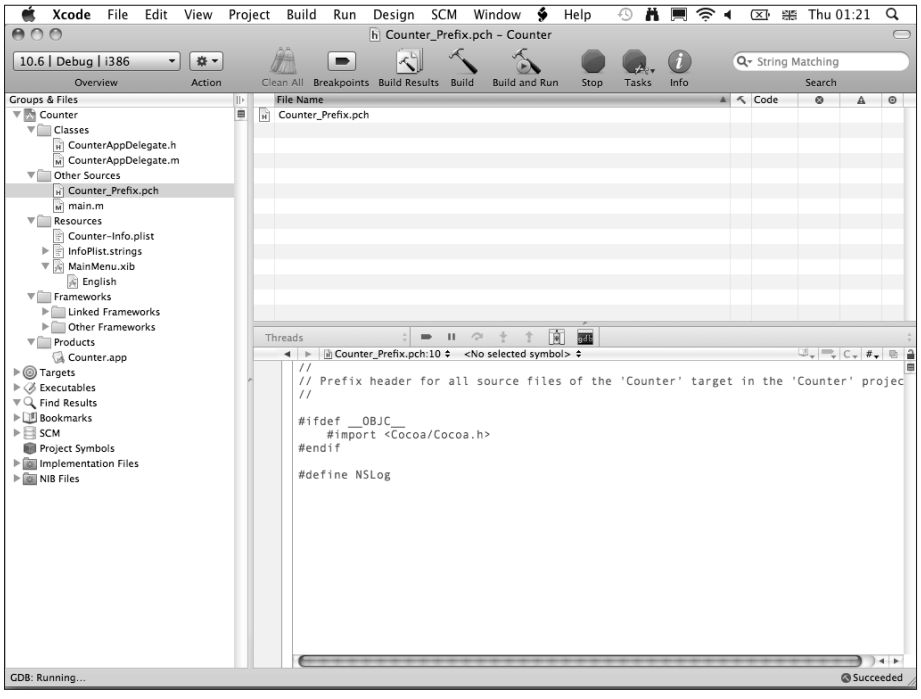
This simple one-line directive, shown in Figure 18.4, defines `NSLog` as a blank function and eliminates all `NSLog` output. You can either comment out this line while debugging or add it before a release build.

More complex solutions are possible. Entire alternative logging classes and frameworks are available online. A popular solution uses macros to create a lightweight customizable replacement. Again, the code is added to the application's `_Prefix.pch` file:

```
#define MY_DEBUG_MODE
#ifdef MY_DEBUG_MODE
    #define MyLog( s, ... ) NSLog( @"<%p %@:(%d)> %@", self,
   [[NSString stringWithUTF8String:__FILE__] lastPathComponent],
   __LINE__, [NSString stringWithFormat:(s), ##__VA_ARGS__] )
#else
    #define MyLog( s, ... )
#endif
```

Figure 18.4

The easy way to disable output from `NSLog`. This is a quick hack and it isn't guaranteed, but it's very efficient and it usually does the job.



Instead of `NSLog`, call `MyLog`. This sample code logs the address, filename, and line number of each statement as a prefix to the standard `NSLog` output. You can customize this feature as needed, adding or removing logging features. You can also define multiple log types and modes so that debugging logs are stripped in release code, but occasional critical log events can be retained. Comment out `MY_DEBUG_MODE` for a release build.

TIP

It's good practice to give the custom debug option a unique or even a personalized name. Don't call it DEBUG or DEBUG_MODE. If you import a framework or library created by another developer, these definitions may already exist.

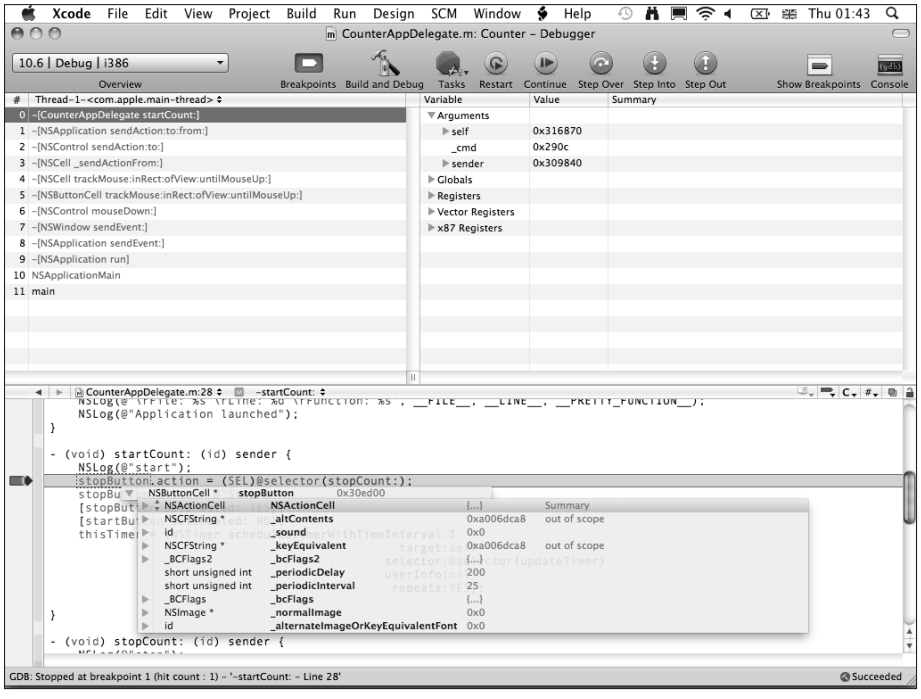
Debugging with Breakpoints and the Debugger Window

Xcode's Debugger window is shown in Figure 18.5. It includes the following features:

- Breakpoints
- Single-stepping
- Crash dumps
- Variable review

Figure 18.5

The Xcode debugger is based on the standard GNU debugger, with a customized Mac interface.



Xcode's debugging features are simple but powerful. Debugging options are closely integrated with the code editor. It's possible to review variables and step through code in the code editor without opening the separate optional Debugger window, which can provide extra detail about code paths and object contents.

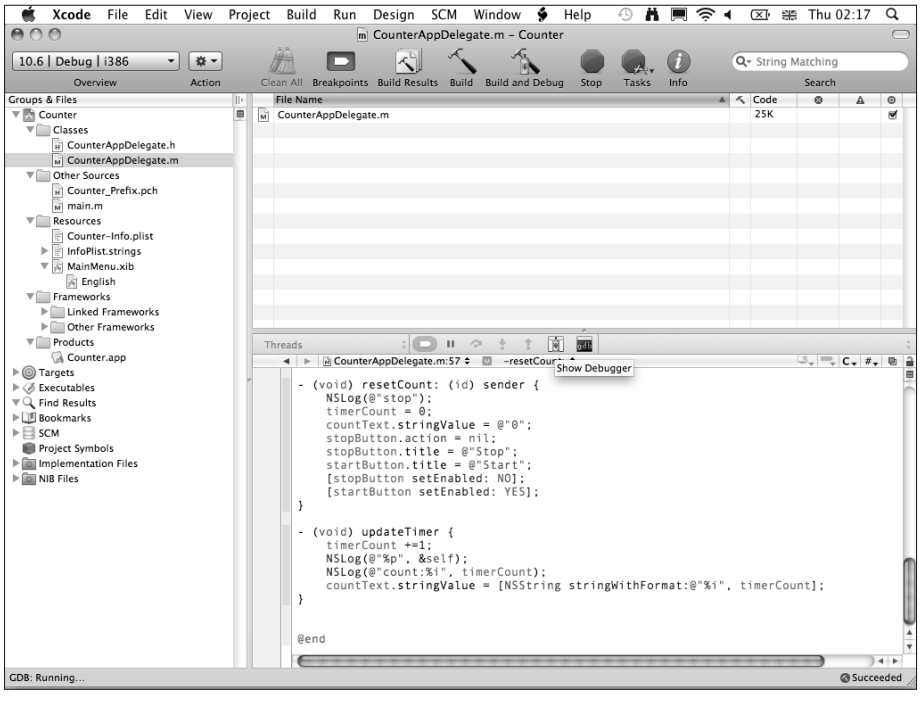
Enabling debugging

The debugger environment runs in a special debug mode, which enables breakpoints. Note that selecting a Debug configuration or creating a debug flag *does not* enable the debugger — it creates a binary that the debugger can work with, but it doesn't run the debugging environment.

To run it, click the Breakpoints button on the main Xcode toolbar. The button changes state, and the Build and Run button is replaced with Build and Debug, illustrated with a stylized spray can, as shown in Figure 18.6. Alternatively, you can add a breakpoint to the code, as described below — this automatically enables debugging.

Figure 18.6

Xcode in debug mode. The Breakpoints button is selected, and the Build and Run button is replaced by the Build and Debug button. Note that the separate Debugger window doesn't appear automatically when Debug mode is selected.

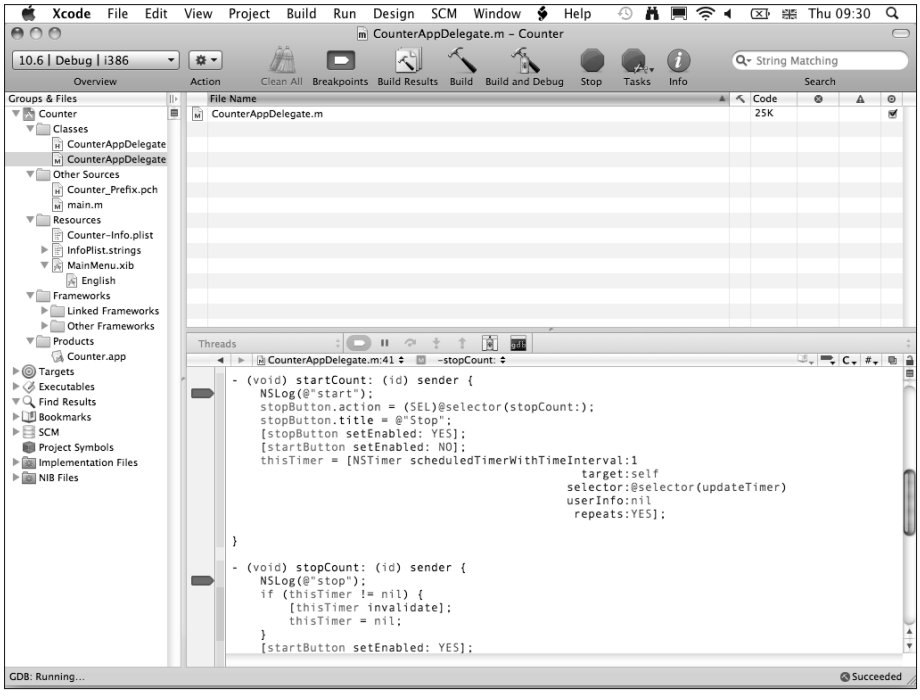


Setting and using breakpoints

To set a breakpoint, select a line of code in Xcode's code editor and then choose Run ➪ Manage Breakpoints ➪ Add Breakpoint at Current Line. An arrow icon appears in the margin to the left of the line. Figure 18.7 shows the result of adding a breakpoint at the start of the `startCount:` and `stopCount:` methods.

Figure 18.7

Adding breakpoints: the code window displays a blue arrow icon in the margin next to each breakpoint. Here breakpoints have been added at two `NSLog` calls, but you can add a breakpoint on any line.



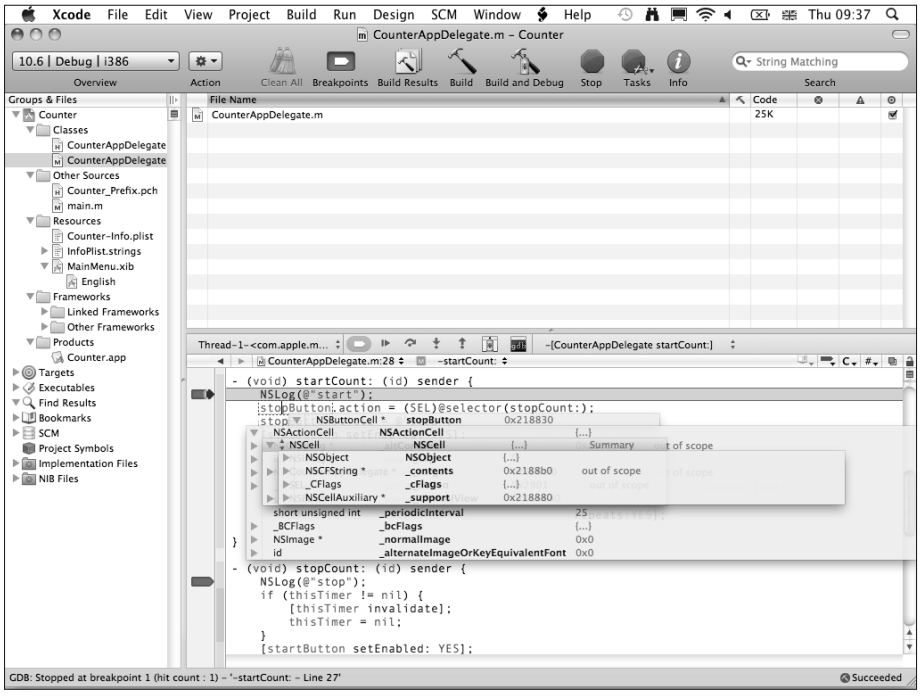
Click Build and Debug, and the application builds and runs with added breakpoint information. When the code reaches a breakpoint, it pauses and the arrow icon displays a smaller extra brown arrow. The line with the breakpoint is highlighted, as shown in Figure 18.8.

You can do three things at a breakpoint:

- Review variables and properties
- Single-step through a method
- Continue execution

Figure 18.8

Stopping at a breakpoint. The breakpoint line is highlighted in blue, and you can hover over objects and variables to explore their values. The values appear in floating overlay views.



If you hover the mouse over any object or variable in the method around the breakpoint, you'll see a floating overlay. When you hover over a simple variable, the overlay displays the item's address and its value. Objects are more complex. You can drill down through the object's structure to view its contents. Hover over each item's reveal triangles to display further overlays.

Cocoa objects are very complex, and it's not always easy to find specific values within them. The overlay review feature is a good way to check the values of simple C variables, but objects can be more challenging. You may find it's easier to check method properties with extra debugging code that logs their values or by adding extra dummy assignments that you can breakpoint as needed.

The review feature has limited but incomplete support for dot syntax properties. It's sometimes possible to review a specific property by hovering over it — for example, holding the mouse over `aProperty` in a line that includes `anObject.aProperty`. This doesn't always work, so you'll need to experiment to check if it's available at any specific breakpoint.

Controlling breakpoint execution

The icons above the code editing window in Xcode control execution after a breakpoint and provide quick links to the Console and the Debugger window. The toolbar is shown in Figure 18.9. From left to right you can:

- Toggle breakpoints
- Continue execution to the next breakpoint, if there is one
- Step over a method or function
- Step into a method or function
- Step out of the current method or function
- Open the Debugger window
- Open the Console window

The step-over and step-into options single-step through code. Stepping out of a method or function "disappears" the execution marker when execution continues within OS X. (Unsurprisingly, you're not allowed to view the OS X source code.) In your own code, execution may continue in another object.

Figure 18.9

The debugging toolbar. The execution icons are only active while single-stepping after a breakpoint.

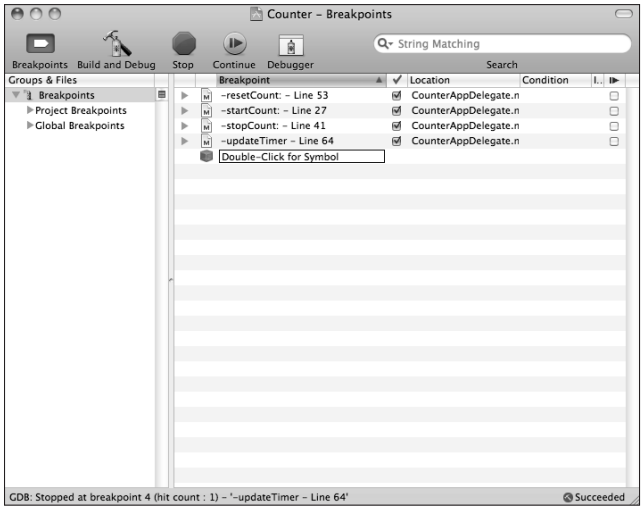
Setting conditional breakpoints

There are two ways to set conditional breakpoints. The first is to define them in the Breakpoints window, shown in Figure 18.10. There is no menu option for this window in Xcode. It can only be opened by pressing ⌘+Alt+B on the keyboard.

If you're familiar with gdb-compatible conditionals, you can double-click under the Condition tab to enter them next to a selected breakpoint. In practice, this approach may be too clumsy to be useful. gdb's conditional parser is limited, and conditional statements that test properties can become very complex. It's often easier to use an alternative approach: add the conditional tests as temporary test code and bury a conventional static breakpoint inside them.

Figure 18.10

The Breakpoints window. You can selectively enable and disable breakpoints without removing them by clicking the check boxes in the second column.



Deleting breakpoints

To remove a single breakpoint, click on its line in Xcode and choose Run ➪ Manage Breakpoints ➪ Remove Breakpoint at Current Line. You can also press ⌘+\ as a keyboard shortcut.

Breakpoints tend to spread, so it can be useful to delete them all in a single operation. The simplest way to remove all breakpoints is to pause the application in the Console window and type delete breakpoints at the (gdb) prompt. Counterintuitively, this *does not* delete the breakpoints; it deactivates them. The effect is similar — they no longer do anything — but the breakpoints are still visible in the code.

To delete all breakpoints, open the Breakpoints window and press ⌘+A to select all of them. Press the Delete key to delete them. There is no menu option for this feature.

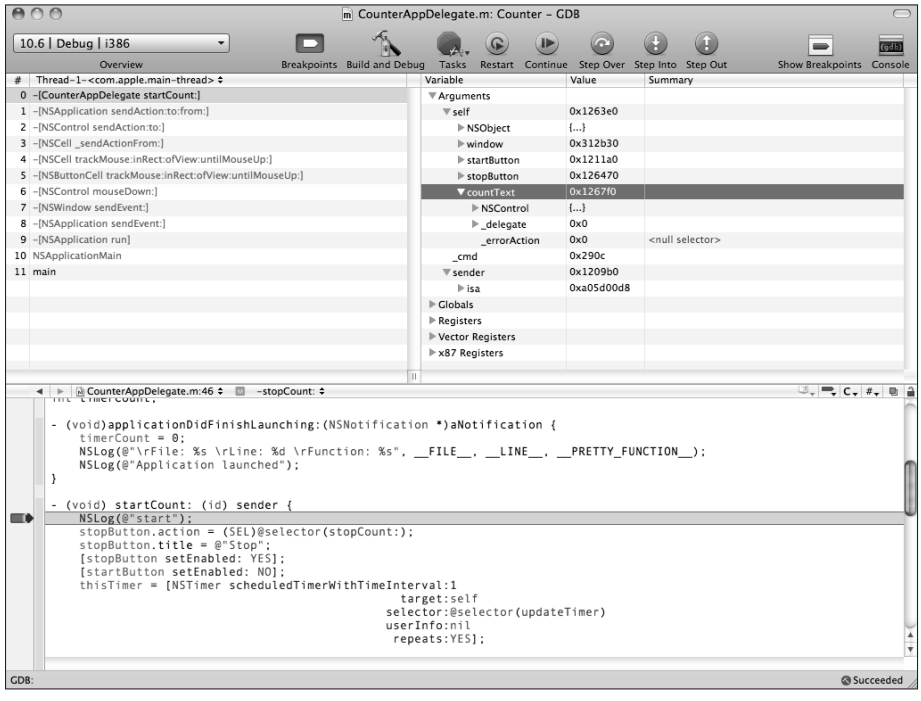
Using the Debugger window

Often, you can debug directly in the code editor pane. The floating overlays and execution options give you enough information to find and fix simple problems. The Debugger window, shown again in Figure 18.11, is useful for more advanced problems, such as tracking crashes and monitoring object properties.

You can open the Debugger window three ways. The slow way is to choose Run ➪ Debugger from the main Xcode menu. A faster way is to use the Shift+⌘+Y keyboard shortcut. The fastest and simplest way is to click the Debugger window icon in the Edit window toolbar, shown previously in Figure 18.9.

Figure 18.11

The Debugger window. The pane at the top left lists the execution stack. The pane at the top right lists every variable and data structure in the current object, and it can also list global values and the contents of the processor's registers.



Using the stack trace

The execution stack appears in a pane at the top left. The stack lists the order in which methods and functions are called. After a crash, the item at the top of the stack caused the crash. This may be — and often is — an internal Cocoa object or function.

This pane doesn't distinguish between Cocoa internals and your code. Typically most of the items in this list are internals. But you can look down the list to find the object, method, or function in your code that is likely to have caused a crash by feeding Cocoa invalid values.

Using the object explorer

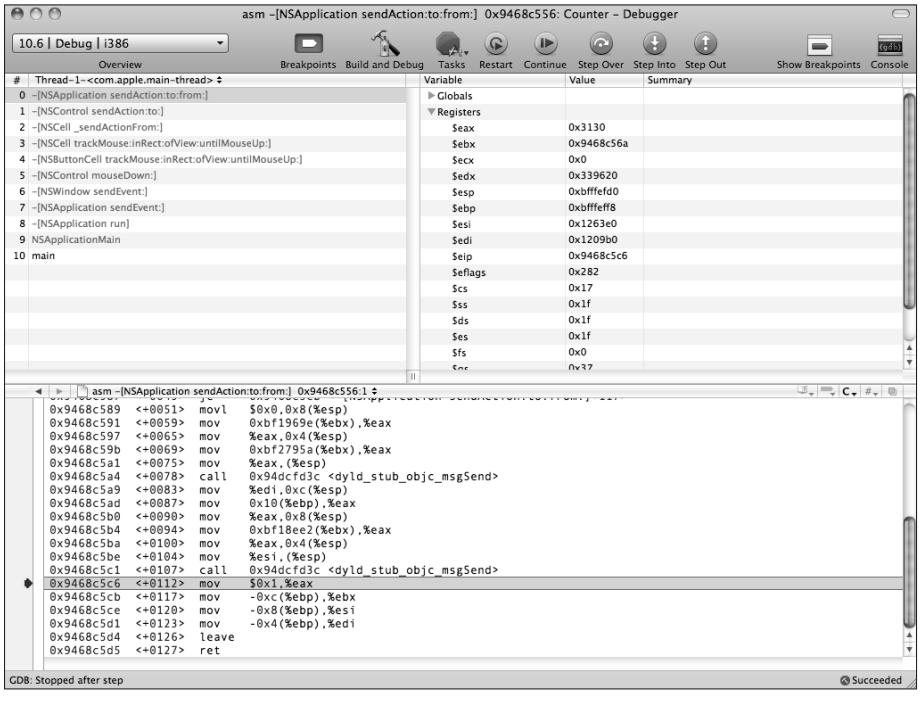
The pane at the top right lists all the values in the current object. It's an extended version of the pop-up overlays that list values, and it suffers from the same limitation — internally, objects are so complex it's difficult to find specific properties. But you can use this feature to review pointer values. It's also possible to use it to view properties, but there's no way to search for a named property. Usually you must expand the various reveal triangles by hand until you find it.

Debugging with machine code

Optionally, you can review global values and processor registers here. When you step out of a method defined in your code into OS X, the bottom window displays the compiled machine code. You can single-step through this, reviewing address literals and pointers and their associated processor registers, as shown in Figure 18.12. This is advanced debugging information and is rarely useful — unless you're writing x86 machine code.

Figure 18.12

Using the Debugger window's machine code view and register listing. Experts sometimes find this information helpful, but you can debug most projects without it.



Using Instruments

Xcode includes a suite of performance tools that can report on almost every aspect of application performance. The tools are listed separately, but they run as a single tool with multiple snap-in options. To run a tool, choose Run ➪ Run with Performance Tool in Xcode. Xcode launches your application and launches the tool at the same time, linking the two so that the tool automatically starts collecting information. Figure 18.13 shows the instruments available in OS X. Figure 18.14 shows the smaller list available to iPhone developers.

TIP

You can combine options to create a customized tool that displays any combination of performance metrics. Choose Instrument ➪ Build New Instrument. Each probe is a separate metric. Choose the metrics from the "of type" menu. Advanced developers can add customized scripting for each probe.

Figure 18.13

The instruments available to OS X developers. OS X developers can monitor threads, crashes, and multicore performance.

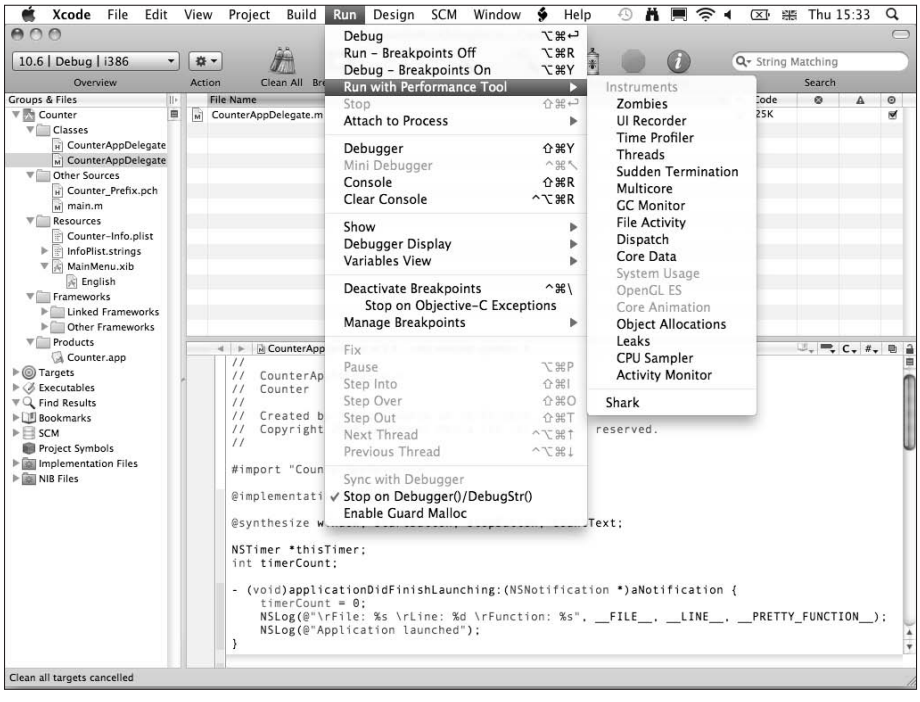
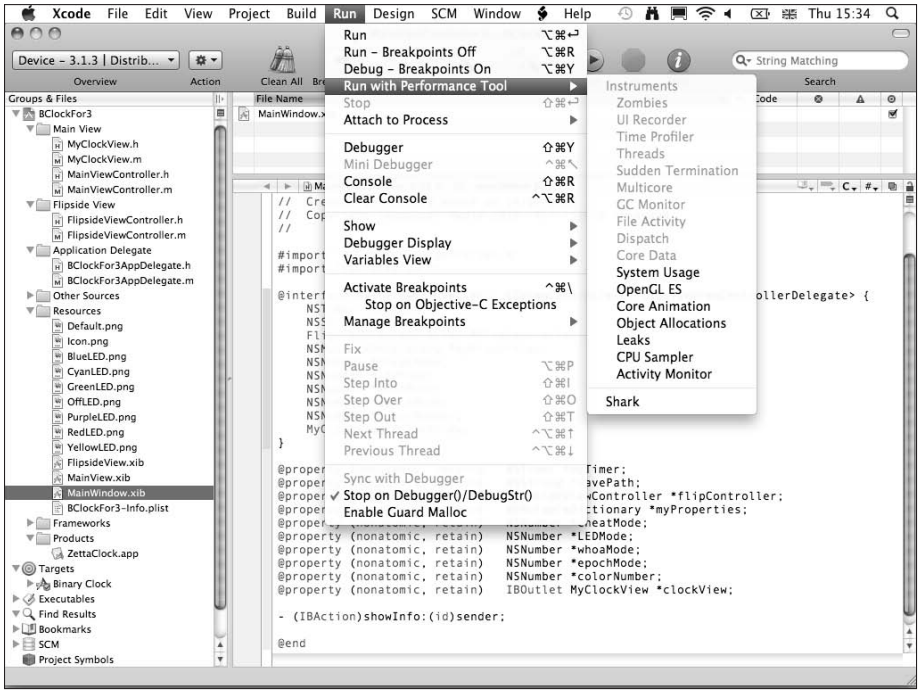


Figure 18.14

The instruments available to iPhone OS developers. Both platforms support Leaks, Object Allocations, the CPU Sampler, and the Activity Monitor. Both also support Shark, which is an independent performance profiler.



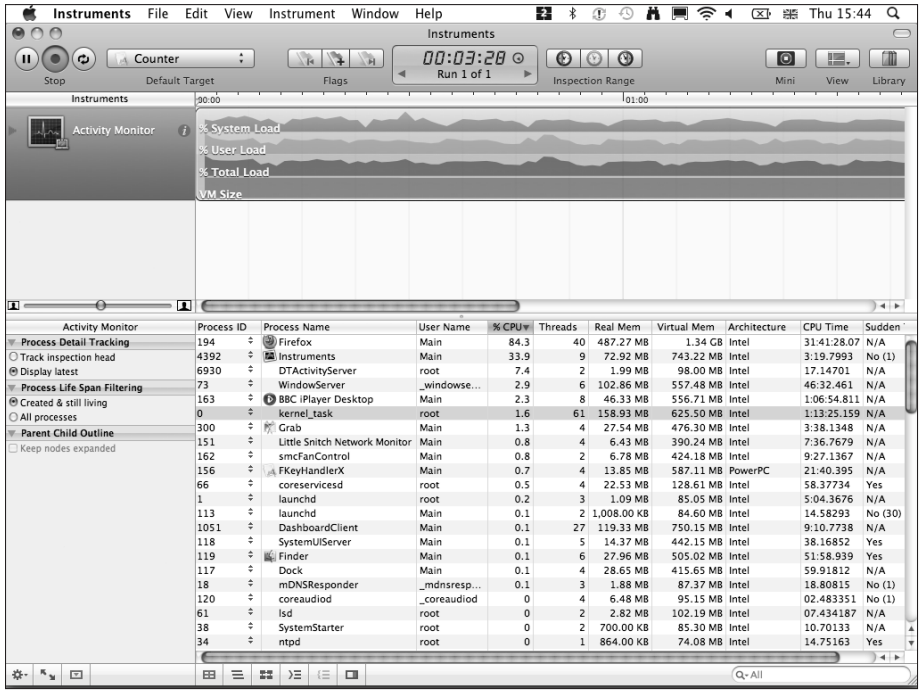
The generic Instruments tool runs as an event and value recorder. The top lane, or lanes, display a time trace of the metrics that are being monitored. Figure 18.15 shows a run of the Activity Monitor instrument, which tracks the performance and resource requirements of every process on a Mac.

All instruments use the same principle: the traces at the top of the window are graphed as the application runs. At the same time, the area under the trace lists numerical estimates of each feature.

You can save each run by choosing File➪Save. You can't replay a timeline, but you can drag the cursor along it manually to review metric values at specific times. This is only occasionally useful. Typically, you look at metrics in real time and watch how they develop as you interact with the application.

Figure 18.15

The Activity Monitor illustrates a typical timeline view generated by the Instruments wrapper. The row of icons under the main window shows or hides extra context-dependent information, which is different for each instrument.



Process ID	Process Name	User Name	% CPU	Threads	Real Mem	Virtual Mem	Architecture	CPU Time	Sudden
194	Firefox	Main	84.3	40	487.27 MB	1.34 GB	Intel	31:41:28.07	N/A
4392	Instruments	Main	33.9	9	72.92 MB	743.22 MB	Intel	3:19.7993	No (1)
6930	DTActivityServer	root	7.4	2	1.99 MB	98.00 MB	Intel	17.14701	N/A
73	WindowServer	_windowse...	2.9	6	102.86 MB	557.48 MB	Intel	46:32.461	N/A
163	BBC iPlayer Desktop	Main	2.3	8	46.33 MB	556.71 MB	Intel	1:06:54.811	N/A
0	kernel_task	root	1.6	61	158.93 MB	625.50 MB	Intel	1:13:25.159	N/A
300	Grab	Main	1.3	4	27.54 MB	476.30 MB	Intel	3:38.1348	N/A
151	Little Snitch Network Monitor	Main	0.8	4	6.43 MB	390.24 MB	Intel	7:36.7679	N/A
162	smcFanControl	Main	0.8	2	6.78 MB	424.18 MB	Intel	9:27.1367	N/A
156	FKeyHandlerX	Main	0.7	4	13.85 MB	587.11 MB	PowerPC	21:40.395	N/A
66	coreservicesd	root	0.5	4	22.53 MB	128.61 MB	Intel	58.37734	Yes
1	launchd	root	0.2	3	1.09 MB	85.05 MB	Intel	5:04.3676	N/A
113	launchd	Main	0.1	2	1,008.00 KB	84.60 MB	Intel	14.58293	No (30)
1051	DashboardClient	Main	0.1	27	119.33 MB	750.15 MB	Intel	9:10.7738	N/A
118	SystemUIServer	Main	0.1	5	14.37 MB	442.15 MB	Intel	38.16852	Yes
119	Finder	Main	0.1	6	27.96 MB	505.02 MB	Intel	51:58.939	Yes
117	Dock	Main	0.1	4	28.65 MB	415.65 MB	Intel	59.91812	N/A
18	mDNSResponder	_mdnsresp...	0.1	3	1.88 MB	87.37 MB	Intel	18.80815	No (1)
120	coreaudiod	_coreaudiod	0	4	6.48 MB	95.15 MB	Intel	02.483351	No (1)
61	lsd	root	0	2	2.82 MB	102.19 MB	Intel	07.434187	N/A
38	SystemStarter	root	0	2	700.00 KB	85.30 MB	Intel	10.70133	N/A
34	ntpd	root	0	1	864.00 KB	74.08 MB	Intel	14.75163	Yes

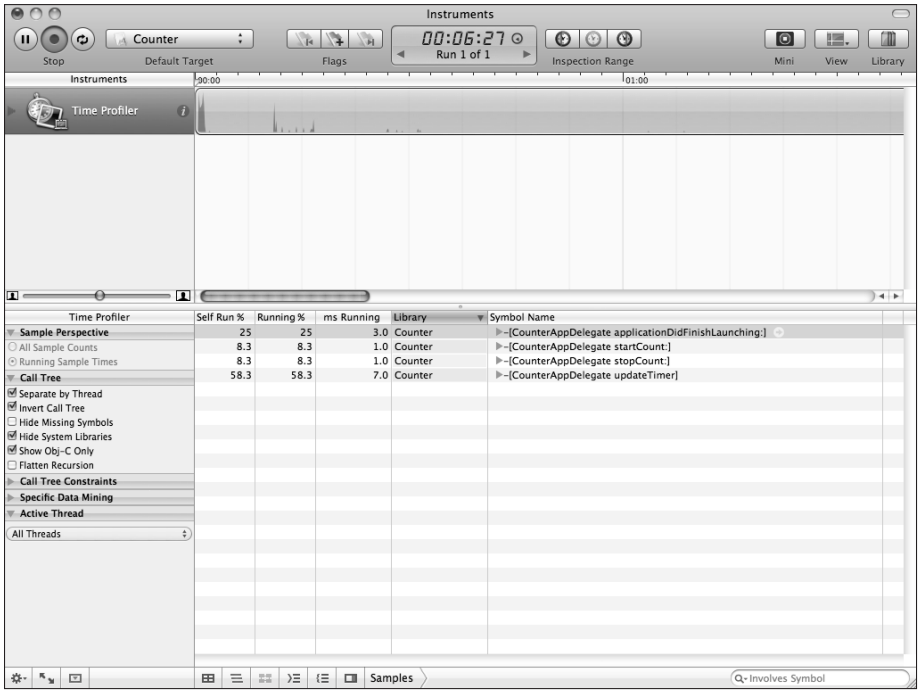
There are many instruments, and some are more useful than others. On the Mac, the Time Profiler profiles the performance of each active element in the application and displays the total processor load. By default, this instrument shows every function and method in an application, including those that are internal to OS X. To filter the list to view the methods in your code, select the Hide System Libraries and Show Obj-C Only check boxes in the Call Tree list at the left, as shown in Figure 18.16. Other useful instruments are introduced in Table 18.2.

Table 18.2 Useful OS X InStruments

Instrument	*Used for*
Time Profiler	Checking method performance
Sudden Termination	Reviewing crash dumps
Multicore (Includes Threads)	Reviewing active threads and multicore despatch
File Activity	Reviewing all file operations and file states (Note: Shows the full system-wide list)

Figure 18.16

Most instruments include filtering features. For the Time Profiler, the Call Tree options at the left can cut down a very long and cluttered list of OS X internals and force the instrument to show only the methods in your code.

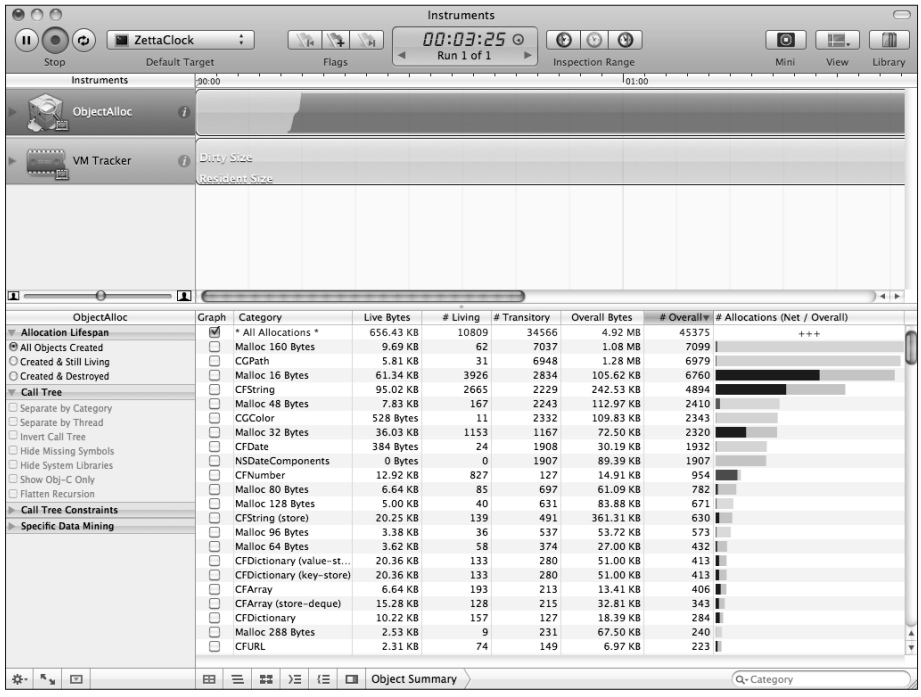


On the iPhone, memory management is one of the core problems, and the most useful instruments report object allocations and memory leaks. Object Allocations, shown in Figure 18.17, is perhaps the single most useful instrument for iPhone development. It displays a list of all allocated objects and the application's total memory footprint.

To check for leaks, view the Live Bytes total for All Allocations. If it increases steadily as the application runs, or if it increases and doesn't return to its original value after a user action, your application is leaking memory and will eventually crash. To find a leak, view the list of items under the # Living column. If this number is increasing, the object or function is leaking memory. Some leaks are built into iPhone OS and can't be resolved. But this instrument can give you instant feedback about leaks in your code, saving you hours of development time.

Figure 18.17

On the iPhone, Object Allocations is often the most useful instrument of all. It displays the total memory used and breaks it down by object and function.



Using Shark

Shark, shown in Figure 18.18, is an alternative profiling tool available for both OS X and iPhone OS development. To use Shark effectively, close all open applications, including Xcode. Run your application independently by double-clicking its app file in its build folder. You can find Shark in the Applications Directory in the Developer Folder. Launch it by double-clicking Shark.app.

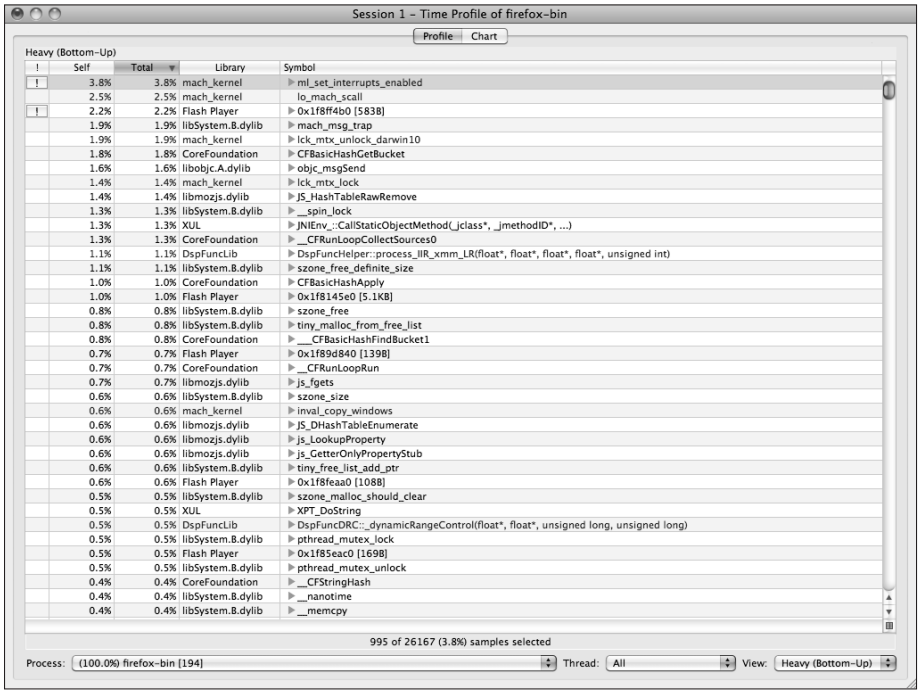
Then, you wait. Shark takes a while to load and appears to do nothing for 30 seconds. During this time, it samples the selected application. Then it analyzes the samples. This can take a few seconds for a simple application, or a few minutes for a complex application with many threads and features. A progress indicator at the bottom right indicates the status of the analysis. When the analysis completes, Shark tables the relative time the application spends in each of the listed methods and functions.

Figure 18.18 shows Shark profiling the performance of the Firefox Web browser. Shark doesn't include filters that distinguish between custom code and processes that are internal to OS X, so you must pick through this list by hand.

Shark works best for simple projects with relatively small numbers of competing processes. As the process and event count increases, the profiling becomes diluted and less useful.

Figure 18.18

Shark versus Firefox. Shark is only sporadically useful. In this example, Firefox was suffering from processor spiking, but it's not easy to find the culprit from this trace.



Managing Code with Snapshots and Source Control

It's often useful to create multiple versions of a project while debugging. It's easier to remove temporary debugging code by reverting to an older version than by manually removing the additions.

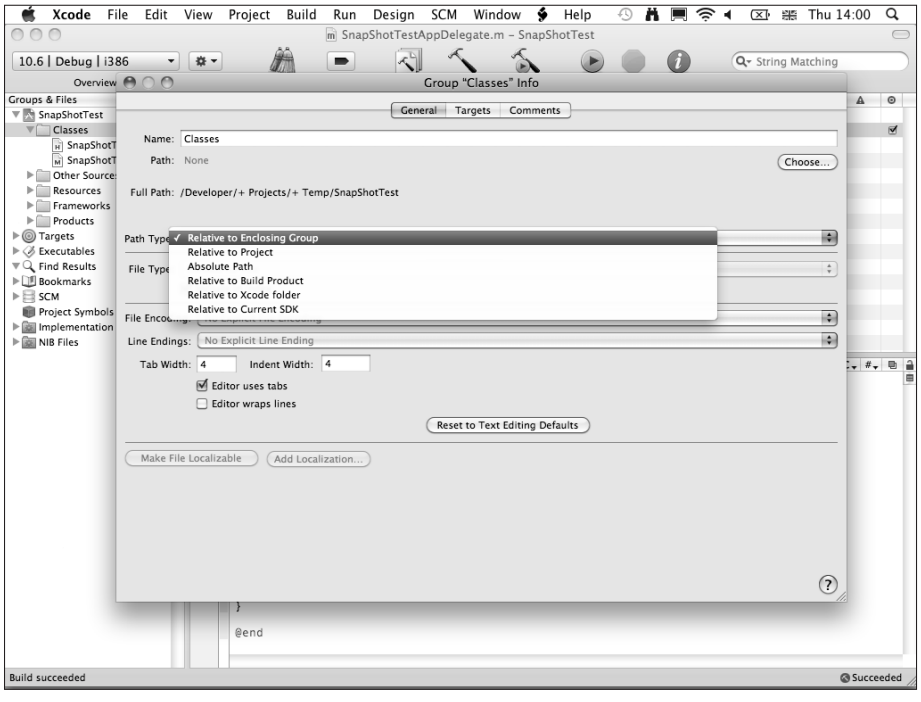
Xcode supports two source control features. *Snapshots* provide a simple and quick way to save and restore project states. *Source Control* provides a much more sophisticated toolset for managing projects, collaborating on projects with other developers, logging changes, comparing files, and so on. You can also copy project folders manually by using Finder.

Copying projects and creating snapshot versions manually

As explained in Chapter 4, it's not always possible to copy or move a default project without breaking it. Figure 18.19 illustrates why. Right-click any group in the Groups & Files pane and select Get Info and the General tab. You'll see the dialog in Figure 18.19.

Figure 18.19

Selecting group path types. To create a portable project, every element in the project must have a path type relative to the project or to the enclosing group.



Xcode's symbolic links are defined relative to a path anchor. The path type defines the path anchor, which is one of the following:

- The enclosing group
- The project folder
- An absolute filepath
- The a build product path
- The Xcode folder
- The current SDK folder

Some options are relative, while others are absolute. If you copy or move a project with absolute paths, the symbolic links will point to items that are no longer available.

Conversely, when you set up project path types correctly, you can

- Make the project available for download on a Web site.
- Use Finder to make snapshot copies of each version as the project evolves, while keeping old versions as a backup by duplicating the project folder and giving it a new name.
- Make backup copies to a different disk or different Mac. You can also build them in place, if you need to.

The safest option is to define every path relative to the project folder or to an enclosing group. There is no easy way to do this globally in Xcode. The most efficient solution is to select every item in a group and use the dialog in Figure 18.19 to change the path type. Repeat this for every group. The group "folders" have an equivalent path specification, so you must set their path types, too.

When you create a new item or add an existing project, remember to set the path specification in the drop-down menu in the equivalent dialog, shown in Figure 18.20. You *must* do this because, by default, Xcode makes all paths relative to the Xcode folder — which is wrong.

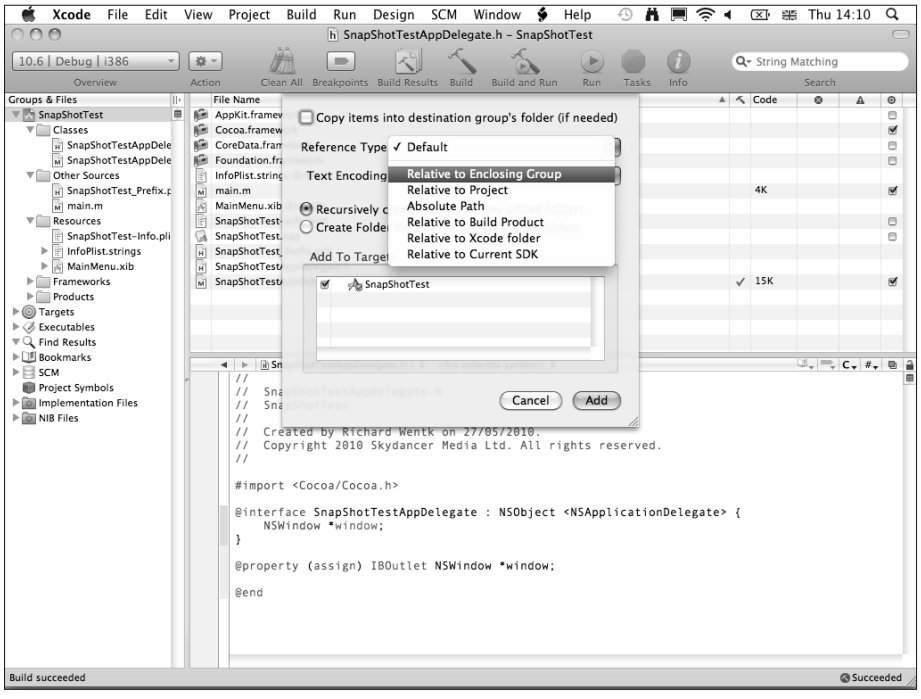
Forgetting to change this setting can literally have disastrous consequences if you copy or move a project folder. Xcode allows you to edit the file in an old project while ignoring the new copy. This can trash an old project as you make no progress on a new one.

CAUTION

Xcode sometimes gets its links confused, creating a perplexing bug. Edits to a modified file appear to be ignored, while the symbolic links appear to be correct. Typically the compiler can't find new object and method definitions, even when they're in the code. When this happens, check that the links aren't broken and that Xcode isn't editing a file in a different folder or project.

Figure 18.20

Don't forget to select the group path types when creating or adding files. The default anchors items to a fixed path, which is not a helpful feature.



Using Snapshots

Xcode includes an automated Snapshots tool for simple version management. The tool is simple but effective, and it is very easy to use. To experiment with it, create a new blank project in Xcode. Give it any name. Choose File➪Make Snapshot to save the initial state of the project.

In the code editor, add one or more lines of new code; for example, a new `NSLog` message. Make another snapshot to save the new state of the project. Choose File➪Snapshots to open the Snapshots window, shown in Figure 18.21.

Figure 18.21

A first look at the Snapshots window

You can now do three things:

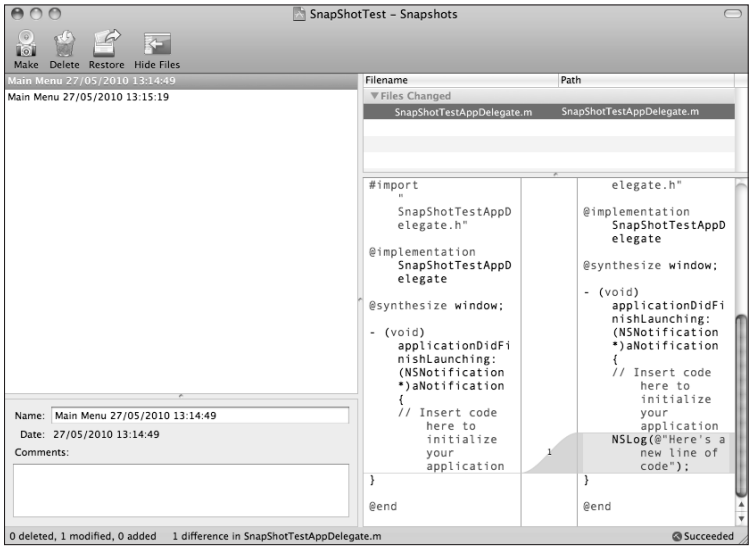
- To restore the previous version, select the first snapshot and click Restore. The original source replaces your changes. A new pre-restore snapshot is added automatically in case you change your mind.
- To delete a snapshot, click the Delete button.
- You can use the Show Files feature to review the differences between files.

To review the changed files, click the Show Files button and select a file from the list. In this simple example, you'll see a single file. A more complex project may list multiple files. You'll see the window shown in Figure 18.22. It literally highlights the differences between the two snapshots.

The Snapshots feature doesn't make copies of the project. Instead, it saves file *differences* in a separate location. Occasionally Xcode loses this location and Snapshots reports an error, complaining that it can't parse the project file.

Figure 18.22

Selecting Show Files and selecting a file displays the differences in the code between the two snapshots. This feature also lists differences in the project's support files. Support files are more difficult to understand — they're not designed to be read as text — but the differences are still clearly visible.



You can rescue the snapshots file by looking in

```
~/Library/Application Support/Developer
/Shared/SnapshotRepository.sparseimage
```

Close Xcode and mount the image file in Disk Utility. The Snapshots should now work again.

NOTE

As usual in OS X, the ~ character expands to `Users/<yourUserName/`.

Using SVN source control

Xcode supports a full implementation of Subversion (SVN) source control. You can run `svn` from the command line in Terminal to check out projects manually, and you can also integrate it into Xcode to allow check outs, edits, and commits to support versioning and collaborative development.

Source control is a complex topic with many variations and advanced requirements. The details are beyond the scope of this book. It's also not essential for solo development; Snapshots and manual copying are enough to manage most solo projects. SVN source control becomes obligatory on collaborative projects where source code is hosted and shared on a remote server. For an introduction, see Apple's documentation at `http://developer.apple.com/mac/articles/server/subversionwithxcode3.html`.

Summary

In this chapter, you were introduced to Xcode's debugging, profiling, and code management features. You learned how to use `NSLog` to report method triggers and property values and how to remove `NSLog` calls from release code to maintain its performance.

Next, you explored Xcode's breakpoint features, and you discovered how to use them in the code editor window and how to explore further features in a separate Debugger window.

Instruments were introduced in the next section, and you learned how to run code with instruments, and which instruments were most useful for OS X and iPhone development.

Finally, you explored the Snapshots and manual code backups, and you were introduced briefly to Subversion, a formal online source control tool supported in Xcode.

19 Developing for the iPhone and iPad

If you can create a Cocoa app, you can develop an iPhone app — and if you can develop an iPhone app, you can sell it in the App Store. Making the transition from Cocoa code on OS X to the iPhone's Cocoa Touch technology is easy, but the business model, the development environment, and the design goals are different. To create a successful app, you must change your aims and learn to work with the smaller screen and touch interface shown in Figure 19.1.

Figure 19.1

The famous iPhone home screen, showing some of the hundreds of thousands of apps that are available

19 In This Chapter

Introducing the iPhone, iPod touch, and iPad

Moving to iOS from OS X

Understanding iOS views and UI designs

Developing for iOS in Xcode

Building a simple application

Selling in the App Store

Introducing the iPhone, iPod touch, and iPad

Apple's mobile devices have created a popular, huge, and lucrative market for content and application developers. The three different iOS devices offer a core set of frameworks and classes for all mobile devices. All devices offer a different mix of hardware features, and there are further minor differences between different models.

The iPod touch is primarily a music and video content platform, but it can also run applications and is a popular choice for games. It is lighter and cheaper than an iPhone and can be bought and used without a contract. It remains popular with buyers and developers. Approximately 90 percent of all apps run on both the iPod touch and the iPhone.

The iPhone adds location sensing, a camera, and cellular voice and data. Unless an app uses these extra hardware features, iPod touch and iPhone apps are literally identical. Code can be compiled for a single target and runs on both platforms without changes.

There are minor but significant differences between the different models in the iPod touch and iPhone ranges. At the time of writing, the entry-level 8GB iPod touch is equivalent in power and performance to an iPhone 3G. The 16GB and 32GB models are equivalent to the iPhone 3GS and include a faster processor and slightly more advanced graphics acceleration. The performance differences are obvious in games and processor-intensive music and media apps. They are almost invisible in simpler apps. There is as yet no direct iPod equivalent of the iPhone 4.

The iPad, shown in Figure 19.2, was added to the range at the beginning of 2010. It has been criticized as a giant iPod touch, but the faster processor, larger screen, and extended operating system make it a distinct new product. On the iPhone and iPod touch, you develop user interfaces using the *UIKit framework* — a collection of fixed prewritten classes that can be added to views. On the iPad, UIKit has been extended with new UI classes that can improve the user experience on the larger screen. Some of the iPad UI features can be customized and extended to create more complex user experiences; for example, you can extend the standard pop-up keyboard with custom symbols and options.

For backward compatibility, the iPad includes an emulation window that runs existing iPhone apps without changes. It includes a 1.5X zoom option that can expand an app to fill the iPad's screen. This is a cosmetic effect. It does not change the original native resolution or make it possible for the app to use any of the OS extensions available on the iPad.

Apps can be *universal,* running the same code on both platforms with dual nib files. However, because of the physical differences between the iPhone and iPad, the different UI options, and the different markets and design goals, users often prefer distinct apps that capitalize on the strengths of each platform. It's possible to convert an iPhone project to an iPad project in Xcode with a couple of mouse clicks, using the original code as the foundation for a more complex app with an expanded UI and extra features.

Figure 19.2

The iPad introducing itself in the Safari browser

TIP

iOS applications on all platforms typically have little in common with OS X applications and are unlikely to share code. The most successful iPhone and iPad apps are developed from scratch to satisfy new user needs, creating experiences that highlight the strengths and unique features of these smaller platforms, while downplaying their weaknesses. If you are moving to iOS from OS X, think small, clever, and beautiful.

Comparing iOS and OS X applications

iPhone apps are written in Objective-C and use the familiar Cocoa framework model. However, there are significant differences between apps for each platform.

- **iOS apps are smaller and simpler than OS X applications.** They are more self-contained, with limited access to external data.
- **User interaction is managed through simpler, leaner views built from the UKIt framework.** User interfaces are smaller and less complex.

- **Many frameworks, features. and data controllers in OS X are not available in iOS.** For example, there is no way to create bindings in Interface Builder (IB). Core Data remains available, but the iOS environment is not ideal for large volumes of data or complex databases. Custom graphics are created with the Quartz 2D framework. Cocoa's graphics classes are not available.
- **Support for extra hardware features — including the multi-touch interface and sensing of tilt, shake, and rotation — has been added to create a more tactile and intuitive experience.**

Because of the differences, iOS can feel constrained. However, the limitations make it easier to concentrate on creating a strong and tightly focused user experience. A typical app is not overloaded with features but does one or two simple things with style. Apps should be as intuitive as possible. Help and instructions should be unnecessary or minimized. Apps that use clever or innovative software technologies are often popular and are more likely to be successful.

In spite of the limitations, iOS is not a crippled version of OS X. It is more useful to think of it as a spin-off, designed to create a different experience for users in a different computing context. For example, apps with location features, illustrated in Figure 19.3, can give users information related to their surroundings. This can include information about shops and local transport, routes to other locations, or new kinds of games.

iOS includes some new features that are not — yet — standard on desktop and laptop computers, including:

- **Location sensing.** All devices include a framework called Core Location that returns position updates. The iPhone 3G, 3GS, and 4 and the iPad include a GPS (Global Positioning System) receiver that can provide very accurate position information. Other devices use a combination of cellular triangulation and WiFi transmitter tagging to report much less accurate position fixes. Location data can be used to offer users information about their surroundings, with optional enhanced reality features. The iPhone 4 and 3GS and the iPad also include a magnetic compass.
- **Multi-touch screen control.** All devices support a multi-touch screen that can track at least five fingers simultaneously. This allows gestural control, as described next, but also makes it possible for apps to link features to other kinds of touch tracking; for example, three simultaneous touches can trigger a unique response, or two touches followed by a third can implement a unique drag option. Applications can use the multi-touch screen to create unique and novel touch interfaces.
- **Gestural control.** iOS 3.2 includes a new `UIGestureRecognizer` class, with predefined subclasses that can recognize taps, pinches, stretches, pans and drags, swipes, rotations, and a touch-and-hold gesture, known as a *long press*. Action methods are triggered as each gesture is recognized. Although custom gestures such as circle, check mark, or spiral movements can be recognized with extra code, users expect at least some of the standard `UIGestureRecognizer` events to be implemented. Unlike custom multi-touch support, gestural control is standardized.
- **Orientation sensing.** iOS includes hardware and messages that can monitor device orientation. As the user rotates the device, apps can switch automatically between portrait and landscape views. Orientation messages can also be used to trigger other types of responses, such as sounds, page updates, view swaps, or game events.

Figure 19.3

Built-in mapping and location sensing make it possible to create iPhone apps that tell users where they are, where they've been, or where they're going.

- **Tilt and motion sensing.** An accelerometer is built into all iOS devices. It controls orientation sensing, and data from it can also be read directly to monitor tilt and motion events. Tilt information reports the angle each device axis makes with the pull of gravity. Tilt control is popular in games, but it can be used in more creative ways; for example, to create a source of random noise source from the least significant bits of the sensor stream. Motion sensing can also be used to report separate shake events. When combined with the compass, where available, applications can use tilt measurements to calculate absolute device orientation with respect to magnetic North. iPhone 4 models include a gyroscope, which makes it possible for applications to monitor six axes of orientation and rotation.

- **Mobile camera input.** Camera support is patchy on iOS devices. The iPhone 3G has a 2MP low resolution still camera. The 3GS camera features improved 3MP resolution and supports video. The iPhone 4 camera is improved further with 5MP resolution and a built-in LED flash. The iPad has no camera at all, but supports external camera hardware. Where available, developers have used the camera in inventive ways, taking full advantage of its portability. Sample applications include barcode scanning, text recognition, font recognition, and even the scanning and re-creation of Sudoku puzzles and crosswords.
- **Built-in applications.** Some or all of the data and features included in the Maps, iPod, Contacts, and Settings applications can be accessed via APIs built into Cocoa Touch. You can build customized versions of the applications into your apps, adding new features as needed or accessing their data sources. Currently there are no APIs for the Weather, Stocks, or Calculator apps.
- **App switching.** Your application can terminate itself and launch Web-based or online applications including Safari, YouTube, the App Store, SMS, and Email by calling on a URL-based interface feature that allows selected and limited data transfer; for example, your app can launch Safari with a specific URL or preload the Mail app with a specific address and e-mail text. Apps can also launch other apps, with limited data sharing.
- **Multitasking.** iOS 4 supports multitasking on the iPhone 4 and 3GS, the iPod touch 16/32GB, and — from November 2010 — the iPad. Multitasking is largely automatic. When the user launches a new app, the existing app is placed in a low-priority background queue. This supports fast start-up, but queued apps must save their state before disappearing, because they can be removed from the queue without notice. Other requirements may apply — for example, an app can specify that it is playing audio in the background, and that it requires the resources needed to do this without stuttering or quitting suddenly.

Understanding the mobile app business model

Many apps are developed for profit and sold through the App Store. Because the app market is large and the development cycle can be very short, it is much easier to become a full-time independent iPhone developer than it is to create a career as a solo Mac developer. However, the commercial benefits of iPhone development are sometimes exaggerated. Some developers do exceptionally well, and many do fairly well, but a large number of developers get little or no income from their apps.

There is no simple formula for a top-selling app, although games remain consistently popular. For other apps, impressive graphic design, a sense of humor and lateral thinking, and strong promotion and marketing are as important as coding skills.

Making money from iPhone and iPod touch development

The iPhone app market is vast. By Q2 2010, total sales of the iPhone and iPod touch were approximately 40 million. Owners are typically students and affluent professionals in their 20s and 30s, although some are older. Many are regular users of social networking and are comfortable using online tools to manage and enhance their social lives.

The typical entry-level iPhone app is visually sophisticated but may have very simple features. Graphic design is extremely important. A visually appealing app with limited features can be more successful than a complex app with mediocre styling.

Because iPhone apps are simple, almost anyone with minimal developer skills can create them. Developers with a background on other platforms can typically produce a saleable app within two or three months. Previous OS X and Cocoa experience can reduce this to a few weeks. The QuoteMonster app shown in Figure 19.4 was created in less than a week.

Commercial developers often aim for volume sales at low prices. iPhone apps are priced in *tiers* (standard price bands) that start at 99 cents. Although it is possible to sell apps for $999, most are priced at the lowest 99 cent and $1.99 tiers. A small minority are priced at $5 or more. A handful of niche apps are much more expensive but are aimed at unusually affluent users and typically offer very obvious professional benefits.

CAUTION

Apple divides App Store income into seven *territories*: the United States and Latin America; the European Union, including Switzerland and Norway; the United Kingdom; Japan; Australia and New Zealand; and All Remaining. Apple pays developers separately for each territory. If income does not cross a $150 threshold for a territory during an accounting period, Apple retains the income until total future sales cross the threshold. If the threshold isn't crossed within an accounting year, Apple pays the developer in full at the end of the year.

Although the iPhone market continues to grow, the app market has started to saturate, making it more difficult for newcomers to profit with relatively simple projects. Popular games, such as Trism, shown in Figure 19.5, have been developed by part-time coders. But more recent game apps are becoming more complex and moving beyond the reach of solo developers. Niche apps with minimal marketing support may sell just a handful of copies.

Games and simple novelty applications with mass appeal offer the best chances of success. More complex apps with unique features are riskier: some will trigger a wave of user interest, but many will not. Marketing skills are as important as programming talent. Getting an app accepted by the App Store is not enough. Business-minded developers must spend time on viral and direct advertising, using social networking tools such as Facebook, Twitter, and MySpace.

Figure 19.4

A typical iPhone app is closer to a dashboard widget than a full OS X app and has a simple function controlled by a minimal streamlined interface. Once a developer has mastered the essentials of app design, the development cycle can become very fast.

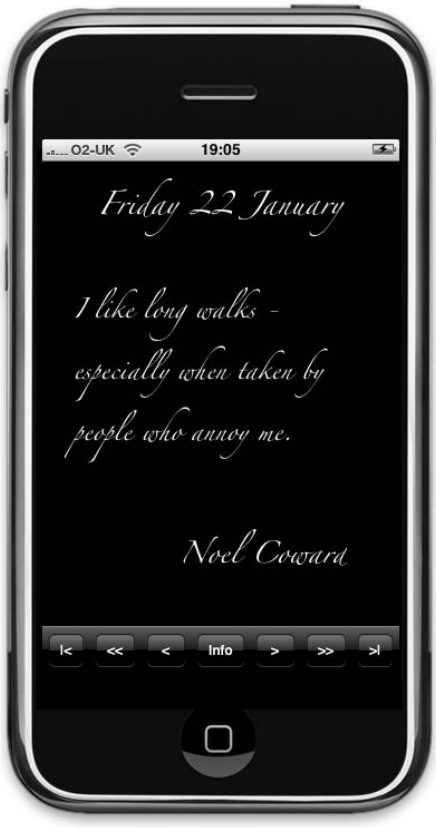


Figure 19.5

The Trism game app earned more than $250,000 in its first two months in the App Store, but it may be more difficult to repeat this success now.

Some developers use ad revenue as a separate income stream. Services such as Medialets (`www.medialets.com`) allow developers to dedicate screen space to an external ad server that pays them automatically for each ad click.

With iOS 4, Apple introduced its own competing ad service called iAd with a supporting framework. Ad revenue can provide a useful income stream from simple entertainment or novelty apps that can be given away for free on the iPhone and the iPad.

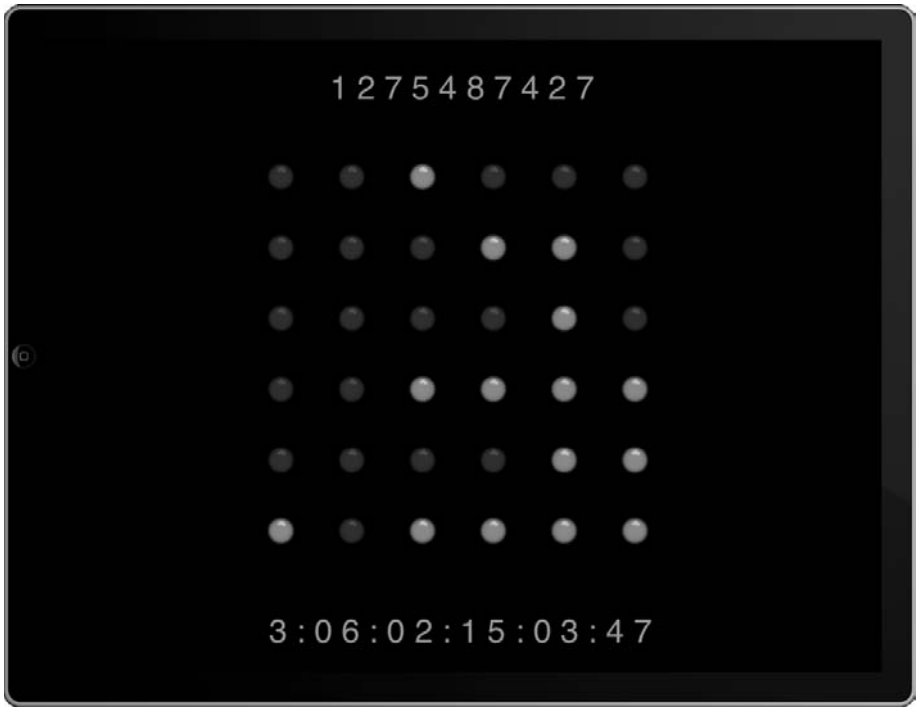
Making money from iPad development

The iPhone and iPad share a similar programming model, but they have a very different business model. The iPad market is smaller — total sales are expected to be around 6 million by the end of 2010, rising to 10 to12 million by the end of 2011 — and the user profile is different, split between the traditional younger iPhone demographic and the older less-experienced computer users who are looking for a simplified experience of e-mail, the Web, games, and other entertainment computing.

Because the iPad uses a faster processor and bigger screen, it can be tempting to keep adding features to apps, extending the development cycle. But the app market is still relatively small, and extra sophistication may not mean more user interest. Niche apps with complex specialized features are likely to have a tiny potential audience with even tinier sales. Longer development times can justify higher app prices, but total sales are likely to remain smaller. Figure 19.6 shows a binary clock application that uses the iPad's larger screen to create a simple app that can be left running permanently, converting the iPad into a domestic accessory.

Figure 19.6

This minimalist binary time display app relies on the large physical size of the iPad. It is designed to run when the iPad is in a cradle or stand, creating an unusual desk clock.



One way to avoid niche limits is by developing games; the iPad has obvious appeal as a handheld gaming platform. It can also be inspiring to think of the iPad more as a content reader and a player for entertainment and education. Content container apps can be developed quickly, easily, and cheaply. The iPad is ideally suited for displaying Web content, Web apps, and subscription services. For a content delivery app, you can use Web technologies — you may not need to use Xcode at all.

A subscription or content purchase scheme can create regular income in ways that apps cannot, and simple Web apps with compelling regular content updates are likely to be at least as successful as traditional iPhone-style apps. Traditional apps can still sell well, but for the moment the iPad is a riskier commercial prospect for solo developers than the iPhone, although success can be more rewarding.

TIP

You can use the Top Charts feature in the App Store to find the most popular and successful apps. Checking the charts regularly and testing the top free and paid-for apps can give you useful insights into the design and features of best-selling apps.

Moving to iOS from OS X

iOS is recognizably similar to OS X. Many of the design patterns are identical. iPhone apps use Model-View-Controller, target/action, key value coding and observing, blocks and threads, the responder chain, and notifications in an identical way. The nib system is also similar, with some simplifications and omissions.

But iPhone apps and Mac applications have different design goals and run on different hardware. Because of the hardware differences and the pared-down classes in iOS, the development environment is less complex.

Getting started with the iPhone SDK

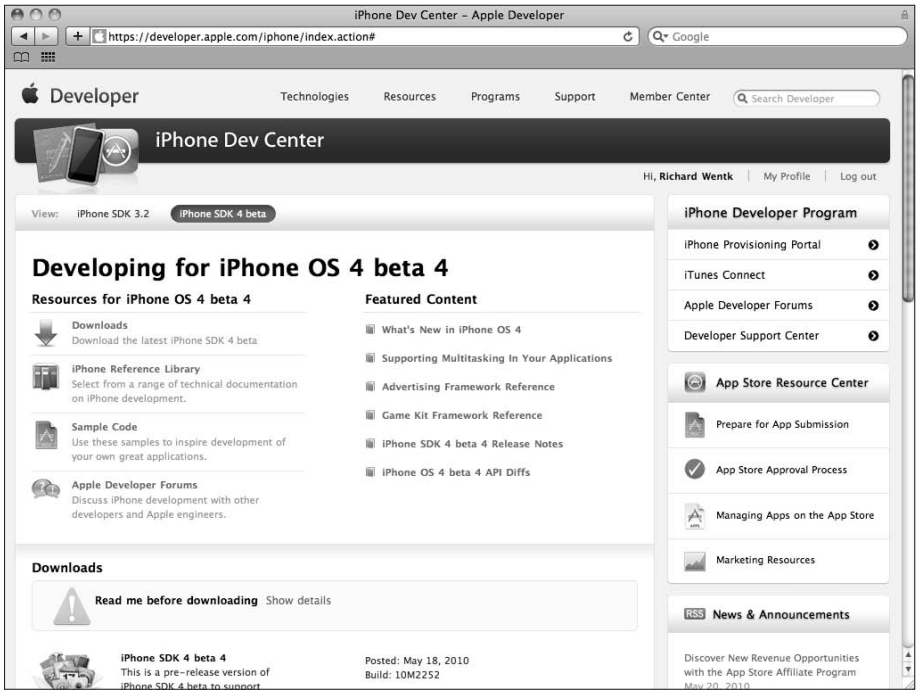
The iPhone has its own developer center and associated developer program, shown in Figure 19.7. From Q1 2010, iOS and OS X developers use the same SDK, which supports both platforms. Both programs are accessed through separate pages on Apple's site, and both offer free and paid-for access with enhanced support and other features.

CROSS-REF

For more information about signing up to the iPhone Developer Program and downloading and installing the Xcode SDK, see Chapter 4.

Figure 19.7

As with the Mac Developer Program, signing up for paid access to the iPhone Developer Program allows developers to download beta seed versions of forthcoming OS updates.



Understanding iOS app design goals

Similarly, the user experience is also streamlined and simplified. Key design goals include:

- **Instant on, instant off.** iOS apps should load and quit as quickly as possible. Long load times are strongly discouraged. Long shut-down times are not allowed at all. Apps that take more than a couple of seconds to save and quit may be terminated by the operating system.
- **Full-screen operation.** Apps do not share the screen with other apps and do not support floating windows. This restriction is rigid on the iPhone but has been relaxed slightly on the iPad, which allows modal and partial views that can be made to pop up or appear attached to other views.
- **Loss of focus without notice.** On the iPhone, incoming calls force an app into the background and bring the built-in Phone application to the foreground. Apps may be paused in the middle of an operation without notice or may be terminated, such as when an SMS message arrives with a Web link and the user taps it to view it with the Safari browser.

- **Restricted battery life.** Battery life is extremely limited on the iPhone, slightly less limited on the iPod touch, and reasonably long-lasting on the iPad. iPhone applications must do everything possible to conserve power, dynamically disabling hardware when it is not needed or powering it up and powering it down again on a regular polled cycle.
- **No explicit data save and restore.** Apps should always save and reload their states automatically. Apple's design guidelines suggest that file operations should be invisible where possible. File save and load features should only be visible to users when there is no other way to manage application data.
- **Constrained data sharing.** Apps have access to their own file area, but they cannot easily share files with other apps or with a desktop computer. This restriction is relaxed slightly in iOS 3.2, but the file system can be challenging to work with.

Games and OpenGL on iOS

iOS mobile devices are an ideal platform for handheld gaming. The touch screen and tilt sensing can create a very tactile and immersive experience; for example, a driving game or space shoot 'em up can link tilt motions to steering and navigation. Complexity is not essential. Simple, easy-to-play games can be as popular and profitable as complex, animated, shared-world adventures.

Apps can use three different technologies for games. Simple puzzle games can use UIKit to handle view management, interfaces to the tilt and orientation features, touch screen support, and simple animation effects. UIKit is ideal for puzzle and grid games with simple animations, and it runs fast enough to make surprisingly complex effects possible.

For more open-ended graphics, the Quartz Core framework, which is a simplified version of the equivalent framework in OS X, is a good choice. The essential drawing and animation features and concepts are similar, but filters are not supported. 3D and 2D affine transforms, layered graphics, and keyframed animations are all available.

The most sophisticated effects are possible with OpenGL ES graphics acceleration, which is ideal for games with animated 3D environments, characters, and objects. 2D graphics are also possible, but OpenGL ES can be a challenging environment for newcomers, and the simpler frameworks are a better solution for apps with simpler requirements. Adventurous developers can use OpenGL to bypass UIKit and create completely customized animated interfaces.

OpenGL code is not identical to OpenGL ES code. OS X games that use OpenGL cannot run on iOS without changes. The iPhone's graphic acceleration hardware has limited texture memory, and apps also have strictly limited memory, so complex textures and models may need to be simplified. The advantage of OpenGL ES is cross-platform support. Other mobile platforms also use OpenGL ES, and an iPhone game can often be ported to a different platform relatively simply.

Two versions of OpenGL ES are used on iOS devices. The iPhone 3G and 8GB iPod touch support OpenGL ES 1.1. The iPhone 3GS, 16/32GB iPod touch, and the iPad support the newer and more sophisticated OpenGL ES 2.0 specification, but can also run apps that use the older technology. The 2.0 specification includes support for *software shaders* — customized dynamic texture and animation code — but is recognizably similar otherwise.

Games can use the *GameKit* framework, which implements Bluetooth discovery and data sharing across two or more devices. GameKit can be extremely rewarding: apps can share data and game events with relatively little programming effort. GameKit features can also drive sales in the App Store, because users are more likely to encourage friends and family to buy apps so they can play together; this is most likely for games that include live sharing but also include a solo mode so that combined play is not essential. Live voice chat is also supported.

Understanding key iOS coding differences

Although Objective-C code for iOS and OS X uses the same coding principles and many of the same design patterns, the programming model is significantly modified. Some differences include:

- **Different classes and frameworks.** For example, UIKit is only loosely related to Application Kit, and it offers a much smaller selection of classes.
- **Limited memory, with no swap file.** Memory is not paged in iOS. When an application receives a memory error message from the OS, there is no more memory available. It must release some or all of its memory, and it may even be shut down without notice.
- **Limited memory management.** iOS uses retain/release memory management and does not implement garbage collection. Reference counting is simple in theory but challenging in practice. Apps often need additional testing to eliminate leaks and to fix crashes caused by pointer release errors; therefore, it is not unusual for apps to be released with leaks.

Considering iOS and hardware compatibility

As iOS has evolved and the range of mobile devices has increased, incompatibilities have started to influence app design. iOS 3.1 runs exclusively on iPhone devices, but apps can also run on the iPad in a backward-compatibility mode. OS 3.2 runs exclusively on the iPad. OS 3.2 apps cannot run on an iPhone.

iOS 4 supports multitasking. Soon both the iPad and a new iPhone 4G handset will be able to run OS 4. Limited backward compatibility is supported. Older devices, including the iPhone 3G and iPod Touch 16MB, can also run OS 4. But selected features, including the multitasking, are not available.

This leaves developers with a problem: Should they try to maximize sales by creating apps that can run on older hardware, or should they create more adventurous apps with limited compatibility?

A practical answer depends on the likely user and sales profile, which will be different for every app, but the question can have a significant influence on likely sales and needs to be considered.

Understanding iOS Views and UI Design

Views and UI design are a fundamental part of every app. The iOS view classes are recognizably similar to those used in OS X. However, views are used differently, and you must learn new techniques to create user experiences that follow Apple's mobile design guidelines.

Working with Windows and views on the iPhone

An iOS includes a single instance of `UIWindow`. Multiple floating windows are not supported. The window does not include scroll bars or other screen furniture, and it remains invisible, acting as a placeholder and frame for various view elements.

Views are typically loaded as nib files, via a view controller. A view always fills the screen; views do not float or overlap. There is some support for content scrolling within views for text, Web pages, PDF files, and static images.

The iPhone screen is too narrow to support drop-down menu feature selection. Instead, views are swapped. UIKit includes a selection of preset navigation controllers that can be built into applications to allow users to trigger view swaps and other events. Swaps can be animated to create special effects, such as cross fades, pushes, spins, and overlays, as well as page curls, shown in Figure 19.8.

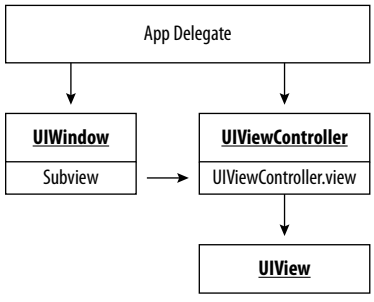
Figure 19.8

Page curls and other animation effects are built into iOS and can enhance the look and feel of any project.

Controllers that implement navigation through view swapping are typically added to the application's main window nib file. Views are swapped in and out behind them, so the controllers remain visible. More complex configurations are possible, but the standard design pattern shown in Figure 19.9 is used in many applications.

Figure 19.9

Anatomy of an iPhone application: The master/root view controller remains permanently loaded. Other view controllers can be loaded from nib files and released as needed. To make them visible, their views are swapped into the master controller's .view property.



Understanding Views, View Controllers, and nib Files

Views, view controllers, and nib files in iOS are similar to those in OS X. You can assemble view designs, link them to outlets and actions, and save them to nib files in the same way — with three differences.

The Bindings and Effects panes are not available in the iOS version of Interface Builder. You can add animation effects to your code manually, but filters are not supported and tool tips are not available. You can emulate bindings with Key Value Observing in your code, but you cannot create and change them in Interface Builder.

In Cocoa, views in windows are relatively static. When an application needs to display a new interaction area, it usually loads it into a new floating window. iOS only supports a single window on the iPhone and iPod touch, and a pair of windows on the iPad. Windows are invisible container objects. Views are swapped into these windows dynamically and unloaded when no longer needed. Managing view swaps is an essential iPhone design skill, and view swapping code is central in almost all apps.

View swaps are managed by `UIWindow`, `UIView`, and `UIViewController`. A typical app includes an instance of `UIWindow` declared as a property of the app delegate. To display a view, call the `addSubview:` method on the window, passing it a `UIView`. The view appears in the window.

continued

continued

The simplest way to manage view swaps is through a subclassed instance of `UIViewController`. Instead of passing a `UIView` directly to the window, pass it indirectly through the controller. Boilerplate code is included in the Xcode templates:

```
[window addSubview:
   aViewController.view];
[window makeKeyAndVisible];
```

To display a different view, update `aViewController.view` with a pointer to a `UIView`.

Views can be swapped semiautomatically. *Navigation controllers* and *tab bar controllers* implement standardized navigation solutions, swapping view controllers and their associated views as needed. Views can also be swapped manually with custom code managed by a view controller. Swaps can be triggered by almost any OS event. In many applications, views are swapped after a button tap. More complex effects are possible; for example, a location-aware app can display a new view when the user gets close to a geographical location. Subviews in a view are stored in an `NSArray`. `UIView` includes methods for inserting and removing views from the array and rearranging their order.

Because view swapping and controller subclassing is a key iOS design pattern, Xcode includes a convenient `UIViewController` subclass file template, with an associated nib file. To create new views in your app, create as many different subclasses of `UIViewController` as you need, and swap their views in as required. You can populate the nib files with user options — buttons, labels, graphics and contexts, text editable text fields, among others — and then add code to the view controller subclass to manage each item, linking outlets and actions as needed. For more on outlets, actions, and links, see Chapter 7.

`index:0` of the `UIView` array is the root view. To implement a manual view swap, allocate an instance of the incoming view controller and load its nib using the `initWithNibName:` method. Alternatively, you can `alloc` and `init` instances of `UIView` dynamically in your code, adding features and setting options as needed. You can then swap in the new view by calling the `insertSubview:atIndex:0` method on the currently visible view. To remove and release a view, call the `removeFromSuperview:` method on the visible view.

Optionally, you can preload the new controller with data or link it to a data source so that it automatically loads and displays the data when it appears. Data is often processed for display in the `viewDidLoad` method in the new controller, which is triggered as soon as the view loads. You can also use the `loadView` method to populate the view dynamically from your code, or by loading a nib at runtime.

Modal views pop up or slide in temporarily — for example, to display app credits or contact information — and disappear when dismissed, revealing the original view. You cannot use modal views for navigation. To display a modal view, allocate an instance of the new view controller and call `presentModalViewController:` in the main view, passing the new controller as a parameter. To dismiss the modal view, call `dismissModalViewController:` in the main view.

All view swaps can be animated. Modal view swaps offer a small selection of preset animation, controlled by a constant loaded into the `modalTransitionStyle` property of the main view controller. Non-modal view swaps can be animated with a custom animation block that selects a preset animation effect, such as a page curl or cross fade, and sets an animation duration.

Because views can be swapped, they must be able to exchange messages and data with each other. Your app can use the protocol/delegate design pattern to implement message passing, or it can make direct connections between objects in the interface and a root controller using the responder chain.

The view system can seem complex, but it does not have to be difficult to work with. To avoid confusion, remember that the instance of `UIViewController` included in the app delegate sets which view is visible. Other instances of `UIViewController` can be loaded and released from memory. They can handle events, but their views remain invisible unless they are linked into the master controller's `.view` property.

Apps typically use boilerplate code to implement view swapping. An example of this appears later in this chapter in the section on building a simple application.

Managing orientation

All mobile devices include orientation sensing. As the user rotates a device, an app can rotate the view so that the content remains vertical. Figure 19.10 shows how Safari autorotates its contents when the iPhone is horizontal.

Figure 19.10

Autorotation in Safari. Rotating the iPhone automatically rotates the browser and its contents.

On the iPhone and iPod touch, orientation support is optional. Apple suggests that the portrait upside-down orientation is not supported because users may find it confusing, particularly if the phone is upside down when a call arrives. On the iPad, apps must support every orientation.

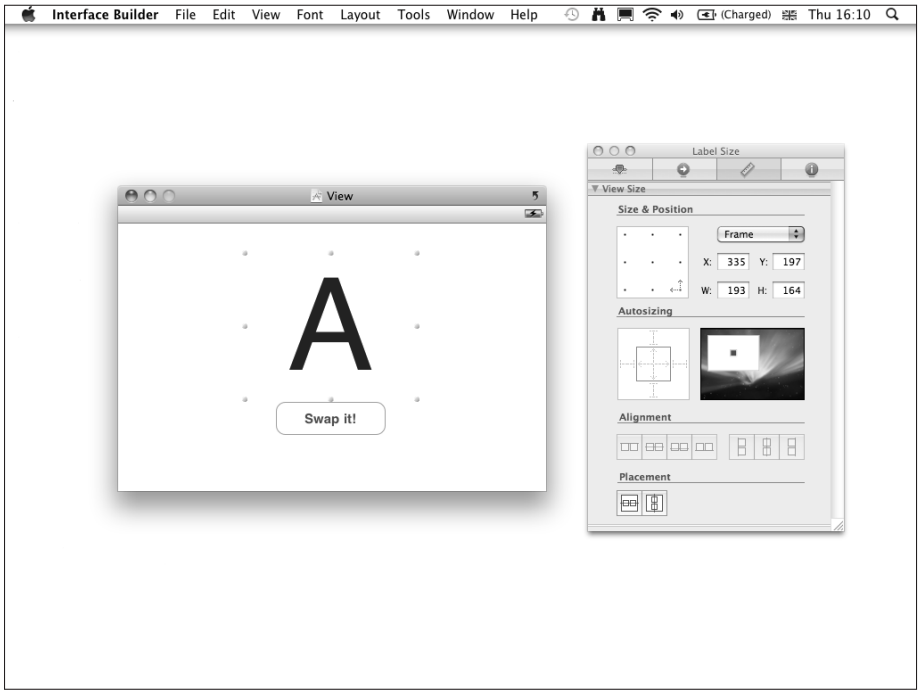
Orientation support is managed by each view controller. Simple apps can implement autorotation by including the `shouldAutorotateToInterfaceOrientation:` method, returning `YES` for each supported orientation. To allow rotation to every orientation, use:

```
- (BOOL) shouldAutorotateToInterfaceOrientation:
   (UIInterfaceOrientation) interfaceOrientation {
 return YES;
}
```

If you include this code in a view controller, iOS automatically rotates the view. Items in the view should have autosizing disabled in Interface Builder, as shown in Figure 19.11, to keep them positioned and sized correctly in the view as it rotates.

Figure 19.11

To disable autosizing, click the lines in the Autosizing box. All lines should be dashed, not solid. Use the rotation preview feature in IB to test the view before saving it.



TIP

You can preview rotations in Interface Builder. To rotate a view, click the curved arrow icon at the top right of the View window. You can use this feature to check that subviews are anchored correctly.

It is not always possible to create views that autorotate correctly; for example, items in busy views may overlap after rotation. To customize rotation effects, you can also create two or more views to support different orientations and swap them in as needed.

A `willRotateToInterfaceOrientation:` message is sent to a view controller just before rotation. For more complex effects, the rotation process can be split into two stages. A `didAnimateFirstHalfOfRotationToInterfaceOrientation:` method is triggered just before the second stage. You can place view switching code, described later in this chapter, in either method to swap views to match the new orientation.

CAUTION

Orientation handling is not completely reliable. To work around this, you can add code to monitor the accelerometer independently, creating rotation events as needed. This is not a simple project, but it can be more reliable and smoother than the standard OS approach. You can also use it to bypass the default rotation animations and replace them with alternative more complex effects, such as cross fades or spin-out/spin-in movements.

Adding navigation and control features

UIKit includes a selection of standard classes to manage in-app navigation. Apps typically include one or two of these classes. Navigation options should be clearly visible and kept as simple as possible.

Tab bars, shown in Figure 19.12, implement modal view switching. When the user taps one of the items on the bar, it automatically selects a preassigned view controller, loads its view from a nib file, and displays it. Users can customize the tab bar, adding items from a preset list, reordering them, and removing them.

You can use tab bars in a very simple way by adding a list of tab bar items to the tab bar in Interface Builder and presenting a view controller class within Interface Builder. Tapping a bar item selects and loads a view controller class, automatically loading and displaying its view from the associated nib. The view switching and class selection are built into the tab bar and do not need extra code.

Tab bars can also create more complex effects, loading other classes such as *table views,* which display lists of selectable and editable items, or *image pickers*, which are prebuilt photo selection classes that access images in a device's photo library. Optionally, tab bar items can be *badged* to display a number or very short string in a red oval to indicate items that need attention or to offer users extra information.

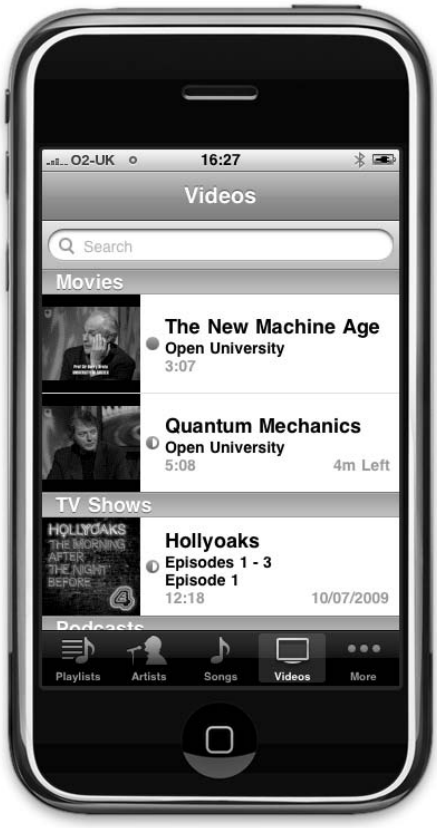
Toolbars, illustrated in Figure 19.13, offer a selection of items that can be tapped to trigger application features, such as play/pause features in a media player or general done/cancel selections. Toolbar items are independent and do not interact. They highlight momentarily when tapped, and they use target/action selection to trigger selected methods in an app.

Figure 19.12

A tab bar template is included in Xcode and demonstrates how to swap between two views without extra code.

Figure 19.13

Xcode does not include a toolbar template, but toolbars are used for navigation in many of the iPhone's built-in apps, such as the iPod app shown here.



Although toolbars can implement view switching and navigation, they are more usually used for features that create a visible response within a view without swapping it. The list of compatible bar button items includes fixed and flexible spacers that control the horizontal layout of buttons on the bar. Adding spacers automatically positions buttons on the bar so they fill the space in the most elegant way.

Both toolbars and tab bars include a small list of graphic *identifiers* — preset text and icons included to support common app features. For example, selecting the Favorites identifier adds a "Favorites" label to a tab bar item and displays a star icon.

The *navigation controller* class implements hierarchical view switching and is often used to simulate drop-down list navigation. The controller implements a navigation stack — a linked list of view controllers. As users drill down into the hierarchy, controllers are pushed onto the stack and a back button automatically appears on the controller bar. As users move back up through the hierarchy, controllers are popped and released. The back button is created and named automatically. It pops the previous controller from the stack and triggers an associated view switch when tapped. The iPad example shown in Figure 19.14 illustrates a split view with two navigation controllers: Settings and Keyboard. The Settings bar is decorative and is used as a large title bar for the combined view. The iPhone equivalent, illustrated later in Figure 19.16, displays a single controller at the top of the view.

NOTE

You must add a navigation bar to a view to implement navigation and add a Navigation Item subview to display a title on the bar. The Navigation Item includes four empty placeholders: the title view, and separate back, left, and right buttons. The back button is allocated and released automatically. You can allocate and link a right button to add extra features. The title view is a generic view and by default displays a label. You can set the label text from within your code using its title property, or give it a fixed title within Interface Builder.

Figure 19.14

The iPad's split view feature works well with two navigation bars and a table view, illustrating how navigation flows naturally from left to right and how table view cells can be customized with images, active subviews, and display groups.

To simulate a drop-down list, navigation controllers are often used with table views, which display a scrollable vertical list of options. Table views are sophisticated and have many features. When used for navigation, they are typically shown with extra chevron views to indicate that tapping on a table cell will view switches and take a user to the next level in the hierarchy. They also have other applications; cells can be displayed with thumbnail photos or icons, switches, sliders, labels, and other standard features. They can also be completely customized and loaded from a nib file. Navigation, control, and display features can be mixed in a single table.

CAUTION

The navigation controller stack manages a list of view controllers. Do not confuse it with the view array. The view array is part of the nib system and is included in every app. The navigation stack is a separate list of controllers, used to implement forward/back navigation. It is only implemented when the app includes a navigation controller.

Custom switching options include view switching through rotation events, tilt and shake events, touch events, or even sound, speech, or music recognition. In iOS, view switching features are not tied to specific classes or selection events; they can be triggered by any of the features in Cocoa Touch. In theory, similar flexibility is possible in Cocoa, but the touch screen and other hardware features built into mobile devices encourage experimentation and can create experiences that would be difficult or impossible to include in a Cocoa application.

A key feature of app design is that views and their controller classes are very closely linked and are often designed anew for each application. Each view implements a small number of cleanly defined features. The associated code in the controller class preloads a view with content from the application's model and data source and manages the user interactions that trigger each feature.

This makes it possible to develop complex applications with a small number of controller classes; for example, a mini-encyclopedia might use only two or three different view controller classes to display and navigate through hundreds of pages of content.

TIP

Custom view controller classes can proliferate uncontrollably in iPhone apps. Do not reinvent the wheel by creating a new class for every view.

Handling touch events

Instead of a mouse, iOS apps generate touch events, which are processed by an application's responder chain. From OS 3.2 onward, you can use the `UIGestureRecognizer` class to trigger actions automatically whenever the OS recognizes one of the standard gestures. This dramatically simplifies touch processing and makes it easier to include a range of touch features.

Some apps still require custom touch handling. Manual touch processing is moderately complex. Touch events trigger four methods: `touchesBegan:` when a new touch is registered; `touchesMoved:` when movement is registered; `touchesEnded:` when a touch ends; and `touchesCancelled:` when the OS deletes existing touches so that it can respond to an external event, such as an incoming phone call.

Touches arrive as an `NSSet` of touch objects, with an associated instance of `UIEvent` that can supply optional timing information. Counting the items in the `NSSet` returns the number of touches. To extra individual objects, copy the `NSSet` to an `NSArray` and read touch items at each index. Each touch can return a position coordinate relative to its view. For example:

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
  UITouch *touch = [touches anyObject];
  NSUInteger numTaps = [touch tapCount];
  CGPoint pt = [touch locationInView:self];
}
```

This sample code implements a single touch handler. The `touch` object is a placeholder used to read touch information passed to the method. `numTaps` returns the current number of taps, and `pt` is a `CGPoint` that holds the touch coordinates in the view. Equivalent code for a `touchesMoved:` implementation is identical. To track a touch, read `pt` in `touchesMoved:` and update your app as it changes. For a multi-touch handler, replace `UITouch *touch` with `NSArray *allTouches` and enumerate through the `UITouch` items in the array to read each position.

To recognize gestures manually, track touch movements and compare them with a model of the gesture. This is a simple problem; for pinch and stretch, gestures save the initial distance when `touchesBegan:` registers two touches and compare it with the current distance calculated in `touchesMoved:`. Optionally, you can use the time stamps in each `UIEvent` to track touch velocity and filter out very slow or very fast movements. More complex gestures require more complex modeling; for example, to recognize a spiral, you must save the initial angle as well as the initial distance, and track changes in both.

Working with windows and views on the iPad

The iPad OS includes all the view controller and event types available on the iPhone. Views, view controllers, user events, and touch methods are identical, but extra classes are available to enhance the larger screen. *Split views* offer two side-by-side views, with fixed widths of 320 and 704 pixels. Split views are only active in landscape orientations. In portrait orientations, only the rightmost view is visible. The split view controller automatically hides the leftmost view. Apple's UI guidelines recommend using split views for master/detail content, typically using a master list of items in a table view at the left, to select more information, and further options at the right. However, either view can be a standard nib-loaded view, with an associated controller; for example, the left view can show a game control panel, while the right view displays a play area.

Popovers can display temporary pop-up views of any size. The position is fixed — the popover cannot be dragged — but it provides a container for any type of view. Possible applications include history lists, palettes of options, high score or other tables, or zoomed previews/overviews.

CAUTION

Toolbars and tab bars do not work well with split views and popovers, especially when they are placed at the bottom of the screen. Apple's guidelines suggest that apps should not use them. However, you can add two separate navigation bars to the views in a split view layout. This improves the appearance of an app, even if the navigation bar is cosmetic and does nothing.

Developing for iOS in Xcode

The iPhone Simulator and its target devices support a subset of the standard Xcode debugging tools. Interface Builder is also simplified. The Library lacks many of the standard objects available in OS X, including menu trees and the various data controller types. UIKIt objects replace the view design objects available in OS X. To support the different frameworks in iOS, the SDK includes alternative documentation, new code samples, and a different set of iOS project templates to match various possible app types. Template details are listed a little later in this chapter.

Using the Xcode Simulator

The Xcode Simulator is adequate for basic testing, and it can emulate display rotation and shake events. However, the Simulator is not a perfect model of the target devices. Code runs much more quickly in the Simulator, so it cannot be used for performance testing. There may also be minor differences between the simulated SDK and the iOS running on a real device. Code that works in the Simulator may occasionally crash on hardware and vice versa.

CROSS-REF

For information about downloading and installing Xcode and the iPhone Simulator, see Chapter 4.

The Simulator can emulate an iPhone and an iPad. To test iPhone apps on the iPad, choose the iPad Simulator as the Active Executable in the Build selector drop-down menu in Xcode. This runs the app inside the iPad's emulation window.

To convert an iPhone project to an iPad project, make a safety copy outside Xcode. Load the copy into Xcode, right-click a compiled target, and choose Upgrade Current Target for the iPad. This modifies the nib file and makes it possible to run the app as a full-screen native iPad app. There is no way to downgrade an iPad project back to the iPhone.

TIP

In spite of its limitations, you should use the Simulator for most of your development cycle. Compiling and loading an app into a hardware device take an extra minute or two, which slows down development. It can also clutter devices with unwanted, half-finished apps.

The accelerometer and compass are absent in the Simulator. Core Location returns a single position fix for the location of Apple's headquarters in Cupertino. Camera features are not available, even on Macs with a camera, and the photo, video, and iPod libraries are not fully implemented. To fully test code that accesses these features, you must run your app on a real device.

NOTE

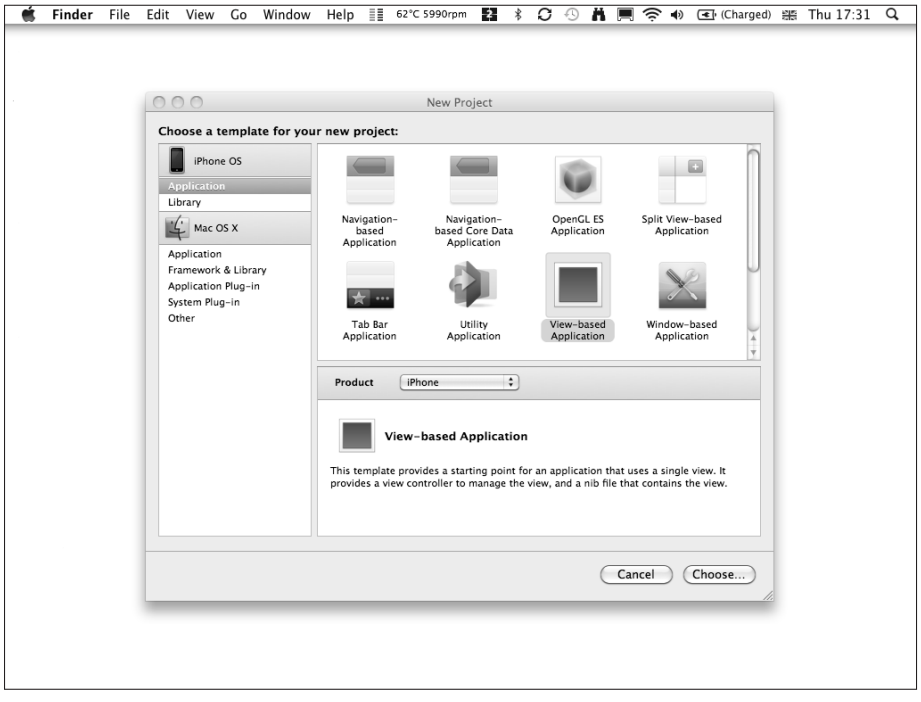
You must use the USB connector to load apps into a device. Bluetooth and wireless downloading are not supported. To use the live debugging tools, you must run apps with the cable connected. Once an app has been installed, you can disconnect the cable and test it away from your Mac, but you cannot debug it without the cable link. Installed apps can be docked, deleted, and rearranged on Springboard like other apps.

Introducing the Xcode templates

As with OS X, Xcode includes a set of application templates for iPhone and iPad projects, shown in Figure 19.15. They include barebones app skeletons that you can flesh out with your own code. A selection of relevant useful methods is included in each file, but most are commented out. The included methods are not comprehensive (many more are listed in the class reference documentation) but are a convenient reminder of the most widely used features in each class.

Figure 19.15

A first view of the templates included in the 3.2 iOS SDK. Use the Product drop-down list to choose either an iPad or iPhone project.



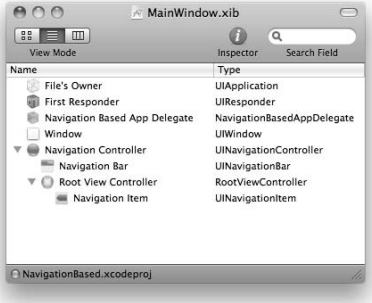
The *Navigation-based Application* template shown in Figure 19.16 creates an empty table view with a navigation controller. You must add your own code to populate the table from a data source, implementing optional features for each cell as needed. Code to switch views by pushing a new controller onto the navigation stack is included but commented out.

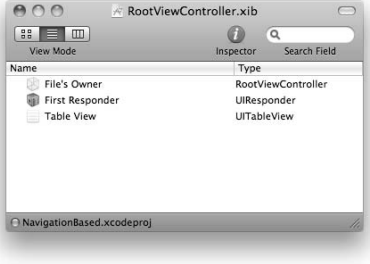
TIP

To create a hierarchical multi-view application using the Navigation-based application, add one or more new table-view-based classes. Classes either display another menu-like selection list or a final view that displays information and control options.

Figure 19.16

In the Navigation-based Application template, the Navigation Controller remains permanently visible and the RootViewController swaps views behind it. The default nib includes an instance of `UITableView`.





The *OpenGL ES Application* template illustrated in Figure 19.17 creates an OpenGL view, with a special `EAGLView` view subclass that provides a context for OpenGL code. There are two rendering classes: `ES1Render` and `ES2Render`. The first supports OpenGL ES 1.1 code, the second supports OpenGL ES 2.

`EAGLview.m` includes an `initWithCoder:` method that automatically picks a renderer by selecting between these two classes. If the OpenGL ES 2 renderer is not available on the target device, the method falls back to the `ES1Render` class.

TIP

By default you must include code for both renderers, which effectively means writing your OpenGL ES code twice. If your app does not use any of the software shaders or other advanced features in OpenGL ES 2, you can simplify it by editing the conditional test in `initWithCoder:` to force selection of the `ES1Render` class. You can then add your code to `ES1Render.m` — and ignore `ES2Render.m` because it is never referenced.

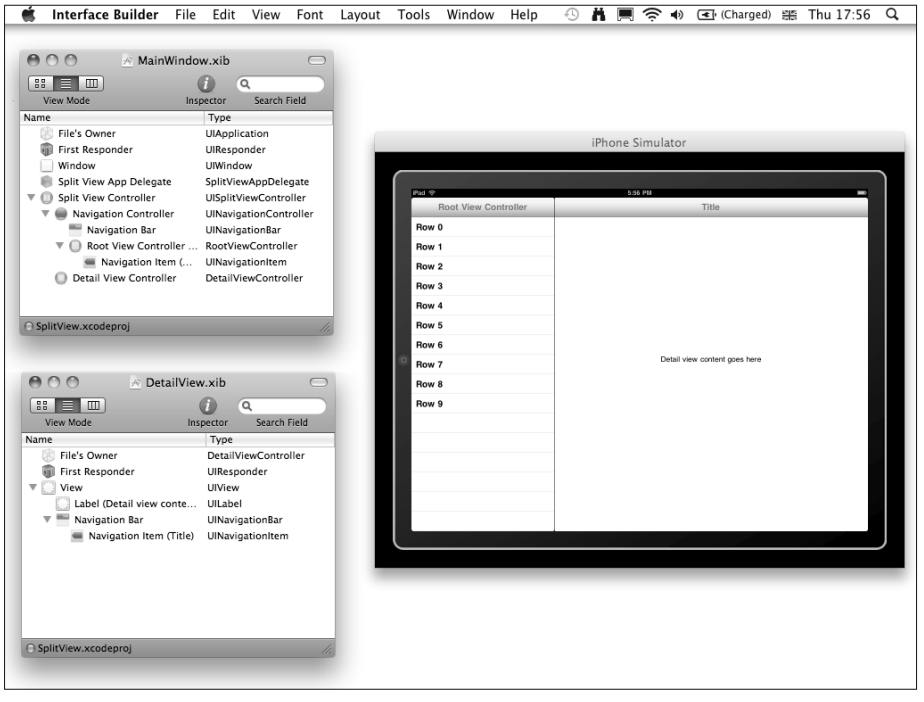
Figure 19.17

The OpenGL ES Application template includes sample code to create and animate a colored square. The Shader files included in the template contain very simple color and animation code used with the `ES2Renderer`.

The *Split View-based* template shown in Figure 19.18 is only available on the iPad. It creates a split view display with a table view at the left and a blank view at the right. A split view controller manages both views. The table view is allocated and initialized dynamically and is not loaded from a nib file. You can customize this template to load a view of your choice — with a custom nib — by replacing the table view creation code with code to load and display a custom view controller class.

Figure 19.18

The table view that appears at the left of the Split-view template is not included in the project nib files; it is added dynamically in the code when the app runs.



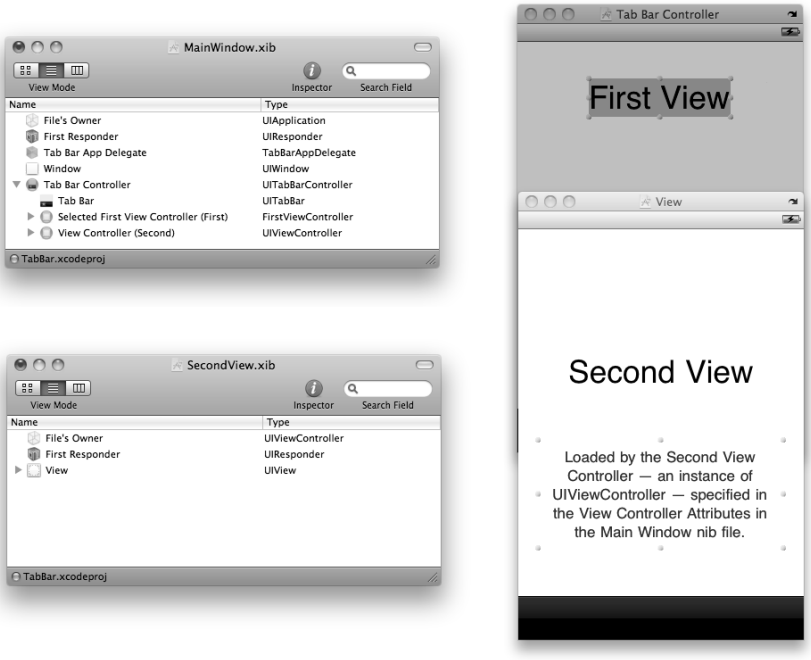
CAUTION

When you create a new view controller class for the iPad with a nib, the nib is a standard size and covers the full screen. You cannot edit this size in Interface Builder, but you can resize a view dynamically in code after it loads. For a split view, the size is irrelevant; the rightmost detail view covers it automatically, leaving a visible 320 pixel width at the left.

The *Tab Bar Application* template illustrated in Figure 19.19 creates an app with two tab bar items that switch between two views. To add more views, or to change the type of view selected by the tabs, open the `MainWindow.xib` file in Interface Builder, select the Tab Bar Controller, and edit the list of the View Controllers visible in the Attributes pane to add or remove controllers or change their type. Add a new view controller class — customized, as needed — for each new list entry. Add a Tab Bar Item subview to each controller. When you arrange the hierarchy like this, the main Tab Bar Controller automatically switches views when each tab bar item is tapped.

Figure 19.19

The two views in the Tab Bar Application template are switched automatically by the Tab Bar Controller. No code is needed.



The *Utility Application* template shown in Figure 19.20 creates a two-view application. The first view is empty with a single information button at the lower right. Tapping the button spins the view to reveal a modal flipside view, with a Navigation bar that shows a title and includes a Done button. Tapping the Done button spins the view again to reveal the original.

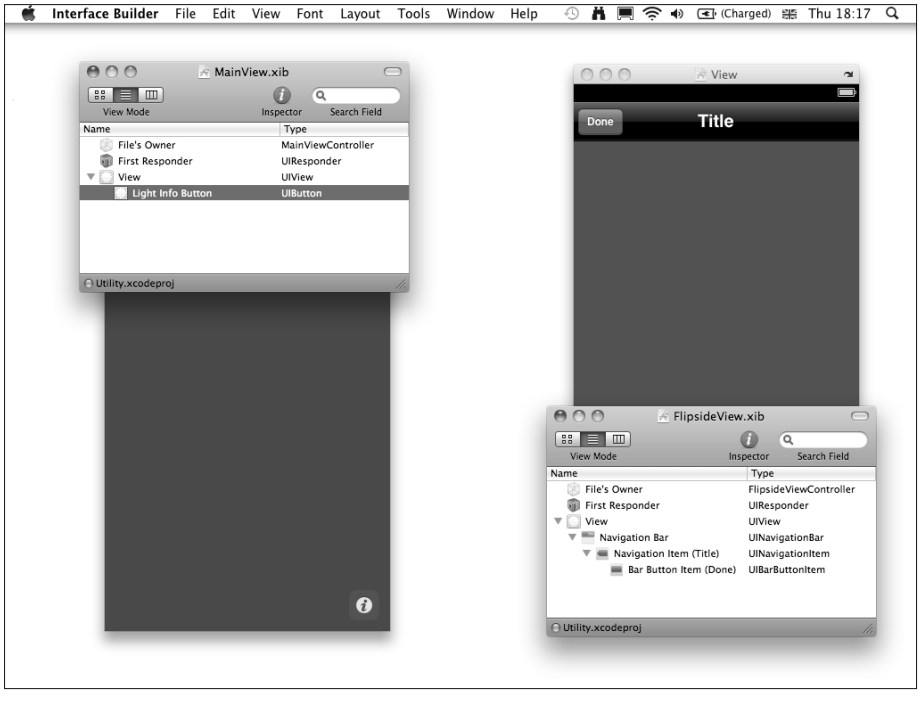
The template is designed for a very simple two-view app. Apps rarely are this simple, and it is more useful as an example of two techniques: switching views with the `presentModalViewController:` method and using protocol/delegate communication between views. The flipside view includes a delegate property, used as a link to the main view controller. When the user requests a return, the flipside view triggers a custom `flipsideViewControllerDidFinish:` method in the delegate. In the delegate — which is the main controller — this method calls the `dismissModalViewControllerAnimated:` method, and the modal view disappears and is replaced by the main view. You can populate both views with further features as needed.

TIP

The Done button and the Navigation bar on the flipside view are large and distractingly visible. A more typical app might implement the Done feature with a button, a segmented switch, or one of the touch methods. To simplify the flipside view, open its nib file, delete the Navigation bar, and replace it with an alternative return method.

Figure 19.20

The main view in the Utility Application template uses a light info button graphic to swap views. The flipside view includes a less subtle navigation bar and Done button.

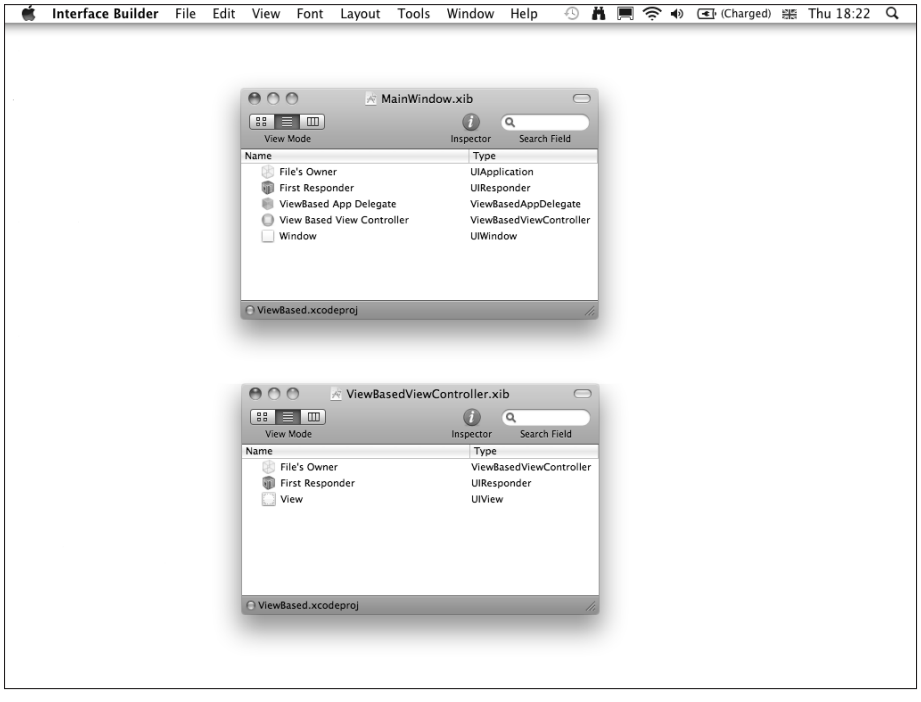


The *View-based Application* template shown in Figure 19.21 is minimal but versatile. It includes a window that loads a separate view controller with an associated nib that displays an empty gray view. The view is blank, but the controller code includes a minimal list of useful methods that have been commented out, including `viewDidLoad`.

To create a more complex app, assemble an interface from subviews in the nib file in the usual way and link them to new custom methods in the view controller. To create an app with more views, add one or more new controller classes to the project and include view switching code in each. You can also use this template to create apps with navigation controller or tab bar switching. Add these items to the main window nib, not the view controller nib. You can then use them to load further view controllers, as needed.

Figure 19.21

The View-based Application template is the classic app configuration, with a subclass of `UIViewController` loading its view from the associated nib.

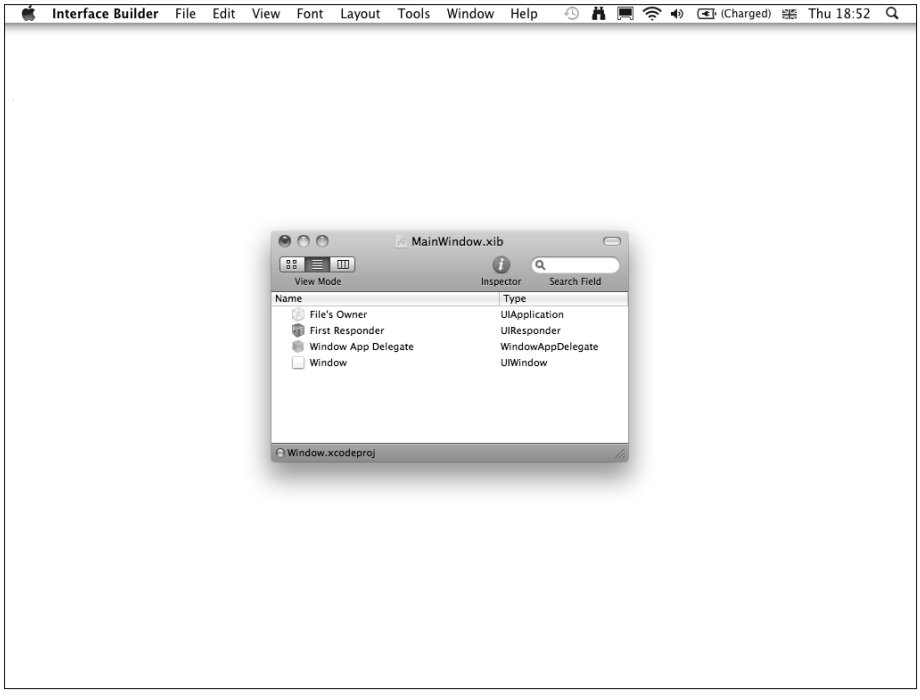


The *Window-based Application* template shown in Figure 19.22 includes a single app delegate with an associated nib file that includes an instance of `UIWindow`. This template is the most skeletal and lacks a view controller. You can use it to build a micro-app that uses as little memory as possible, adding extra features, outlets, and action methods directly to the app delegate.

You can also use this template as a starting point for apps that use a navigation controller or a tab bar. Generally, it is easier to start from the other prebuilt templates for these app types, but if your app has unique navigation requirements, it can be simpler to build them from scratch.

Figure 19.22

The Window-based Application template is similar to the View-based Application template without the extra controller — and without it, there is no way to swap views.



Building a Simple Application

Now that you have been introduced to the key features of iOS, you can create your first app. This app is more complex than a hello world app; it demonstrates how to build views with simple features and how to implement view swapping.

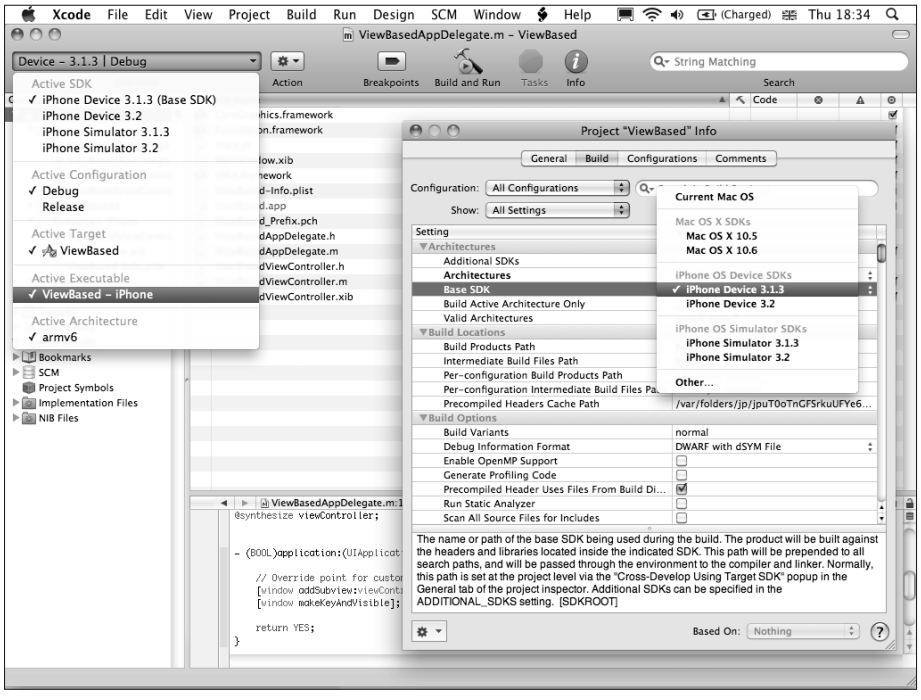
Begin by creating a new project in Xcode using the View-based Application template and selecting the iPhone as a target in the drop-down list in the center of the template window. Save the project as **FirstApp**. Select the Simulator as a build target using the drop-down list at the top left of the Xcode window. Check that the Target SDK is available, as shown in Figure 19.23. From the same menu, set the Active Executable to iPhone. Build and Run the template.

CAUTION

In some versions of the SDK, you must set the base SDK manually in the Project Info dialog before you can select it in the Configuration drop-down list. It may be set to a default that does not support the iPhone SDK.

Figure 19.23

If the target SDK is missing, open the Project Info window, choose Build ➪ Architectures ➪ Base SDK to display a menu that can add it to the project. You can then choose it in the Configuration drop-down menu and set it as an Active Executable.



The app loads in the iPhone Simulator. If it appears in the iPad Simulator, double-check that you set the Active Target option correctly. You may need to set it again whenever you quit and restart Xcode.

Adding view controller subclasses

The empty gray view isn't very exciting, but you're about to change that. Your new app will have two views, with buttons that switch between them. Start by making two new subclasses of `UIViewController` to hold the extra views. Many apps use a subclass of `UIViewController`, so it is included in the list of file templates in Xcode, with an option to create a new nib file with the new class files.

Right-click the Classes folder and choose Add➪New File. Choose Cocoa Touch Class at the top left, and click the `UIViewController` subclass file template. Make sure the With Xib for user interface option is selected.

Click Next, and save the new class as **AViewController**. Xcode adds it to the project, with a nib file. Repeat this sequence to add another new class called **BViewController**, with its own separate nib.

The app now has three view controllers and three nib files — the original controller with a blank view generated by the template, and two new view controllers, as shown in Figure 19.24. Three controllers may seem overkill for a two-view app, but it simplifies the project. Advantages include:

- **Simplified memory management.** View controller classes cannot hold pointers to themselves or their properties after they have been released. A root controller acts as a convenient pointer store while managing other views, making it possible to release them safely.
- **Simplified event handling.** You can include code to manage user events in their view controller, or you can let the responder chain pass them up to the root controller. This can simplify event handling; it is easier to code than an explicit delegate, and it "just works."
- **Simplified data storage.** Although it is not quite official Model-View-Controller (MVC), you can add a data store, or an interface to a data store, to the root controller to manage all of the app's data needs.

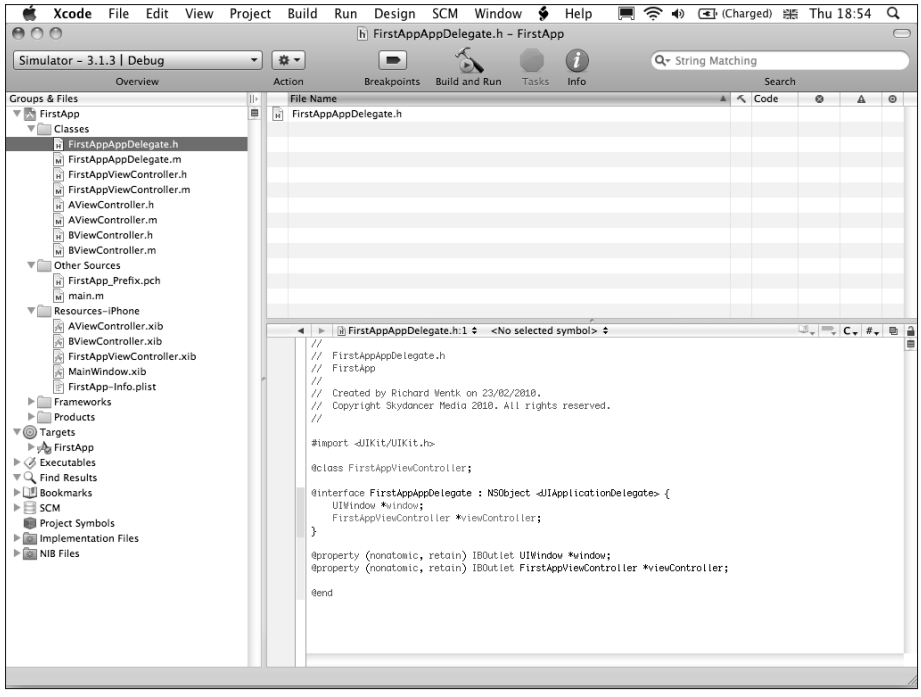
Implementing the view controllers

With this design pattern in mind, you can start coding the app. Begin by defining the extra view controllers as properties in the root by adding them to the header file. Include an action method called `showBView:` that triggers the initial view swap.

```
@class AViewController;
@class BViewController;
@interface FirstAppViewController : UIViewController {
 AViewController *aViewController;
 BViewController *bViewController;
}
@property (nonatomic, retain) AViewController *aViewController;
@property (nonatomic, retain) BViewController *bViewController;
-(IBAction) showBView: (id) sender;
@end
```

Figure 19.24

The complete class and resource list for the app includes an app delegate, three view controller classes, and four nib files. Many apps use a similar design, so you may want to save this project for re-use.



In the implementation file, add start-up code that releases the original empty gray view and replaces it with the contents of `AViewController.xib`. This code goes into the `viewDidLoad` method in the root controller, which is broadly equivalent to `awakeFromNib` in a Cocoa project.

```
#import "FirstAppViewController.h"
#import "AViewController.h"
#import "BViewController.h"
@implementation FirstAppViewController
@synthesize aViewController, bViewController;
- (void)viewDidLoad {
 //Load the first view to replace the default view in the root
 //controller nib
 self.aViewController = [[AViewController alloc]
  initWithNibName: @"AViewController" bundle: nil];
 [self.view release];
 [self.view insertSubview:aViewController.view atIndex:0];
}
```

This code is simple, but there is a lot going on. It loads an instance of the `AViewController` and initializes it with its nib. Then it plugs a pointer to this new controller into the corresponding property in the root view, saving it for later.

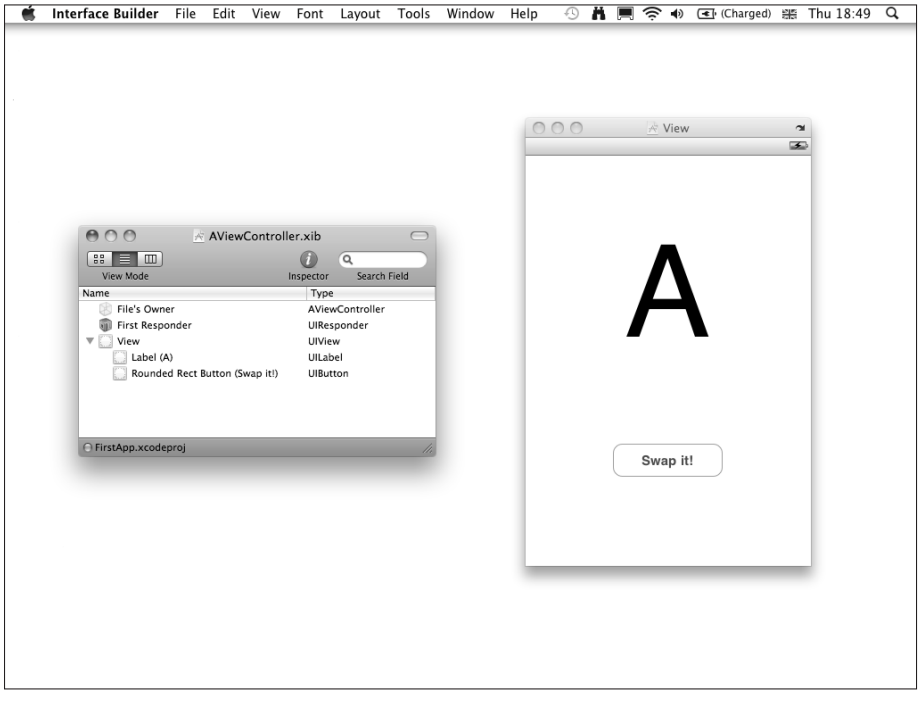
The next line releases the default gray view from memory. This is slightly risky: this code may crash if the OS tries to refresh or reference the view. In this example, the view is not being redrawn or refreshed, so it works reliably. Finally, the `insertSubview:` method adds the view loaded from `AViewController.xib` into the app's view hierarchy, displaying it.

Creating views

You can now edit the `AViewController.xib` file. The example view shown in Figure 19.25 includes a large label and a smaller round rect button. The button triggers the swap to the BView.

Figure 19.25

The first view, with a label and a button. You can customize the button graphic by loading it with your own PNG files in the Inspector. This example uses the default.



If you save the new nib and build and run the app, it will load and display the AView nib design, as shown in Figure 19.26.

Figure 19.26

Running the app loads the new nib and displays it.

The new button remains unlinked. It flashes when tapped, but does nothing. To fix that, add a `showBView:` method to `FirstAppViewController.m` with code to load and display the B view.

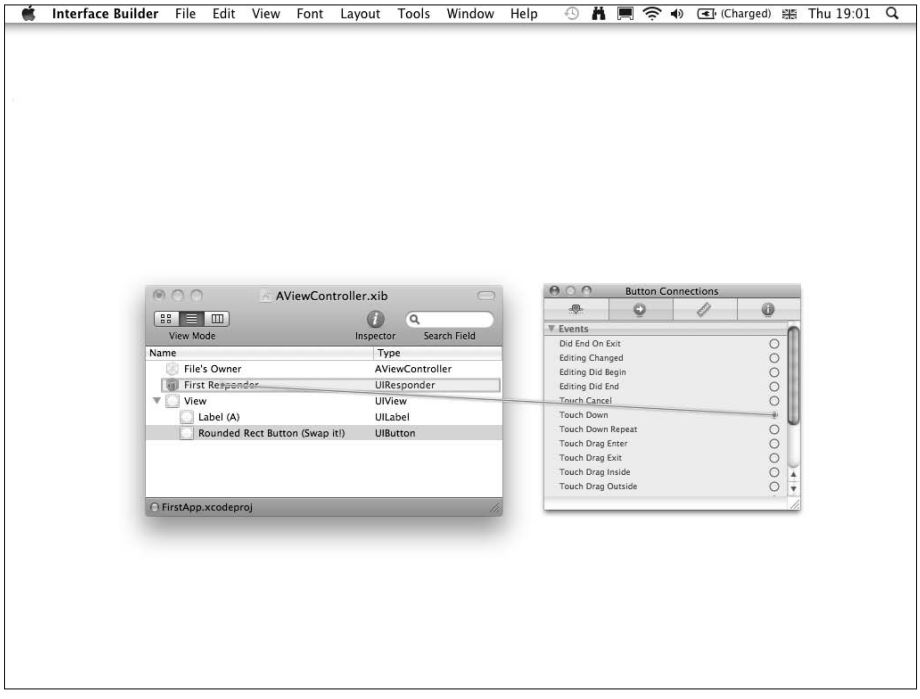
```
-(IBAction) showBView: (id) sender
{
 self.bViewController = [[BViewController alloc] initWithNibName:
   @"BViewController" bundle:nil];
 [aViewController.view removeFromSuperview];
 [self.view insertSubview:bViewController.view atIndex:0];
}
```

This is almost identical to the code in `viewDidLoad`. It loads the `BViewController` nib file, then calls `removeFromSuperview` to unlink the previously visible view from the view array and release it. The last line inserts the B view into the hierarchy to display it.

Save the file and choose File➪Reload All Class Files in Interface Builder to add the new `showbView:` method as a linkable action. Click the round rect button in the view to select it. Click the Connections tab in the Inspector window to reveal a list of events. Link the button's Touch Down event to the First Responder object, as shown in Figure 19.27. Select the `showBView:` method from the floating pop-up menu when it appears.

Figure 19.27

Linking a button event to First Responder, which bypasses the class controller and sends it to the app's root view controller.



CAUTION

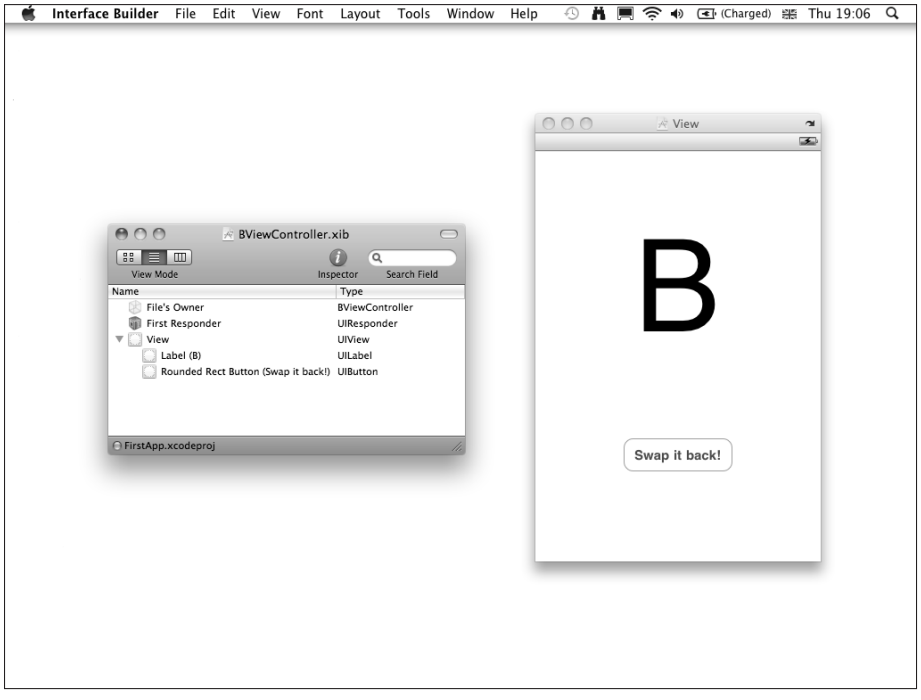
The list of user events is a standard feature in UIKit. Objects display the full list in IB, but most generate only some of the events; for example, round rect buttons do not support editing, so they never send the Editing Changed event. Text fields do not generate Touch Down events. If your view is not generating events as it should, double-check the events you have linked here.

Handling events with the Responder Chain is a tricky linking technique — linking to First Responder bypasses `AViewController` and sends Touch Down events directly to `showB View:` in the root controller. You can use this trick to handle almost any event type in any context, but it is not MVC-approved and you should save it for view swap events handled by a root controller. If you extend `AViewController` with more features — buttons, sliders, and so on — add the code to handle them to the `AViewController.m` file in the usual way.

You can now edit the B view nib to show a large label and another round rect button, as shown in Figure 19.28. Leave the button unlinked for now.

Figure 19.28

Designing the B view, which is very similar to the A view, with minor labeling changes



Save the edited nib, and build and run the application. Tap the button in the A view, and the B view appears, as illustrated in Figure 19.29.

Figure 19.29

Testing that the button in the A view displays the B view

Handling events with protocol messaging

Next, add code to the B view to swap back the views; until you do this, the button in the B view remains inactive. You could repeat the previous steps to add a `showAView:` method in the root controller, modifying the code slightly so it loads the A view nib and linking it to the button in the B view via First Responder. This is the simplest solution and it would work perfectly.

This example demonstrates how to solve the problem by creating a delegate protocol. Unlike the simpler option, the protocol can pass parameters between the view controllers. Begin by adding a new delegate protocol to the interface of the root view controller. Change the declaration in `FirstAppViewController.h` to:

```
@interface FirstAppViewController : UIViewController <BViewControllerDelegate>{…
```

Now you can define the details of this protocol in `BViewController.h`, adding other class features to create a delegate property and an action method.

```
@protocol BViewControllerDelegate;
@interface BViewController : UIViewController {
id <BViewControllerDelegate> delegate;
}
@property (nonatomic, assign) id <BViewControllerDelegate>
   delegate;
- (IBAction)showAController;
@end
@protocol BViewControllerDelegate
- (void)BViewControllerDidFinish:(NSString *)aString;
@end
```

The header includes a separate `showAController` method to handle button taps from the button in the view. Implement this method in `BViewController.m` so that it calls the `BViewControllerDidFinish:` method on the delegate object.

```
@implementation BViewController
@synthesize delegate;
-(IBAction) showAController {
 [self.delegate BViewControllerDidFinish:@"Back to View A…"];
}
```

In IB, reload all class files and link this method to the button in the B view.

To finish the app, add an implementation of `BViewControllerDidFinish:` in `FirstAppViewController.m`. If this implementation is missing, the app will crash with a selector error when the return button is tapped in the B view. Here is a simple version:

```
- (void)BViewControllerDidFinish:(NSString *)aString {
 NSLog(aString);
 self.aViewController = [[AViewController alloc] initWithNibName:
   @"AViewController" bundle:nil];
 [bViewController.view removeFromSuperview];
 [self.view insertSubview:aViewController.view atIndex:0];
}
```

It is almost identical to the previous view swapping code fragments: it allocates and initializes the A view controller, removes the B view, and replaces it with the A view.

One final step is missing: there is no delegate defined for the B view controller. To set one, add a line to the `showBView:` method on `FirstAppController.m`, nominating the root controller as the delegate.

```
-(IBAction) showBView: (id) sender
{
 self.bViewController = [[BViewController alloc] initWithNibName:
   @"BViewController" bundle:nil];
 self.bViewController.delegate = self;
 [aViewController.view removeFromSuperview];
 [self.view insertSubview:bViewController.view atIndex:0];
}
```

Build and run the app, and test it by tapping the buttons to swap the views. An `NSLog` call lists the string to the console, so that you can confirm that data is being passed to the root view controller from the B view, as shown in Figure 19.30.

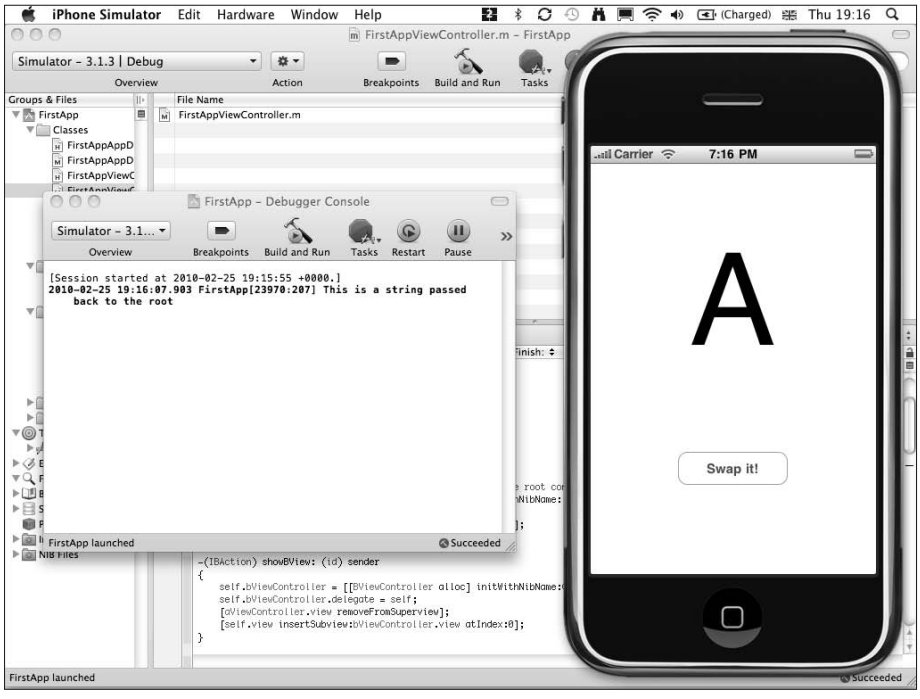
Creating an animated view swap

The instant view swap effect works, but is unimpressive. You can improve your app's production values by animating it. iOS animation code is similar to OS X animation code, which was introduced in Chapter 17. Adding animation code around the view swap creates longer versions.

```
- (void)BViewControllerDidFinish:(NSString *)aString {
 NSLog(aString);
 self.aViewController = [[AViewController alloc] initWithNibName:
   @"AViewController" bundle:nil];
 //Start of the animation code
 [UIView beginAnimations:@"An Arbitrary Transition Name"
   context:nil];
 [UIView setAnimationCurve:UIViewAnimationCurveEaseOut];
 [UIView setAnimationDuration:1];
 UIViewAnimationTransition transition;
 //Create and load a transition type
 transition = UIViewAnimationTransitionFlipFromRight;
 [UIView setAnimationTransition: transition forView:self.view
   cache:YES];
 [bViewController.view removeFromSuperview];
 [self.view insertSubview:aViewController.view atIndex:0];
 [UIView commitAnimations];
}
```

Figure 19.30

The finished app. Choose Run ➪ Console to confirm that the B view is passing a string parameter back to the root view controller when its button is tapped.



In this example, a `beginAnimations:` method marks the start of an animation block, and a `commitAnimations` method ends it and also triggers the animation. The `transition` property selects an animation type; here, a flip effect, shown in Figure 19.31. The `animationCurve` constant defines how the animation develops over time. To animate the first view swap, add equivalent animation code to the `showBView:` method. For more details about animation properties and constants, see the `UIView` class documentation.

Figure 19.31

The B view in motion. The view is automatically darkened as it spins to enhance the effect. The `UIWindow` background color attribute has also been set to black in IB to contrast the white views with a black background.

Selling in the App Store

Anyone who has qualified for the $99 Standard Developer Program annual fee can submit an app to the App Store. Not all apps are accepted. According to Apple, approximately 95 percent of submitted apps pass certification, with an average approval time of two weeks. Apps may be refused because of one or more of the following reasons:

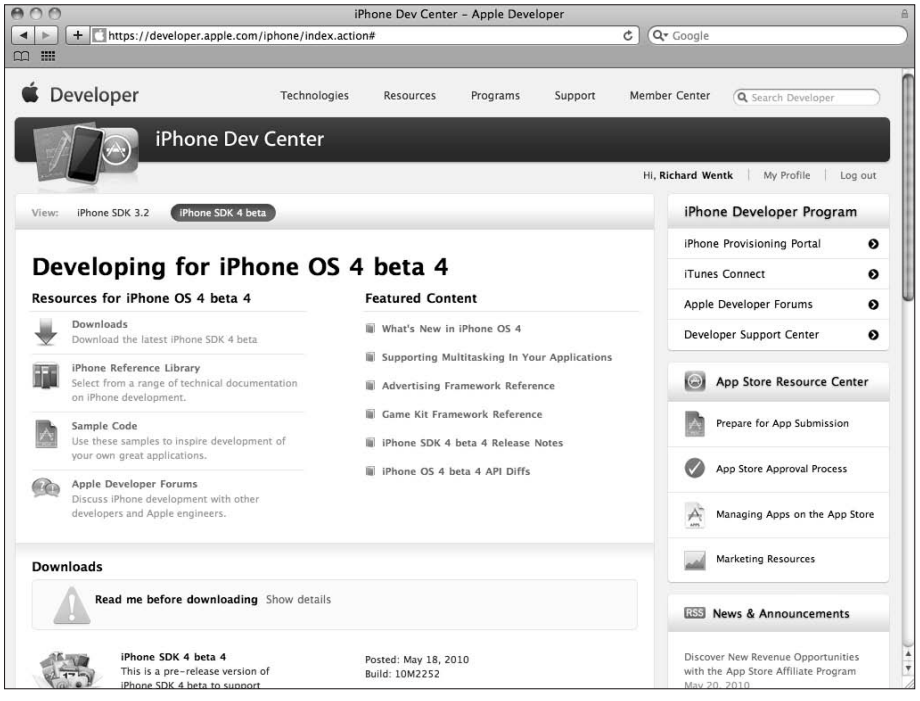
- **Bugs and limited functionality.** If the app crashes during testing or its features do not match its description, it is refused.
- **Unacceptable content.** Apple rates apps and uses an approval grid to check for adult features, representations of real or simulated violence, and other criteria. Access for minors is controlled. Apps that cannot be rated are not accepted.
- **Unacceptable commercial features.** Apps that use excessive cellular bandwidth or conflict with the current or future plans of Apple and its mobile partners are not accepted.
- **Unacceptable software features.** Apps that use undocumented API (Application Programming Interface) calls or non-Apple frameworks are not allowed.

There is no way to buy, sell, or load apps independently without *jailbreaking* — downloading a tool to hack the security of — an iPhone. Jailbreaking is inherently risky and may *brick* (lock) a phone permanently. An alternative developer network exists for jailbroken phones, with its own alternative app market, but it is much smaller than Apple's App Store and is not officially supported.

Figure 19.32 shows the Developer Program Web site.

Figure 19.32

The Developer Program Web site includes links to iTunes Connect for App Store uploads and the Developer Program Portal for security and provisioning at the top right of the page.



CAUTION

Jailbroken phones are relocked by each new official iOS update, wiping existing unofficial apps. There are advantages to jailbreaking, including open app development and simple Unix shell access to the iPhone's file system, but jailbreaking remains more of a hobbyist and enthusiast interest and has had a minimal impact on the main app market.

Understanding certificates, provisioning profiles, and permissions

Before you can submit an app to the App Store, you should test it on hardware and then create a special App Store distribution build. Testing and distribution are controlled with *code signing* technology. Apps run on the Simulator without code signing, but can only be tested on hardware devices after a complex certification process. This gives Apple more control over the app market — permission to test and sell apps can be revoked without notice — but it also complicates development.

You can manage code signing permissions and files in the *iPhone Developer Program Portal,* which is listed as a link on the main iPhone Dev Center Web page. Creating a full set of permissions isn't simple, but most of the steps are a one-time requirement.

The full set of permissions includes:

- **A Developer Certificate.** Needed to test apps on hardware.
- **A Distribution Certificate.** Used to create a distribution build of an app, ready for uploading to the App Store.

NOTE

Developer and Distribution Certificates are mutually exclusive — an app can only be signed with one or the other. When you create a distribution build for the app store and sign it with a Distribution Certificate, you cannot run that build on a device. The only way to run a distribution build is to buy it from the App Store. Fortunately, promotional certificates are available in iTunes Connect, and you can use them to "purchase" your own apps for free. Unfortunately, they are only valid in the U.S. iTunes Store.

- **An App ID.** A short identification string. A single wildcard App ID can be used as a placeholder for an app's Bundle Identifier in Xcode, and you can use the same ID in all of your apps.

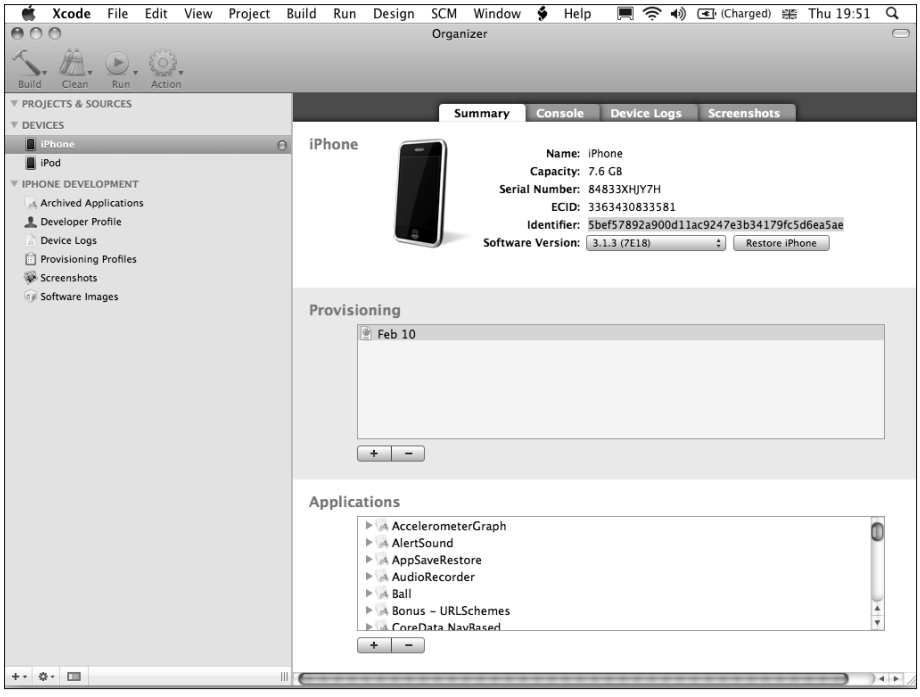
NOTE

If you use Apple's Push Notification or In-App Purchase services, which allow users to buy content from an online store in their apps or receive event notifications from a Web server, you must create a unique ID for each app.

- **A Device List.** Keeps a list of your test devices. You can add up to 100 devices per year. Figure 19.33 illustrates how to find and copy a device ID in Xcode.
- **A Provisioning Profile.** A temporary key imported into Xcode and installed on your hardware devices. It includes a copy of the ID in your Developer Certificate and a list of your devices. When the two match, Xcode allows you to install your app on a hardware device, and the device allows you to run it. A Provisioning Profile expires after three months. Apps built with it stop working.

Figure 19.33

Highlighting a device ID to copy it when you create a device list. When the online wizard creates a provisioning profile, download it and drop it onto the Provisioning Profiles item in the left-hand pane.



Generating and using certificates and permissions

Apple has now automated this process and created an online Assistant that takes you through it step by step, as shown in Figure 19.34. When the Assistant completes, you can download a profile to your desktop.

Xcode projects include a Code Signing selection feature in the Project Properties dialog. Choose your Developer Certificate here, while you are creating a project, and your Distribution Certificate before the final App Store build. Xcode reads the certificates from your keychain and builds your user keys into each app.

To install a Provisioning Profile in Xcode, choose Window➪Organizer. Drag and drop the provisioning profile file from the desktop onto the Provisioning Profiles item under IPHONE DEVELOPMENT. When an app builds for a device target, Xcode automatically tags it with the profile.

Figure 19.34

The online Assistant

To install a profile on a device, choose Provisioning Profiles to see a list of your profiles. Attach a device, wait for it to be recognized, and drag a profile from the list at the top to the device under the DEVICES divider.

Before you can run the app on the device, you must select a profile by choosing Settings ➪ General ➪ Profile on the device. This is a one-time requirement. When the profile expires, repeat Step 12 to create a new profile, install it in the Organizer to load it into Xcode, and select the new profile on the device. You can use the same profile for every device you have listed on the Developer Program Portal.

After completing these steps, choose the Device build target in Xcode to build, install, and run an app on a device.

NOTE

You can use the *ad hoc distribution* option on the Developer Program Portal to create and share profiles and test builds among beta testers without using the App Store. Testers can install the profile and app via iTunes. For details, choose Distribution ➪ Ad Hoc on the Portal.

Packaging an app for the App Store

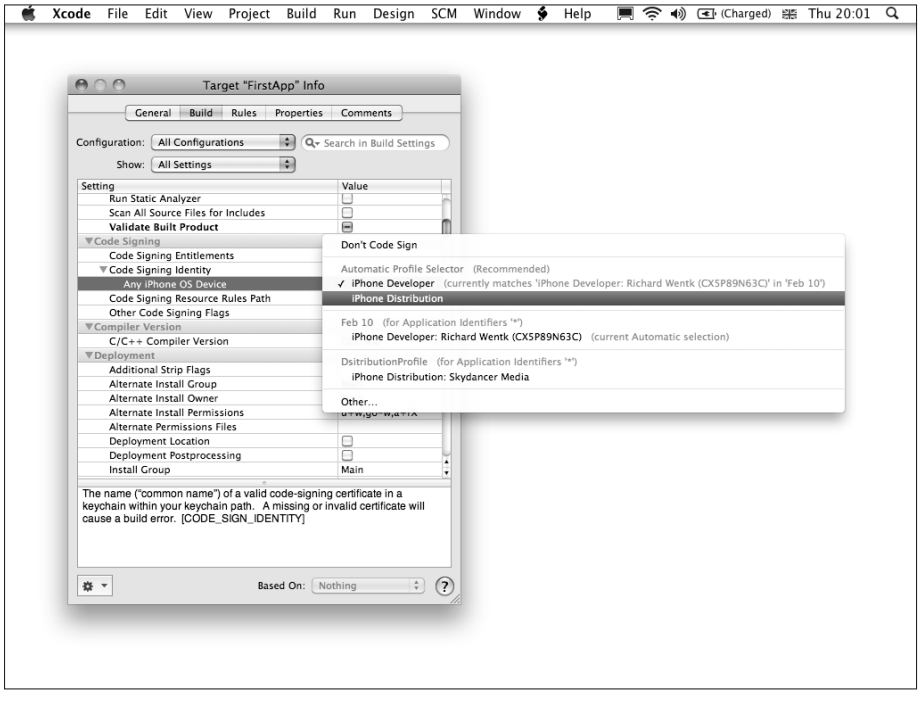
Although Xcode includes a Release build target, this simply removes debugging information from the build. It is not suitable for App Store submissions. Packaging an app for distribution requires an extra step that installs a Distribution Certificate, as shown in Figure 19.35. If you do not have a Distribution Certificate, follow the steps in the previous section to create one before you start.

1. **In Xcode, right-click the project and choose Get Info from the pop-up list.** Click the Configurations tab. Click the Release configuration and then click Duplicate. When the copy appears, give it a new name, such as **Distribution.**

2. **Click the Build tab to reveal the build settings.** Find the Code Signing Identity section. Click the Code Signing Identity triangle, right-click iPhone Developer, and choose iPhone Distribution from the pop-up list. Close the window.

Figure 19.35

Selecting the Distribution Certificate before building for distribution. The Automatic Profile Selector feature makes a best guess about the most appropriate certificate — and usually gets it right.



3. **Right-click the build target, under the Targets header.** Choose Get Info from the pop-up list. Click the Properties tab and replace the generic identifier with a unique app name. Fill in the app's icon file and version number.

NOTE

The icon file is a 57 × 57 pixel PNG and identifies the app in Springboard. Springboard automatically adds glassy highlights; do not try to add your own.

4. **Select the new Distribution configuration and choose Build ➪ Build to build it.**
5. **Click the build product in the Products folder.** Choose Reveal in Finder from the drop-down tools list. In Finder, right-click the `.app` file and choose Compress <file name> from the pop-up list. This creates the distribution file, ready for upload.

Uploading an app to the App Store

App uploads, financials, sales reports, and tax and banking information are managed using iTunes Connect, accessible from a link at the top right of the iPhone Developer Program page. Before you upload an app, choose Contracts, Tax and Banking to enter your bank details and accept the standard iTunes Connect contract. You must also define your tax status for the territories in which your app will be sold.

CAUTION

International taxation details are complex and outside the scope of this book. iTunes Connect includes digital paperwork to minimize international tax issues, but you may still need to supply proof of status and correspond with national taxation agencies; otherwise a proportion of your income may be withheld in certain territories. For information and advice, consult a professional accountant.

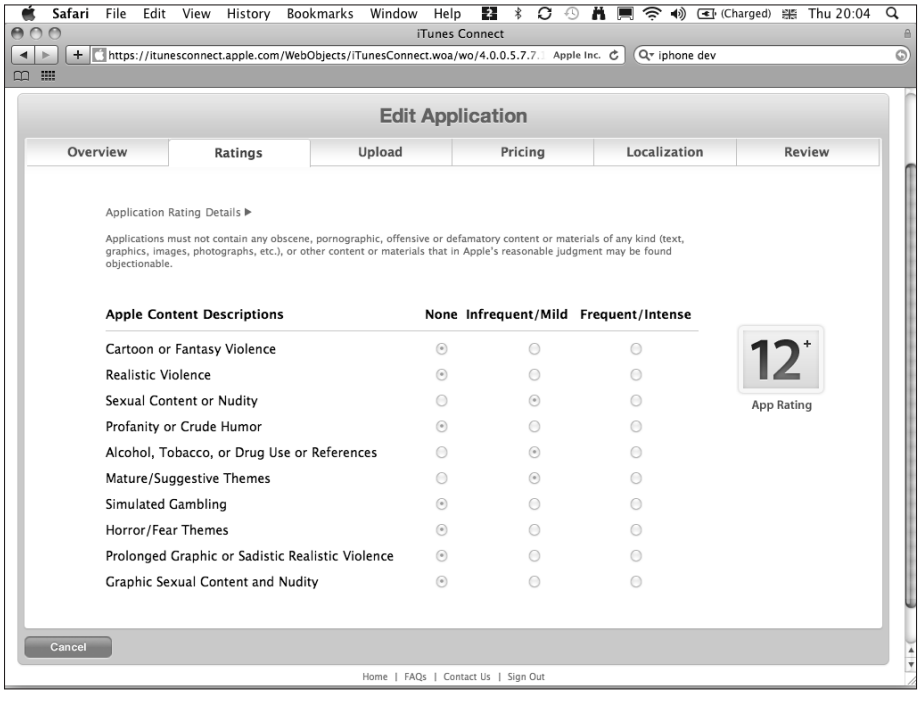
Before uploading an app, collect the following:

- The compressed distribution build file.
- At least one but no more than five PNG screenshots to illustrate your app in the App Store.
- A 512 × 512 PNG version of the app's icon file
- A SKU number (an arbitrary product ID code of your choice). The code must be different for every app.
- A sales description. The maximum is 4,000 characters.
- A list of keywords. The maximum is 100 characters.

App upload is simple. In iTunes Connect, choose Manage Your Applications and click Add New Application. Follow the prompts to define the app's listing in the App Store and upload the files. You must define your app's rating on a content grid, shown in Figure 19.36, that flags mature content. After a successful upload, you will receive a notification e-mail when the app is listed in the App Store.

Figure 19.36

The App Store ratings grid. Apps that require a check mark in the Frequent/Intense column are unlikely to be accepted.



Summary

In this chapter, you were introduced to the iOS device range and the key differences between Apple's mobile products. You learned how to understand the App Store business model, and you discovered the commercial differences between iPhone and iPad development. You were introduced to design hints to help you create apps that have the greatest chance of success in the App Store, concentrating on popular app types, and avoiding specialized, niche-oriented, and time-consuming projects.

Next, you learned about the differences between the OS X and iOS development models and were introduced to the limitations and design opportunities of the mobile platform. You explored some of the key features of the iOS programming model and discovered the importance of views and view controllers in app design. You also explored the properties of the iOS templates in Xcode, and you examined in detail the code and view designs of a simple view-switching app with two different approaches to view control and event handling.

Finally, you were introduced to the iOS security model and discovered how to acquire the certificated and provisioning profiles needed to test apps on physical devices, how to install them in Xcode, and how to build apps for testing and distribution. You were also introduced to the iPhone Developer Program Portal and iTunes Connect, and you learned how to prepare, package, and upload an app to the App Store and how to set up contracts, tax information, and bank details to receive income from your app.

Appendixes

In This Part

Appendix A
Building Dashboard Widgets

Appendix B
Maximizing Productivity and Avoiding Errors

Building Dashboard Widgets

Widgets were introduced in 2005 in OS X 10.4 Tiger. Like iPhone apps, widgets offer users a simple, accessible benefit in a stylish wrapper. Some widgets provide unit or currency conversions or utilities such as clocks, calculators, and countdown timers. Others package entertainment — cartoons are a popular choice — or Web information, such as traffic reports or weather. And widget games are always popular.

Users continue to find widgets fascinating because they make it easy to customize a Mac with a unique collection of attractively packaged micro-applications.

The look and feel of widgets is based very loosely on Yahoo!'s Konfabulator project, which was formerly sold as a separate OS X application. Internally, widgets use JavaScript, HTML, and CSS, but include links to elements of Cocoa, and can be extended with plug-ins written in Objective-C.

Introducing Dashboard

Widgets load and run automatically within *Dashboard* — the widget manager application. Dashboard, shown in Figure A.1, runs permanently as an OS X background task. It remains hidden until the user activates it by typing a function key or moving the mouse cursor to one of Exposé's hot corners.

To the user, Dashboard is somewhat like an alternative desktop. A selection of widgets appears in the Dashboard area, where they can be repositioned by clicking and dragging.

Clicking the Open (+) button at the bottom left of the Dashboard area displays a dock-like menu that lists all installed widgets. Choosing a widget in the dock copies it to the Dashboard "desktop" and installs it with an attractive ripple-like animation, illustrated in Figure A.2.

In This Appendix

Introducing Dashboard

Profiting from widgets

Understanding widget technology

Introducing Dashcode

Building a widget in Dashcode

Figure A.1

Introducing Dashboard, with a selection of Apple standard, downloaded, and homebrewed widgets. All widgets can potentially pull data from the Web and display it inside a custom frame.

Dashboard is used modally — it dims the background when it appears, and widgets zoom in from the top of the screen. Background applications remain running, but widgets are the focus as long as the Dashboard is visible. To dismiss Dashboard, a user can type a preset function key, trigger an Active Screen Corner action, set up the Exposé and Spaces System Preferences panel, or click anywhere on Dashboard's background.

TIP

Widgets can continue running as background processes when Dashboard is hidden, but they should only use processor cycles when they need to. It's good practice to include code that halts or pauses a widget when Dashboard isn't visible. The ideal widget pauses when Dashboard is dismissed and quickly updates itself when Dashboard appears. Apple's widget templates include code for handling the various possible Dashboard states.

Figure A.2

Installing a new widget creates a ripple animation. The cross icons are added automatically in edit mode. Clicking these icons closes the widgets, removing them from the main Dashboard view, but not uninstalling them permanently.

Profiting from Widgets

Although iPhone apps and widgets use different technologies, they offer similar user benefits. When widgets first appeared, they created a new market for developers and sparked a small gold rush. Widgets are simpler and easier to build than iPhone apps, and they can be sold directly from a Web site without Apple approval. Optionally, they can also be submitted for an official listing in Apple's widget "store" at `www.apple.com/downloads/dashboard`.

However, when the iPhone's App Store was founded, it immediately eclipsed the widget market. Currently, most widgets are listed as freeware, and opportunities for direct widget sales are very small. Although only a small number continue to sell well, widget development still has commercial potential, and custom widget creation remains a small but potentially lucrative market for designers and consultants. Some of the advantages to widgets are as follows:

- **Although widgets can appear toy-like and simple, they can be powerful.** For example, they can process and summarize Web and server data, create secure in-house communications tools, or provide continuous news and information updates. Widgets can use polling or push technology to display information as it changes.
- **Widgets can support ad sales and donation boxes.** Adding an ad banner space to a widget and linking it to an ad server is relatively easy. Adding a donation box to the widget itself or to a download page is even simpler. A widget can display a new ad every time Dashboard is loaded or on a regular timed update cycle. Unlike ads on Web pages, widget content can't be blocked with an ad filter.
- **Widgets can support mobile apps and provide a front end to a commercial Web service.** For example, widgets for Twitter and Facebook remain very popular. Even though stand-alone applications may have more features, widgets offer instant-on convenience.

TIP

Users are often unenthusiastic about ad-based products. If a widget is too gaudy or intrusive, users will uninstall it. But if the widget's features are interesting enough, and ads are carefully targeted or made simple and unobtrusive — for example, as short understated text messages, rather than as garish animated banners — widgets can be an effective marketing medium for services, content, news, apps, and other products.

- **Widgets are popular and accessible.** They may appear in the future on platforms such as the iPad, or even the iPhone, and if a version of Dashboard appears on a mobile platform, developers can expect another gold rush.

Understanding Widget Technology

Internally, a widget is a scripted Web page in a custom wrapper. The wrapper provides a graphical frame and includes *parts* — placeholder objects that contain text, images, animations, video files, and other information. The widget's code reads or generates content and writes it to the placeholders. It is possible to build very simple widgets with plain HTML and no scripted features, but code is usually added to create a richer and more interactive user experience.

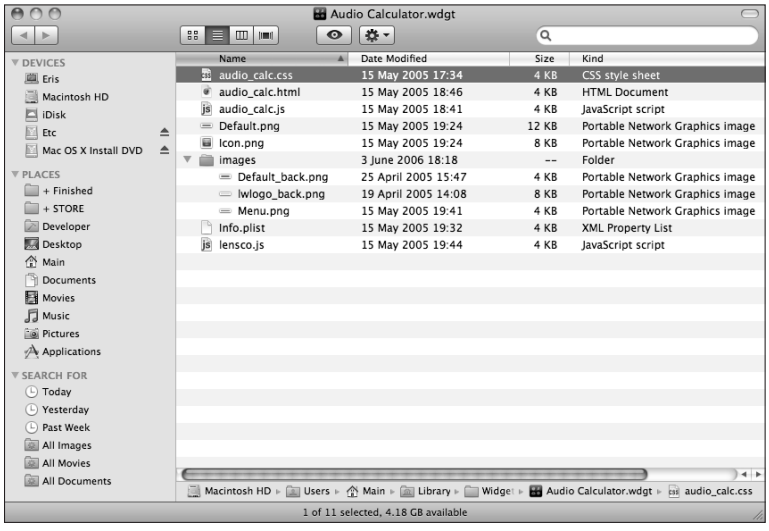
Although widgets can have any dimensions, 640 × 480 is a realistic maximum size. Widgets should be simple and should be able to share the Dashboard "desktop" with other widgets. Do not design them as complex independent applications with an extended feature set. Widgets should do one or two things elegantly in a small screen area.

Disassembling widgets manually

The `.wdgt` file type is a plain zipped file. Widgets are stored in two locations. The default Apple widgets are in `<Macintosh HD>/Library/Widgets`. User-downloaded widgets are installed in `<user name>/Library/Widgets`. To view the contents of a widget, select it in Finder, right-click it, and choose Show Package Contents. You'll see a directory listing similar to Figure A.3.

Figure A.3

Disassembling a widget reveals its component files. All files inside a widget can be opened, copied, and edited.



Widgets are built from a collection of standard components, with optional additions. The list for a minimal widget includes:

- An `.html` file with basic Web elements and content placeholders within HTML `div` tags.
- A `.css` (Cascading Style Sheet) file that controls the styling and position of the `div` elements.
- At least one `.js` (JavaScript) file with the widget's code.
- One `Default.png` file with the widget's background graphics. The default file can have any reasonable dimensions and may include alpha transparency. The D in "Default" must be capitalized.

- An `Icon.png` file. The icon file is the widget's icon in Dashboard's dock. It has a square aspect ratio and should be 85 × 85 pixels. Content should be limited to an area of 74 × 74 pixels. The I in "Icon" must be capitalized.
- An `info.plist` file with bundle and version information, including the widget's name, identifier, and dimensions. This file also names the .html and .js files in the widget. Dashboard reads these references when the widget loads.
- A `version.plist` file with version numbering information, for Dashboard's internal use only.

Optionally, widgets can be localized with separate `<language_name>.lproj` folders, each of which includes an `infoPlist.strings` file with UTF-8 (Unicode Transformation Format-8) strings for each supported language. Many widgets include an extra `/Images` folder that holds graphics files.

A key feature of widgets is that *the components are unprotected.* Code, HTML, and graphics can be opened in any editor, copied, and changed by anyone with a basic understanding of widget technology, as shown in Figure A.4. Mac users typically lack the skills to disassemble widgets, but as a developer you must assume that other developers will be able to read and copy your code.

CAUTION

Although the JavaScript files in a widget can be read by anyone, most widgets aren't formally open sourced and are rarely developed collaboratively with an open license. You should consider all code private and proprietary unless the developer includes a license explicitly stating otherwise. If you release a widget that includes a direct copy of code by another developer without his permission, the other developer may well find out — and there might be consequences.

Many popular widgets provide front-ends for Web services such as Twitter, bit.ly, and Facebook. Although you do not have legal rights to reuse code from other widgets directly, widget scripts can be an excellent way to discover how other developers use public application programming interfaces (APIs). Even if you don't plan to develop widgets, this is one of the most useful features of widget technology. Attempting to master an API without good sample code can be difficult. Widgets can provide you with this code.

NOTE

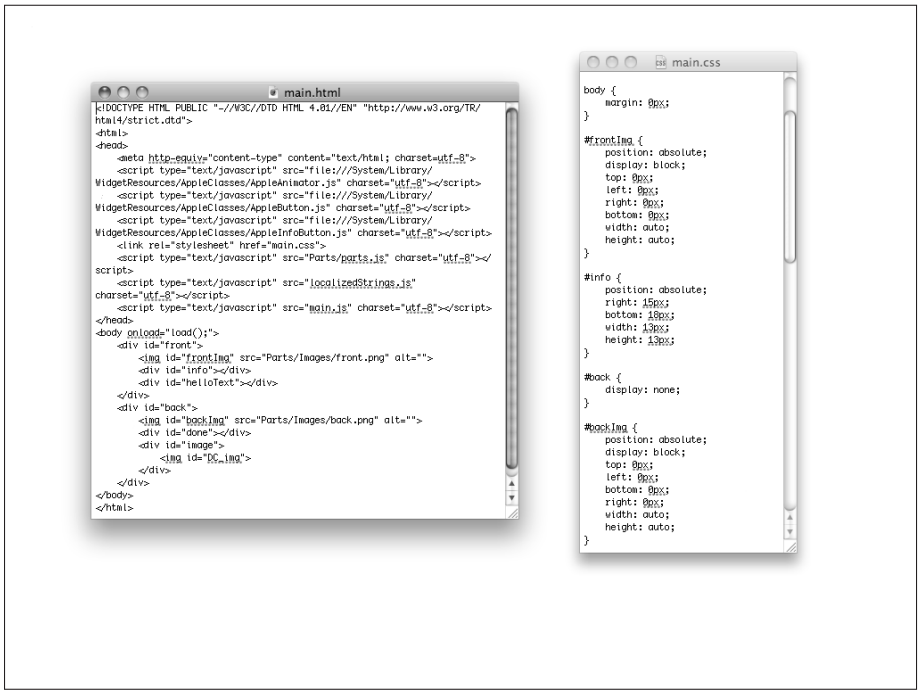
Although widgets are written in JavaScript and not Objective-C, these languages are similar enough to be easy to translate in either direction — even if you're not perfectly fluent in both.

Assembling widgets manually

You can assemble a minimal widget by hand using a text editor to create the HTML, CSS, and JavaScript, an image editor to create the graphics, and the OS X plist editor to assemble the plist files. This is a practical option for very simple widgets, and it offers the advantage of the smallest possible bundle sizes. If your widget includes the essential elements and is zipped into a folder and renamed with the `.wdgt` extension, it can be installed in Dashboard.

Figure A.4

A widget's HTML and CSS files include standard Web content. Optionally, advanced widgets can include a standard Apple widget code library that manages animations and button events.



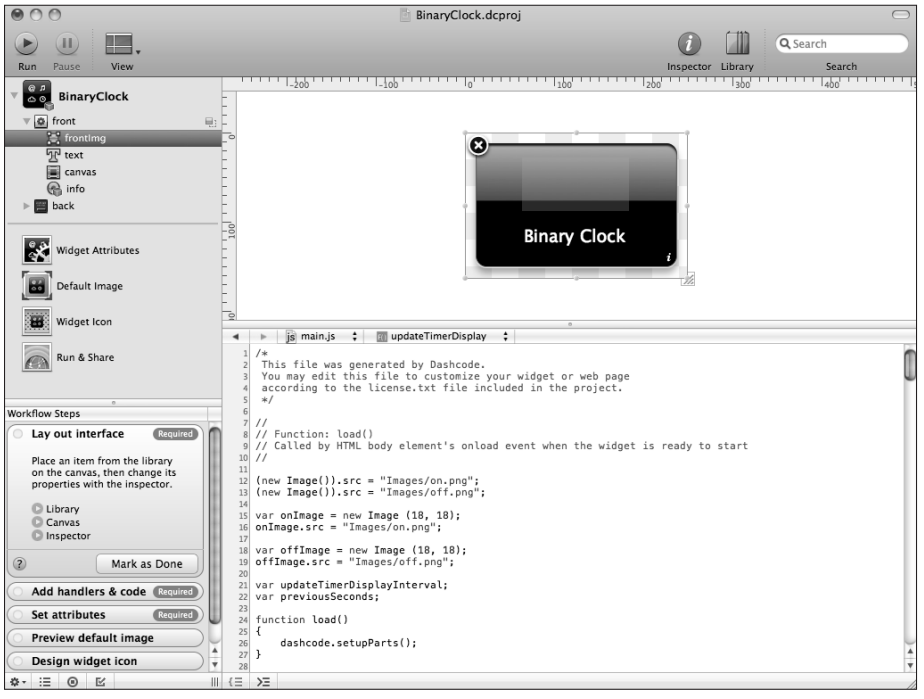
There are many disadvantages to manual assembly; debugging is almost impossible, functionality is limited, and graphic design is laborious. More fundamentally, widgets use Apple-specific CSS and HTML extensions that are not part of the W3C standard. To include these features in your projects, you must use *Dashcode* — the widget and Web kit editor bundled with the OS X SDK.

Introducing Dashcode

Dashcode, shown in Figure A.5, is a sophisticated tool that combines many of the features of Xcode and Interface Builder. It includes a range of prewritten widget templates, an environment for graphic design and prototyping, a Library of widget parts, and sample code snippets that support them. It also includes a runtime environment for widget testing and debugging, and an import feature that can load any existing widget and make its components available for editing.

Figure A.5

Dashcode's window areas are similar to Xcode's panes, with parts rather than classes at the top left, resources under the parts window, and a separate source code view.



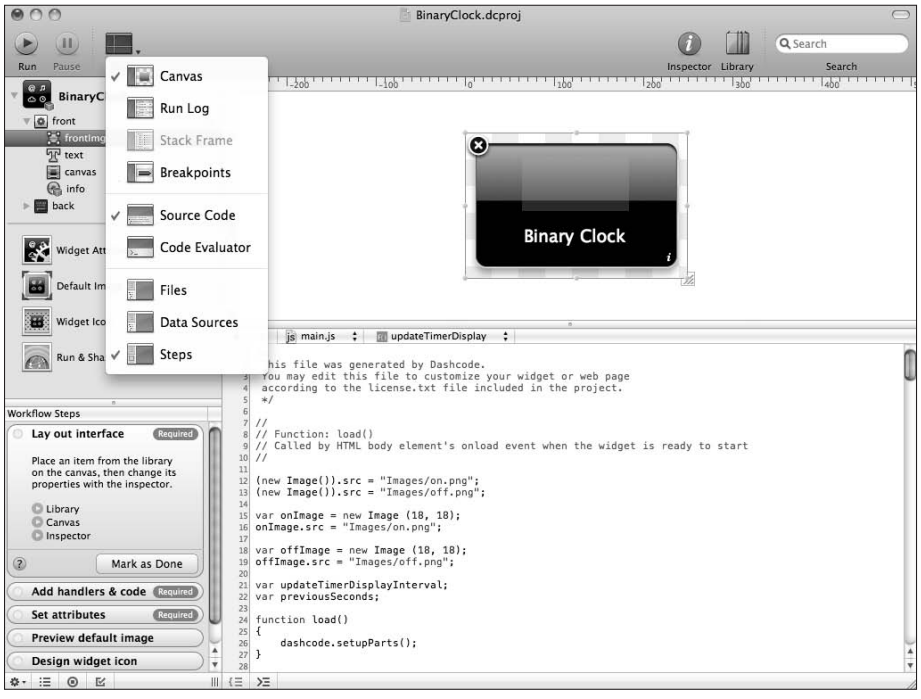
You can find Dashcode in `/Developer/Applications`. If you are familiar with Xcode and Interface Builder, you should find it easy to work in Dashcode. The arrangement of features and windows is similar to Xcode's, and many of the idioms and concepts used in Dashcode are recognizable. However, there are important differences. You can make development feel simpler and go smoother by familiarizing yourself with them before beginning your first widget.

Exploring the Dashcode interface

Parts of the Dashcode interface, illustrated in Figure A.6, are relatively self-explanatory, such as the graphic preview window used to design and lay out widget graphics and the source code window. Unfortunately, two other key elements described below are presented in a much less intuitive way. Dashcode can be difficult to work with until you identify and master these features.

Figure A.6

The View drop-down menu holds some of Dashcode's useful secret features and has no equivalent in Xcode.



The View drop-down menu at the top left of the Dashcode window is Dashcode's master view selector. Use it to select the contents that appear in each of Dashcode's main panes; for example, you can display a Run Log in the pane usually used for graphic editing. You can also replace the source code pane with a Code Evaluator for instant calculations or for direct interaction with a widget's part properties.

The row of icons at the bottom left of the Dashboard window, shown in Figure A.7, is an alternative key view selector. The icons are labeled with minimal tool tips and are easy to miss, but they are a critical part of the Dashboard interface.

Figure A.7

The bottom left icon series selects the elements that appear in the lower-left pane. The Files view is usually the most useful.

From left to right, the icons are as follows:

- **File Controller.** Use this icon to add new files to the project.
- **Files.** Use this icon to show and hide a Files pane, with a list of the files in a project. This is one of the key icons in Dashcode, and you should familiarize yourself with it before you begin editing.
- **Data Sources.** Use this icon to add a data source — a URL that points to a JSON (JavaScript Object Notation) or XML (extensible markup language) feed — to the project. A full discussion of JSON and XML is beyond the scope of this book. You can find an introduction at `www.json.org`.
- **Workflow.** Use this icon to display the Workflow Steps pane. (This is one of the less useful features of Dashcode.)

TIP

When it first loads, Dashcode displays the Workflow Steps pane. It's likely you'll want to switch this view so it shows the project file list. It can be helpful to get into the habit of clicking the Files icon to replace the Workflow Steps list as soon as Dashcode loads.

Working with parts and the Library

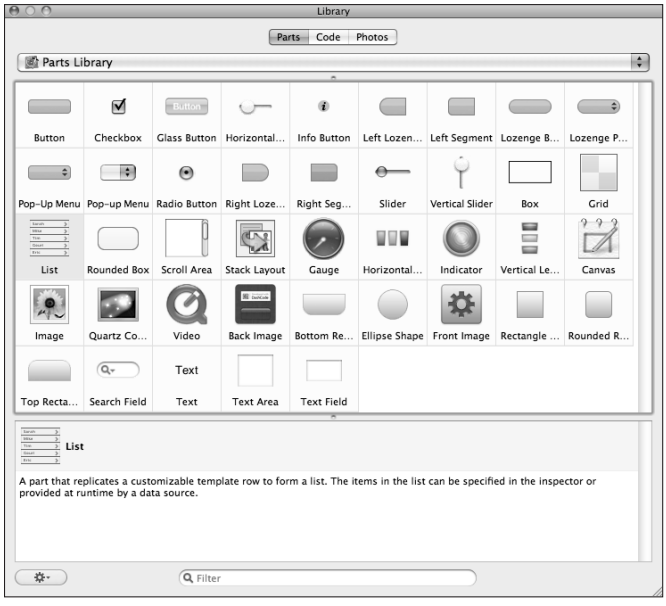
Choose Window ➪ Show Library to reveal the Parts list, shown in Figure A.8. You can also click the Library icon near the top right of the Dashcode window. Parts are similar to Interface Builder's objects, but can't be subclassed. They are designed to be dropped into a widget design as is, and then linked to event handlers in the code. To add a part to a widget, drag and drop it onto the widget's front or back image.

The Library is complex and rich, and it includes special-effect parts such as a Quartz Composer frame, a navigation stack, and a customizable list. Although widgets should be simple, they do not have to be crude. You can use parts in an adventurous way to create unique and original widgets that offer unusual user experiences.

Click the Code tab in the Library window to see a list of code snippets. You can copy this code, paste it into your widget's source code window, and modify it as needed. Code and parts are independent — the code snippets can be used anywhere in your script. They are not designed to work with any particular part.

Figure A.8

Dashcode's parts list includes 41 widget objects that you can add to a background frame. You can also view code snippets and click Photos in the top tab to include a custom photo or video in a widget.



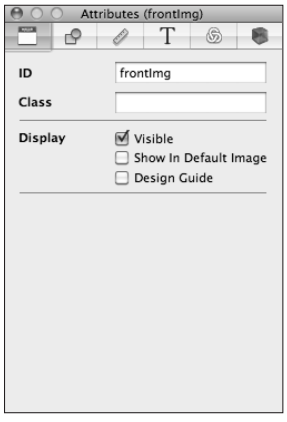
Using the Inspector

Choose Window ➪ Show Inspector to reveal the Inspector. You can also click the Inspector icon near the top right of the Dashcode window, as shown in Figure A.9. The Inspector window displays and edits the properties of parts and objects. Properties include settings that control the visible appearance of a part and also define data links and event triggers to widget code and external sources of information. The six panes are:

- **Attributes.** Holds information about the ID and class of each part and the settings for localization.
- **Fill & Stroke.** Sets fill and stroke effects for a part, and includes graphic effects such as rounded corners, gradient fills, glass stylings, and recessed bevels.
- **Metrics.** Controls the position and size of the part, its resizing options, and its size constraints.
- **Text.** Sets the font, color, shadow, font size, and alignment for parts that use text strings.
- **Bindings.** Binds part properties to a data source, with optional transformations.
- **Behaviors.** Displays a list of part events, with optional links to handler functions in the code.

Figure A.9

Although the Attributes window in the Inspector appears simple, the ID field is one of the key properties for each part included in a widget.



TIP

As in Interface Builder, the Library and the Inspector windows often disappear behind other objects. You can reveal them by choosing Window ➪ Bring All To Front.

Building a Widget in Dashcode

To create a new widget, choose File ➪ New to display Dashcode's standard list of templates, shown in Figure A.10. You can see a description of each template by clicking it. Many of the templates are unlikely to serve your needs, but the RSS (Really Simple Syndication) and Custom templates are reliably useful. For a minimal template with a rounded box with a gradient fill and a single line of text, select Custom.

Creating widget graphics

To modify the look of the background graphics, select the `frontimg` part in the Parts pane at the top left of the Dashcode window. You may need to click the triangle next to the `front` part to reveal its contents.

Dashcode includes glass-effect and gradient tools in the Fill & Stroke pane in Inspector. Experiment with the Gradient fill effect in the Style pane, adding optional corner roundness, reflection, and opacity. In the Effects pane, you can add Glass and Recess — inset — effects.

Figure A.10

Unfortunately there's no template for displaying or updating an image from a Web source. Adobe Flash elements are not supported at all.

TIP

You can use the graphics editor as a convenient tool for creating glass-effect graphics for other projects. When you save a widget project, image files are included in the project folder. You can read these files with any image editor. This is a convenient and fast way to create custom glassy buttons for any iPhone or OS X project.

Using JavaScript in widgets

JavaScript is a complex object-oriented language with many features, but you can create working widgets without becoming a JavaScript expert. A full description of JavaScript is outside the scope of this book, but you can use the hints in this section to get started with widget programming.

Understanding DOM object access

JavaScript uses the Document Object Model (DOM) to manage object access. Every part in a widget has its own distinct ID, and you can access the object's properties by referencing the ID. For example, to access the front view of a widget use

```
var front = document.getElementById("front");
```

You can then access the front object's properties using standard dot syntax. To set a custom ID for a part, type the ID name into the ID field in the Attributes pane in the Inspector. Each part includes a fixed list of accessible properties via DOM. Some, but unfortunately not quite all, properties are listed and described in the Dashcode User Guide. To view the User Guide, choose Help➪Dashcode User Guide.

DOM is an essential part of widget management. Use it to change basic properties, such as text strings, or to swap objects in and out of visibility. Some parts, such as the Canvas, have more sophisticated interfaces; for example, you can draw into a Canvas using JavaScript versions of standard canvas functions.

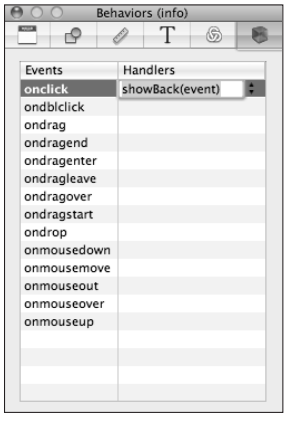
Working with Events and Handlers

As in an Objective-C application, visible parts in a widget can be connected to handler functions that are triggered when the user clicks with a mouse, drags an object, and so on. For example, a part's `onclick` event is triggered when the user clicks the object. The list of events for each object or part is visible in the Behaviors pane in the Inspector.

To assign a Handler, select an event and double-click in the area under the Handler header to its right. An empty box appears. To finish the assignment, type the name of the handler routine into the box, as shown in Figure A.11. There is no hinting, and no menu or list of possible handlers — you must type the name in manually. You can check that your assignment is correct by clicking the arrow to the right of the handler name; the corresponding code routine is highlighted in light blue.

Figure A.11

Because the Handler field is dumb, you must type in the function name manually. If you get it wrong, the Event won't trigger correctly.



Working with widget events

Widget templates typically include code to create a spin effect triggered by clicking the Info part, typically in the `showFront(event)` and `showRear(event)` routines. You won't usually need to change this code, but you can add parts to the reverse of the widget with custom handlers of your own design — for example, to ask for a donation with a link to your Web site.

At the bottom of every template is a boilerplate conditional:

```
if (window.widget) {
 widget.onremove = remove;
 widget.onhide = hide;
 widget.onshow = show;
 widget.onsync = sync;
}
```

This assigns custom handler functions — `hide`, `remove`, `show`, `sync` — to standard widget events that are triggered when Dashcode changes state. Stubs for each custom handler are included at the top of the template scripts. You can add extra code to these stubs to enable or disable widget features as needed for each state.

Working with time

Many widgets use timers, but JavaScript lacks a simple drop-in equivalent for `NSTimer`. To create a timer, use:

```
aTimer = setInterval(aFunctionCalledWhenTheTimerExpires,
   aTimeInMilliseconds);
```

This calls `aFunctionCalledWhenTheTimerExpires` after `aTimeInMilliseconds`. Timer functions can call themselves for repeated timing events. To stop a timer and delete it, use

```
clearInterval(aTimer);
aTimer = null;
```

To get the current time, use the `Date()` function:

```
var currentTime = new Date();
var currentHours = currentTime.getMinutes();
var currentSeconds = currentTime.getSeconds();
```

Loading images

Images can be preloaded for speed with

```
(new Image()).src = "/Images/anImage.png";
var anImage = new Image (width, height);
anImage.src = "/Images/anImage.png";
```

You can then write or link `anImage` into any part in the widget or draw it to a canvas.

Deploying and importing widgets

To deploy a widget, choose File➪Deploy. This compresses it into a Zip file, renames it with the `.wdgt` extension, copies it to your Mac's widgets directory, and loads it into Dashboard. You can confirm or cancel installation in Dashboard, as shown in Figure A.12. To deploy the widget elsewhere, for example on a Web site, navigate to the widgets directory and copy or move the new `.wdgt` file, as needed.

Figure A.12

Deploying a widget adds it to Dashboard automatically. You can choose to delete the new widget if you change your mind.

To import and disassemble a third-party widget, copy it or install it to your Mac, and select File➪Import Widget in Dashcode. Navigate to the `.wdgt` file and click Open. Dashcode uncompresses the widget and loads its contents, ready for editing.

NOTE

You can find code and graphics for a sample Binary Clock widget on the Web site at `www.wiley.com/go/cocoadevref`.

Summary

In this appendix you learned about widgets and discovered how to install them and work with them in Dashboard. You were introduced to the Dashcode widget editing and creation tool, and you were given a brief tour of some of the more useful elements of JavaScript you can use in widget creation. Finally, you discovered how to deploy a completed widget and how to disassemble an existing widget created by a different developer.

Maximizing Productivity and Avoiding Errors

Software development can be rewarding, but it can also be frustrating. Most developers have the experience of code that refuses to work for days or weeks as various possible solutions are tried and discarded.

Many solo projects are improvised rather than developed in a structured way. The reliability of formal project management in commercial software engineering is inconsistent, and no technique or process can guarantee that a project will be completed on time or on budget. But as a solo developer, you can use simple checklists and hold discussions with other developers to make your projects less stressful, more productive, and less time consuming.

Managing Projects Successfully

Three simple project management techniques can dramatically improve your productivity:

- **Use source code management and version control.** Even the simplest projects can benefit from the version control features built into Xcode and introduced in Chapter 3 of this book. You can use them to roll back major coding or design mistakes, without losing other work.
- **Produce reference documentation as you go.** Reference documentation, including notes and object diagrams, takes almost no time to produce but is a tested and proven way to clarify design ideas. If your ideas are clear, your code is more likely to be correct, and the application is more likely to work as you want it to. With a free tool such as doxygen — from `www.doxygen.org` and illustrated in Figure B.1 — you can build reference documentation as you code by writing it directly into comment fields.

In This Appendix

Managing projects successfully

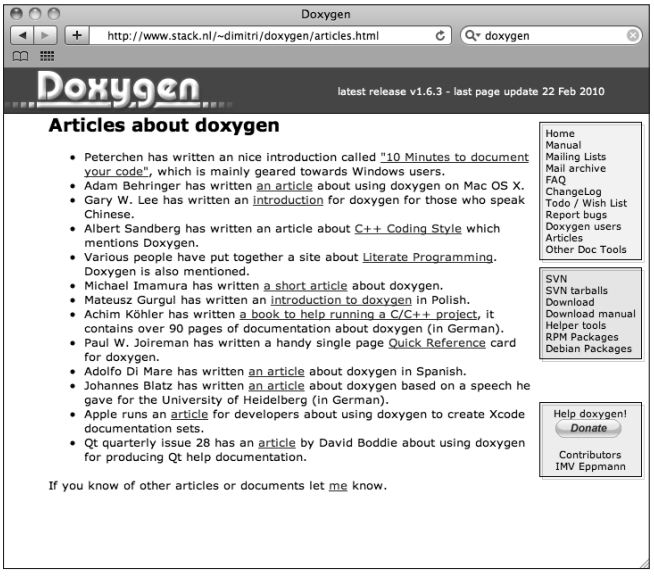
Getting help

Solving impossible problems

- **Eliminate compiler errors with a checklist of standard solutions.** Many errors are caused by a small number of simple mistakes. You can eliminate them by working through a formal error checklist.

Figure B.1

Doxygen was developed to simplify documentation, and it can translate comments and notes into a variety of document formats.



Solving common problems

Tables B.1 and B.2 list some of the more common syntax and build errors. If you find yourself making other repetitive errors, add them to a list. Many developers repeat the same mistakes; if you list yours, it can help condition you to avoid them.

Table B.1 Common Syntax Errors

Error type	*Description*
Missing semicolon	This is common, but it's easy to spot. Xcode flags it on the next line with an `Expected ';'` ... error.
Missing or extra curly bracket	Xcode's bracket balancing doesn't always catch this error, which typically causes multiple error messages in a single file. Look for `Expected declaration or statement at end of input` at the end of a file and work back to find the first undeclared method. The missing bracket is usually in the method above it.
Missing or extra square bracket	This is another simple mistake that can cause multiple error messages. Look for `Expected ']'` ... on the line with the missing bracket.
Missing or extra capitalization	This is one of the most common errors in Objective-C, and it can be very easy to miss. Look for properties, variables, and methods that appear correct but aren't recognized by the compiler. Triple-check capitalization around the error. Check again wherever the word `view`, or one of its compounds, such as `Superview`, appears; they're often wrongly capitalized in method names.
Missing or extra colon in a method name	Remember that `aMethod:` and `aMethod` are unrelated and distinct. Avoid declaring methods with easily confused names. Look for crashes and console messages warning that a selector wasn't found.
Missing or extra parameters in method calls	A popular mistake is to create a custom method based on a system method with added extra parameters in a class prototype, and then call the original unaltered method in the class implementation. Avoid this by creating a formal category.
`[anObject = aMethod[etc]];`	Assignments within square brackets are usually syntax errors. Avoid them even where they're nominally correct by breaking out the assignment onto a separate line.
`if (a = b)` ...	Everyone does this — usually more than once.
`If (aString == bString)` ...	Use `isEqual`, or one of its typed equivalents, when comparing objects. The standard comparison compiles and runs without errors, but compares pointer addresses instead of object content.
Incomplete subproperty lists	Common for complex object with many subproperties — for example, `window.width` instead of `window.frame.size.width`. Look for the famous `Request for member 'property' in something not a structure or union` message.

TIP

You can avoid subproperty errors by creating a reference list of common objects and their subproperties. It can be difficult to remember complete subproperty lists — a reference can save you hours of time.

Table B.2 Common Build and Design Errors

Error type	*Description*
Forgotten links or bindings in Interface Builder	When you add new features to a view, don't forget to link their outlets and actions in Interface Builder or to add bindings — and don't forget to save the nib file after editing.
Missing delegate features	If a class feature isn't working, check for and implement any compulsory delegate methods.
Missing focus or First Responder status	Some App Kit classes may not work without window focus or First Responder status. Check that you're handling these features correctly.
Broken or misleading Xcode links	If you import files to a project or add classes to it, remember to set the Path Type to `Relative to Enclosing Group`. If you leave it set to `Default` you may find yourself editing files across a mix of project folders. Xcode doesn't warn you when you do this.

TIP

Simple mistakes can waste time, but don't let them sap your confidence or raise your stress levels. Even experienced developers regularly make basic errors.

Managing classes and files

Xcode doesn't include automated support to help you manage classes, properties, and files. Managing them manually makes them error prone. Use these checklists to avoid the most common errors. The steps may seem obvious, but it's easy to miss them when you're concentrating on some other aspect of a project.

Including other classes in a class header

- Add `#import` directives for other classes that are referenced.
- Add `@class` directives for each class.
- Add `#import` directives for framework headers, as needed.

Creating a new subclass

- After you create the file, change the subclass name in the interface.
- Add any delegate protocols, as needed.

Adding new properties or methods

- In the header file, add the property or method between the prototype angle brackets.
- Add a property declaration in the area below the angle brackets.
- In the implementation file, synthesize the new property or add a method stub.
- In Interface Builder, add new links and bindings.

TIP

If you rename a method or property, use Xcode's global Replace All feature to rename every instance in the code. Don't try to rename features manually — you're unlikely to spot every instance. To use the Replace All feature, choose Edit ➪ Find ➪ Replace All.

Working in Interface Builder

- Reload classes whenever you add or edit a class.
- Create links and add bindings as soon they're needed — you may forget later.
- Check that links are connected to the correct object; for example, to First Responder rather than File's Owner, as needed.
- Save the nib file before building.

Adding frameworks

- Remember to add frameworks to the project when you use framework-specific features.
- Remember to `#import` the framework headers.

Getting Help

With an almost endless proliferation of blogs, message boards, discussion groups, tutorials, and worked solutions, the Internet is the most useful developer resource. Comments from other developers can help shift a stuck project, and complete worked solutions may be available online. Helpfulness varies; a minority of developers may be more interested in telling you to RTM, and most message boards have a selection of posted problems with no comments or replies. While the Internet is sometimes noisy and never infallible, it remains the go-to source for help, feedback, and solutions for developers at every level of experience.

TIP

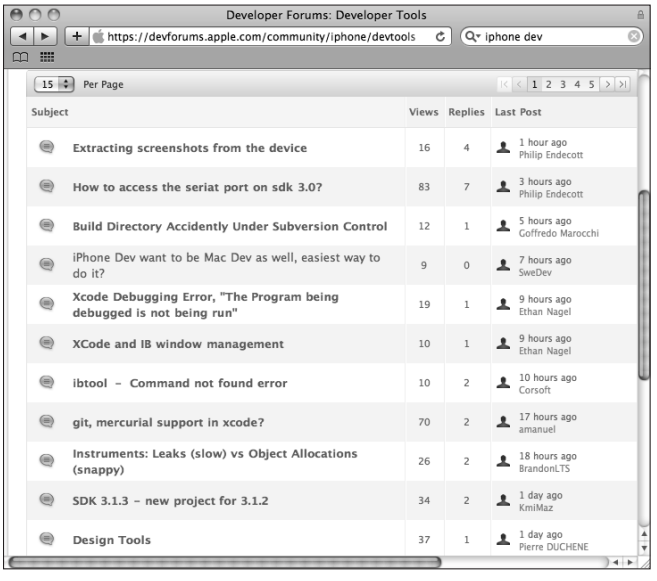
Searching for "OS X" or even "Cocoa" brings up many, many pages of user-level information, which is unlikely to be useful. To target searches more effectively, add "dev" or "developer" to the start of your search strings.

Resources include:

- **Blogs.** Many developers keep blogs, posting solved problems and sample code snippets. The iPhone developer community is particularly active, but you can also find Cocoa and general OS X blog content.
- **Official developer forums.** Apple's developer forums, illustrated in Figure B.2, are a useful resource, but they are only accessible to developers who are enrolled in an official developer program. The official developer community is still growing. For the moment, the forums are best considered a useful extra rather than a definitive source for solutions.

Figure B.2

Apple's forums remain slightly experimental, but they are likely to become more popular and useful as the iPhone and Mac developer programs continue to grow.



- **Unofficial developer forums.** There are tens or even hundreds of unofficial forums, and new forums appear regularly. CocoaDev (`www.cocoadev.com`), shown in Figure B.3, is a popular center, with an excellent collection of open-source projects. Don't limit yourself to a single forum — you'll typically get better results from a general Internet-wide topic search than by following postings in a single location.
- **Mailing lists.** Although eclipsed by forums and blogs, lists continue to offer useful help and content. The Apple cocoa-dev list, shown in Figure B.4, has been running for ten years now; see `http://lists.apple.com/mailman/listinfo/cocoa-dev` for details.
- **Usenet/Google groups.** Usenet/Google groups no longer have the prominence they once did and can be prone to spam, but they continue to be helpful. iPhone groups are more prominent than Cocoa groups, but cocoa-dev, shown in Figure B.5, is a useful resource for Cocoa help.

Figure B.3

CocoaDev is a popular and useful online forum among the many other popular and useful forums available online.

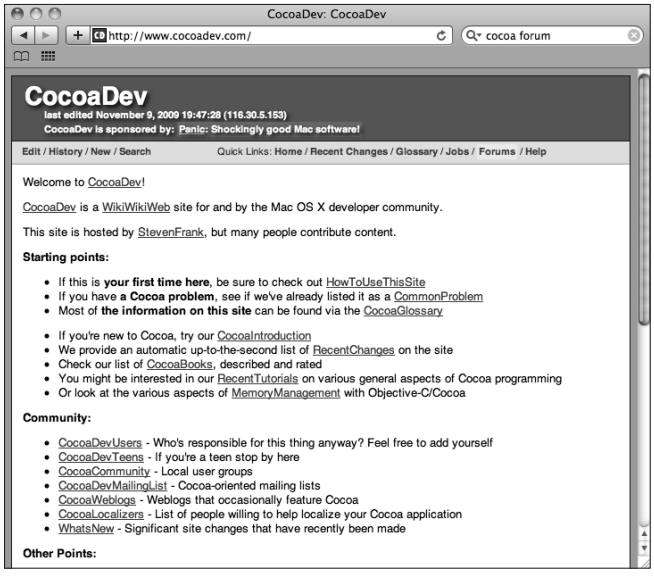
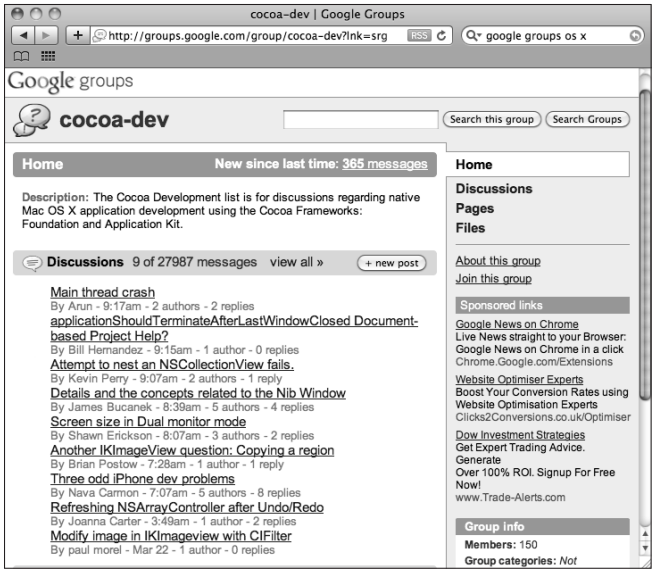


Figure B.4

According to Apple, the first archived post to the cocoa-dev list was made in 1969. Most posts are more recent and often more useful.

Figure B.5

Google groups have moved closer to the Yahoo! groups model and away from their Usenet origins. You can use the Google groups pages to find archives for both types.



Solving Impossible Problems

If you spend more than a couple of days on a problem without making any progress, and you've exhausted the official documentation and all possible Internet resources, you may be dealing with an Impossible Problem. Not all Impossible Problems are truly impossible — sometimes sustained effort can lead to a simple solution. But they can divert a project into an unproductive dead end, and waste time that might be spent more usefully.

When trying to solve an Impossible Problem, it can be useful to step back and consider project strategy rather than coding tactics. Strategy defines design goals, and tactics define how they're implemented. Many Impossible Problems are tactical: thinking strategically can sometimes help you sidestep them and keep the project moving. Consider these questions:

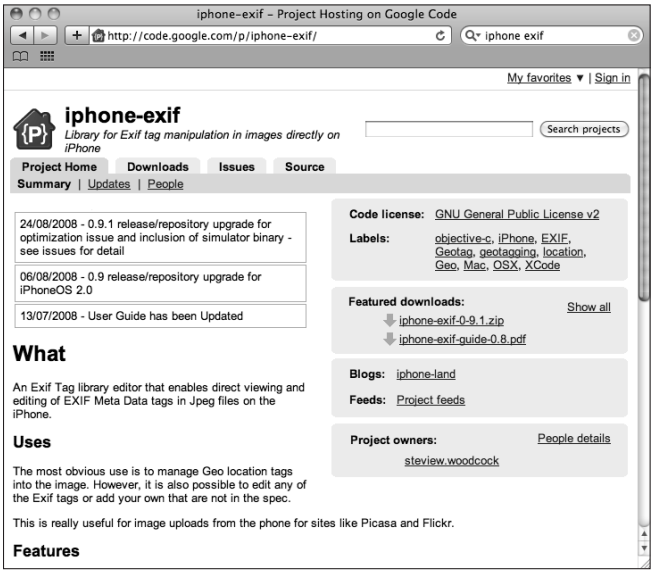
- **Does the problem need to be solved?** Impossible Problems often develop a momentum of their own, monopolizing your attention. You can minimize this by considering the problem from a user's point of view. How essential is the feature that's being implemented? If a user won't miss it if it's not included, leave it out or move it to the feature wish list for later versions.
- **Does the problem need to be solved now?** Leaving a problem unsolved can feel very unsatisfying, but it may be the only way to move a project forward. Putting a problem to one side may move a project in a new direction, making the problem irrelevant or solving it in an unexpected way.
- **Is there another way to achieve the same result?** When the final aim is a specific user benefit, there may be a simpler way to offer it.
- **Can the problem be solved?** Some problems are truly impossible for either formal or informal reasons; for example, they may be mathematically intractable, or they may require too many processor cycles.
- **Has the problem already been solved?** Many problems are easy to solve in theory, but too difficult or time-consuming to code in practice. A satisfying solution is to use code or a framework created by another developer. Because so much code is either formally open sourced or available online, common problems are likely to have at least one ready-made solution, such as the iPhone Exif handling framework, shown in Figure B.6, which is a solution for a complex problem that would otherwise require days or weeks of independent effort.

TIP

Open source code may include licensing restrictions; for example, you may be expected to open source your own project if you include code released under the General Public License (GPL). There is a distinction between using code as is and reverse engineering it. Taking concepts from an existing solution without copying and pasting code is one possible way to avoid licensing restrictions.

Figure B.6

Code repositories such as Source Forge (`http://sourceforge.net`), GitHub (`http://github.com`), and Google Code (`http://code.google.com`) have hundreds of prewritten frameworks you can drop into a project, such as the EXIF framework shown here.



Summary

In this appendix you were introduced to some simple but effective project management techniques, you explored checklists of common errors with likely solutions, and you looked at resources for online help. Finally, you were given hints for keeping a project moving when faced with problems that appear impossible — but often aren't.

Index

SPECIAL CHARACTERS

!= conditional, 310
* (asterisk), 37–38
: (colon), 126, 567
; (semicolon), 111, 567
@ character, 38
[ ] (square brackets), 40, 127, 567
^ (caret), 277, 278
{ } (curly brackets), 111, 141, 277, 567
‘ (apostrophe), 277

A

Ableton’s Live, 9–10
aBlock, 281
Abstract classes, 113
accelerometer, 497
accessors, 97
actions
    creating direct links to, 191–192
    defining in implementation, 188
    defining in interface, 185–187
    description, 184
    placing, 202–203
ActionScript, 48, 53
Active Executable, 525
Active Screen Corner, 548
Activity Monitor, 483
ad banners, 550
ad hoc distribution, 540
addAnimation: method, 452
addOne: method, 219–220
addOperation:, 281
Address Book, 96, 117
addSubview: method, 413, 435
Adobe Creative Suite, 9
aesthetics in design, 19
aFile.ext, 236
aFrame, 129
aFunctionCalledWhenTheTimerExpires, 561
AKey, 342
Align Horizontal Center button, 182
Alignment button, 183
allMatches, 237
alloc message, 103
alloc method, 37
alloc/init code, 154, 410, 413
aLongMethod, 269
alpha, 426
Amazon, 11, 247
aMethod, 215
anImage, 561
AnImaginaryProtocol, 135
animatable filter, 453–456
animated view swaps, creating, 534–535
animationCurve, 535
animationDidEnd:, 446, 447, 458
animationDidStart:, 446, 447, 458
animations
    creating property animation code, 442
    creating timer for, 441
    overview, 439
    types, 440
    using animator
        creating path animation, 450–452
        creating simple proxy animation, 443–445
        creating values/keytimes animation, 449
        customizing animation object, 446–449
        setting duration, 446
        timing functions, 448–449
    using CALayer
        animatable filter, 453–456
        animating filter, 456–458
        using layers for animation, 452–453
    using drawRect:, 443
    using OpenGL, 458–463
animator
    animatable filter, 453–456
    animating filter, 456–458
    creating path animation, 450–452
    creating simple proxy animation, 443–445
    creating values/keytimes animation, 449
    customized animation, 440
    customizing animation object, 446–449
    default animation, 440
    setting duration, 446
    timing functions, 448–449
anInstance, 99
anObject, 215
anotherInstance, 99

aNotification, 139
Ansca Corona, 76
aParameter, 112
Aperture, 4
APIs (application programming interface)
creating asynchronous Web requests, 258–260
creating long URL, 252–253
creating XML requests, 255–256
parsing XML returns as text, 257–258
selecting XML format, 256–257
sending URL request, 253–255
using, 247–248
apostrophe ('), 277
App Delegate, 304
.app file, 48, 88
App ID, 538
App Store
certification process, 538–540
income territories, 499
online assistant, 539–540
packaging apps for, 541–542
permissions, 538–540
ratings grid, 543
reasons for refusing apps, 536–537
selling in, 82, 536–537
uploading apps to, 542–543
app switching, 498
AppDelegate suffix, 118
appendBezierPath:, 419
appendBezierPathWithArcFromPoint:toPoint:, 419
appendBezierPathWithGlyph:InFont:, 419
appendBezierPathWithPoints:, 419
AppKiDo, 57, 59
Apple Developer Convention, 80
Apple Developer Forums, 80
Apple Human Interface Guidelines, 20, 60
Apple ID, 80
Application Delegate
code for, 344
key features, 344
link destinations, 189
application delegates
converting notifications to, 230–231
description, 118–121
errors, 568
messages, 138–139
methods, 139–142
notifications and, 230–231
receiving message from OS X with, 134–137
Application Framework layer, 22
application frameworks, 24–25
Application Kit Functions Reference, 131
application programming interface. *See* APIs
applicationDidChangeScreenParameters:, 138
applicationDidFinishLaunching:
auto-refresh, 373
delegation, 138
description, 120
event logging, 467
Key-Value Observing, 328
object status, 33
Protocol Reference, 137
receiving message from OS X, 134
timer method, 196
applications
adding navigation and control features to, 511–514
application delegates, 118–121
application frameworks, 24–25
building simple application
adding view controller subclasses, 526
creating animated view swap, 534–535
creating new project in Xcode, 525
creating views, 528–532
handling events with protocol messaging, 532–534
implementing view controllers, 526–528
business model
making money from iPad development, 501–502
making money from iPhone and iPod touch development, 499
Carbon, 25
C-language features, 129–132
class references, 121–125
code layers, 25
Code Sense, 126–127
Core Data
adding code, 363–364
adding entity, 354
adding properties, 355–356
building the application, 361–363
creating fetch requests, 367
creating relationships, 356–359
displaying search results, 370–373
generating user interface, 359–361
modifying nib, 363–364

project template, 352–353
sort descriptors, 368
to-many fetch requests, 368–370
using predicates, 367–368
creating category on NSWindow, 149–150
custom switching, 514
delegate, 118–121
design goals
constrained data sharing, 504
full-screen operation, 503
instant on, instant off, 503
loss of focus without notice, 503
no explicit data save and restore, 504
orientation sensing, 509–511
restricted battery life, 504
developing features across layers, 25
elements, 22–23, 117–118
expanding search to related classes, 133–134
frameworks, 27–29
handling touch events, 514–515
iOs vs. OS X apps, 30, 495–496
iPad apps, 30
iPhone apps, 30
layers, 22–23, 27–29
vs. Mac apps, 13
market, 499
master/root view controller, 507
multiple classes, 128
prices, 499
profiting from, 12–14
receiving message from OS X delegate
delegate methods, 139–142
overview, 134
protocol, 135–136
Protocol Reference page, 137–138
receiving message from OS X with NSResponder, 142–144
selling in App Store
certification process, 538–540
income territories, 499
online assistant, 539–540
packaging apps for, 541–542
permissions, 538–540
ratings grid, 543
reasons for refusing apps, 536–537
selling in, 82, 536–537
uploading apps to, 542–543
signatures, 125–126
subclassing NSWindow, 144–149
"toll-free bridged" layers, 25
uploading to App Store, 542–543
user experience, 23–24
workflow, 29
X-code only policy for, 76
Applications folder, 82
applicationShouldTerminate:, 138
applicationWillHide:, 139, 141
applicationWillUnhide:, 138, 141–142
Aqua
default graphics features, 24
design guidelines, 20–21
user experience, 23–24
using with Cocoa, 21–22
archiving, 44, 214, 300–301
argc parameter, 116
argv parameter, 116
arrangedObjects, 338
ArrayControllerSubclass, 333
arrays, without using NSArray, 295
articles, 61–62
Assembler, 54
assignments, KVO-compliant, 226–227
asterisk (*), 37–38
aStructWrapper, 291
asynchronous requests, 258–260
aThread, 269
atomically parameter, 238
attributed strings
drawing text on path, 380
loading, 379–380
saving, 379–380
using, 380
attributes
description, 180–181
document, 379
text, 379
Attributes pane, 557
Attributes tab, Inspector window, 164
Audio Toolbox, 27
autocompletion feature, 126–127, 237–238
Automator, 10
autorotation, 509
AViewController.m file, 531
AViewController.xib, 526–528
awakeFromNib, 408, 425, 448, 526
aWindow, 36

B

backgroundFilters, 453
BackgroundTask class, 272
BaseInBaseOut, 448
battery life, 504
beginAnimations: method, 535
Behaviors pane, 557
Berners-Lee, Tim, 7
beta updates, 81
Bezier paths, 419
bindings. *See also* preferences
  creating, 315–320
  description, 163
  errors, 568
  facts and fictions, 314
  formatters, 328–330
  implementing preferences with, 340–346
    creating application with, 343–346
    initializing, 341–342
    reading values, 342–343
    setting keys, 342
  incompatible objects, 322–323
  managing interactivity with, 323–326
  managing with KVO, 326–328
  overview, 313
  requirements, 314
  simple, 315–320
  uses for, 314
  using with controllers
    adding controller, 332–333
    data source, 334–335
    reading data from controller, 335–340
    selecting controller keys, 339
  working around keypath limitations, 321
Bindings pane, 557
Bindings tab, Inspector window, 166–167
bit.ly
  API, 251–252
  app delegate header, 248
  creating application, 248–249
  getting key, 250
  getting text from text field, 249–250
  overview, 248
  site, 250
blank application template, 169
block directive, 277
blockOperationWithBlock:, 280, 281
blocks. *See also* threads
  code, 277
  description, 267
  NSBlockOperation, 279–281
  overview, 277
  pointer, 277
  syntax, 277–278
  values, 278–279
  variables, 278–279
blogs, 570
Bonjour, 45
Bounds, 416
boundsRect, 416
boxing, 51
breakpoints. *See also* debugging
  conditional, 477
  controlling execution, 477
  deleting, 478
  setting, 475–476
  using, 475–476
browser, using MVC in, 212
Build and Run button, 88, 141
build errors, 568
built-in applications, 498
bundles
  description, 48
  path, 236–237
Button, ActionScript, 53
buttonWasPressed: method, 413
BViewControllerDidFinish:, 533
BViewController.h, 533
BViewController.m, 533
BWToolkit, 173
byValue: property, 446

C

C array, 285
C main() function, 118
C programming language, 54, 129–130
C# programming language, 50–51
CAAnimation, 446
CABasicAnimation, 446
CAGroupAnimation, 447
CAKeyframeAnimation, 447
CalculationMode property, 449
calibrated colors, 422
CAMediaTiming, 447

CAMediaTimingFunction, 447, 449
canPerformSelector, 221
capitalization error, 567
capitalizedString: method, 376
CAPropertyAnimation, 446
Carbon, 24, 25
Carbonization, 25
caret (^), 277, 278
caseInsensitiveCompare: method, 376
casts, 38, 201
@catch directive, 233–234
category, creating, 149–150
cell objects, selecting, 181–182
celsFloat, 325
center method, 124
CFMutableDictionary, 39
CFString, 27
CGContext, 437
CGGeometry, 131
CGLayer, 436
CGPoint, 403, 428
CGPointMake, 403
CGRect, 403
Character Viewer, 398
characterEncoding method, 378
Cheetah, 7
CIContext, 437
CIFilter, 433
CIImage, 403, 436
CITorusLensDistortion, 455
CIVector, 437, 455
class checking, 201–202
@class directives, 568
class implementation, 110–111
class methods, 97, 104
Class Methods section, 65
class references
  description, 64–65
  exploring, 123
  finding and using, 121–126
classes. *See also* objects
  creating, 107–112
  creating with NSCoder, 302–304
  defined, 97
  expanding search to, 133–134
  getters, 109
  implementation, 110–111
  including in class header, 568
  interface, 107–109
  multiple, 128–129
  naming, 101
  public methods, 111–112
  public properties, 111
  self, 110
  setters, 109
  setting, 170–172
Classes tab, Library window, 161–162
className, 201
closePath: method, 419
Cocoa
  application networks and, 24–25
  application type, 24
  archiving, 44
  bundles, 48
  code layers, 25–26
  creating media projects with, 14
  data objects
    comparing, 40
    copying, 39–40
    description, 38–39, 286–287
    key-value pairs, 41
  designing for, 19–20
  developing iPhone applications with, 12–14
  developing Mac applications with, 9–11
  file management
    archiving, 44–45
    file paths, 45
    URLs, 45
    using NSCoder, 45
  file paths, 45
  frameworks, 29
  history, 3–8
  messaging and notifications, 36
  Model-View-Controller, 46–48
  moving to, 31
  network support, 45
  vs. other platforms
    abstraction, 48–49
    Assembler, 54
    C programming language, 54
    Flash, 52–53
    Java, 53–54
    object orientation, 48–49
    setup and teardown, 48–49
    Windows, 50–52
  overview, 3

Cocoa *(continued)*
plists, 48
profiting from, 9–11
URLs, 45
user experience, 23–24
using Aqua with, 21–22
views, 45–46
windows, 45–46
Cocoa Application template, 86
Cocoa applications
application delegates, 118–121
application frameworks, 24–25
Carbon, 25
C-language features, 129–132
class references, 121–125
code layers, 25
Code Sense, 126–127
creating category on `NSWindow`, 149–150
developing features across layers, 25
elements, 22–23, 117–118
expanding search to related classes, 133–134
frameworks, 27–29
iPad apps, 30
iPhone apps, 30
layers, 22–23, 27–29
multiple classes, 128
receiving message from OS X delegate
delegate methods, 139–142
overview, 134
protocol, 135–136
Protocol Reference page, 137–138
receiving message from OS X with `NSResponder`, 142–144
signatures, 125–126
subclassing `NSWindow`, 144–149
"toll-free bridged" layers, 25
user experience, 23–24
workflow, 29
Cocoa documentation
articles, 61–62
download site, 57
Featured, 61
flowchart, 73
Getting Started, 60, 62–63
guides, 63–64
references, 64–65
release notes, 66
Required Reading, 60
sample code, 66–67
sorting, 71
source code, 72
technical notes, 67–68
technical Q&As, 68–69
Topics breakdown, 69–70
Cocoa Layer, 25, 28, 29
Cocoa Touch, 30
CocoaDev, 571
Cocos2D, 53, 76
code management
copying projects, 487–489
creating snapshots manually, 487–489
using Snapshots, 489–491
using Subversion source control, 491–492
Code Sense, 126–127
code signing, 538
Code tab, 556
colon (:), 126, 567
colors
calibrated, 422
creating, 418
using, 420–421
`colorWithCalibratedHue:`, 422
`commitAnimations:` method, 535
compatibility testing, 82
Compatibility Testing labs, 80
compilation, 76, 88
`componentsSeparatedByString:` method, 376
composite Bezier paths, 419
conditionals, 44
`Conforms to` field, 106
`connection:didFailWithError:` method, 260
`connectionDidFinishLoading:` method, 259, 260
`connection:didReceiveData:` method, 260
`connection:didReceiveResponse:` method, 260
Connections tab, Inspector window, 167, 168
constants, 101
constructors, 102–103
container view, 404–405
Content Filters pane, 430
Content View, 156
content view, 404–405
`contentFilters`, 453, 455
Contents page, 124
Controller Content, 334–335
controllers
adding, 332–333
benefits of, 210
creating, 213

data source, 334–335
glue code, 48
keys, 337–338
reading data from, 335–340
selecting controller keys, 339
using bindings with, 330–340
using MVC with, 212–213
controls, 46
`controlTextDidEndEditing:` method, 282
copyright, 117
`copyWithZone:` method, 40
Core Animation
description, 221
independent timers, 268
layer animation, 27, 452
library, 101
Core Audio, 27
Core Data
approaches to development, 351–352
creating application
adding code, 363–364
adding entity, 354
adding properties, 355–356
building the application, 361–363
creating relationships, 356–359
generating user interface, 359–361
modifying nib, 363–364
project template, 352–353
defined, 351
description, 214
displaying search results, 370–373
features, 351
objects
for editing and search, 367
environment, 366
programming model
creating fetch requests, 367
creating to-many fetch requests, 368–370
using predicates, 367–368
using sort descriptors, 368
Core Foundation, 27, 253
Core Foundation framework, 39
Core Foundation Time Utilities library, 101
Core Graphics, 27, 427–428, 450
Core Location, 496
Core OS Layer, 26
Core Services Layer, 26
CoreAnimation, 268, 431, 439–440
CoreAudio, 101
`CoreImage`
adding effects in Interface Builder, 430–432
applying filters to image, 436–437
creating filter controller interfaces, 435–436
data type, 403
filter keys, 434–435
filters, 429–430
setting up filters for processing, 432–434
`count` method, 299
Counter App Delegate object, 189
`CounterAppDelegate.m`, 188
Counterpart icon, 92
`countText`, 196
Cox, Brad, 4
crashes, 104
Create a New Xcode Project button, 84–85
`.css` file, 551
curly brackets ({}), 111, 141, 277, 567
custom objects, 35
custom switching, 514
Customize Toolbar, 93–94

D

Darwin, 23
Dashboard
application type, 24
description, 543
using modally, 548
Dashboard widgets
adding ad banner space to, 550
adding donation boxes to, 550
adding mobile apps to, 550
advantages, 550
assembling manually, 552–553
building in Dashcode
creating widget graphics, 558–559
using JavaScript, 559–561
components, 551–552
deploying, 562
description, 543
dimensions, 550
disassembling manually, 551–552
displaying, 24
importing, 562
installing, 549
profiting from, 549–550
running as background processes, 548
wrapper, 550

Dashcode
description, 553–554
icons, 556
interface, 554–556
location of, 554
panes, 555
using Inspector, 557
view drop-down menu, 555
window areas, 554
working with Library, 556–557
working with parts, 556–557
data collection, 285
data collection objects. *See also* objects
archiving, 299–300, 304–306
arrays, 295
de-archiving, 299–300, 304–306
defining, 214
enumeration
counting items, 299
mutable collections, 299
performance considerations, 299
using fast enumeration, 298
using implicit enumeration, 298
using `NSEnumerator`, 297
features, 286
key-value observing, 288–290
`NSArray`, 291–292, 293–295
`NSCoder`, 300–306
`NSData`, 300
`NSDictionary`, 296
`NSMutableArray`, 292–295
`NSMutableSet`, 297
`NSNumber`, 288–290
`NSSet`, 297
`NSValue`, 288–290
objects and values, 287
`setValue:forKey:` method, 287–288
using `id` with, 200
data formatter, 363
Data layer, 23
Data Management layer, 23
data model, defining, 214
data objects
comparing, 40
copying, 39–40
description, 38–39, 286–287
key-value pairs, 41
data source, 334–335
Data Sources icon, Dashcode, 556
data storage, 526
data structures, 416
data types, 130
database management, using MVC in, 211
`dataFromRange:` method, 380
`dataOfType:` method, 391
`dealloc` method, 310
`Debug` folder, 88
Debugger window
debugging with machine code, 478
features, 473
object explorer, 478
opening, 479
overview, 478
stack trace, 478
debugging
breakpoints
conditional, 477
controlling execution, 477
deleting, 478
setting, 475–476
using, 475–476
`Debug` folder, 88–89
Debugger window
debugging with machine code, 478
features, 473
object explorer, 478
opening, 479
overview, 478
stack trace, 478
enabling, 474
overview, 465
using instruments, 478–485
using `NSLog`
Console window, 467
format options, 468–469
overview, 466–467
reporting events with, 467–469
reporting line numbers and function names with, 470
reporting values with, 468–469
selectively enabling, 471–472
using Shark, 485–486
`decode:` method, 300, 303
`decodeWithCoder:` method, 300

`Default.png` file, 551
delegates
  converting notifications to, 230–231
  description, 118–121
  errors, 568
  messages, 138–139
  methods, 139–142
  notifications and, 230–231
  receiving message from OS X with, 134–137
delegation
  description, 113
  question-response messaging and, 34
`description` method, 469
descriptive names, 100
descriptors, 368
design elements, 19
design errors, 568
design patterns, 31
developer, registering as, 77–80
Developer Certificate, 538
`Developer` directory, 82, 84
developer forums, 570–571
developer name, 117
Developer Program, 537
device list, 538
dictionary, writing to disk, 44
`didAnimateFirstHalfOfOrientation ToInterfaceOrientation:` method, 511
`didEndEditing:` method, 249
DigiDNA, 12–13
direct property access, 41
direct property animation. *See also* animations
  creating property animation code, 442
  creating timer for, 441
  description, 440
`dismissModalViewController:` method, 508
`dismissModalViewControllerAnimated:` method, 522
Display PostScript, 6
Distribution Certificate, 538, 541
`docFormatFromRange:` method, 380
Dock, 20
Document Object Model (DOM), 559–560
document types, setting, 388–391
Document window, Interface Builder
  adding objects to nib, 176
  creating links, 189
  description, 156–159
  opening, 173
documentation
  articles, 61–62
  download site, 57
  Featured, 61
  flowchart, 73
  Getting Started, 60, 62–63
  guides, 63–64
  references, 64–65
  release notes, 66
  Required Reading, 60
  sample code, 66–67
  sorting, 71
  source code, 72
  technical notes, 67–68
  technical Q&As, 68–69
  Topics breakdown, 69–70
documents
  default nib file, 387–388
  implementing save and open code, 391–393
  printing, 393–394
  setting types, 388–391
  structure, 386
  template, 385
donation boxes, 550
`doSomething` method, 36
downButton, 219
Download Sample Code link, 67
`dragImage:` method, 36
`drawAtPoint:` method, 380
`drawImage:` method, 437
`drawInRect:` method, 380, 426
`drawRect:` method, 105, 428, 436, 443
Driver layer, 23, 26
drop shadow, 36
dual-monitor system, 77
DUNS number, 82
dynamic loading, 100, 112

E

`EAGLview.m`, 519
Edit window, Interface Builder
  adding objects to view in, 176
  description, 159–160
  opening, 173
Edit Window, XCode, 92
editable files, selecting, 92
Effects tab, Inspector window, 164–165

elements
  spacing of, within windows, 21
  standard, 20
`encode:` method, 300, 303
`encodeObject:` method, 306
`encodeWithCoder:` method, 300
encoding
  creating buffers for UTF-8 text, 378
  default, 377–378
  description, 300–301
  UTF-8, 377, 400
`encoding` parameter, 238
Enterprise iPhone program, 82
entities
  adding, 354
  defining, 214
Entity pane, 353
`enumerateObjectsUsingBlock:` method, 370
enumeration
  counting items, 299
  fast, 298
  implicit, 298
  mutable collections, 299
  performance considerations, 299
  using `NSEnumerator`, 297
errors
  build, 568
  checklist, 566
  converting into `NSAlert`, 232–233
  design, 568
  handling, 232–233
  `NSError`, 232
  `NSException`, 233–234
  syntax, 567
`ES1Render`, 519
`ES2Render`, 519
`ES2Renderer`, 519
escaped characters, 253
`estimatedProgress` method, 264
`Event`, ActionScript, 53
events
  handling, 33, 526, 532–534
  reporting with `NSLog`, 467–469
  warning before, 33
  in widgets, 560
exceptions, handling, 233–234
EXIF (Exchangeable Image File Format), 351
explicit instances, 100
explicit object creation, 112
Exposé and Spaces System Preference panel, 548
extra capitalization, 567
extra colon, 567
extra curly bracket, 567
extra parameter, 567
extra square bracket, 567

F

Facebook, 11, 247, 550
`fahrFloat`, 325
fast enumeration, 298
features, naming, 101
fetch requests, 367
File Activity instrument, 483
file attributes, 296
File Controller icon, Dashcode, 556
file handles, 238–239
File List, 92
file management
  Cocoa archiving, 44–45
  description, 44
  file paths, 45
  URLs, 45
  using `NSCoder`, 45
File Manager, 239
file paths
  autocompletion feature, 237–238
  bundle, 236–237
  creating with `NSString`, 236
  standard directories, 237
  URLs and, 45
  using, 238
File Selector sheet, 85
FileManager, 239
`fileReferenceURL:` method, 240
Files icon, Dashcode, 556
File's Owner, 168, 170
`fileURLWithPaths`, 385
Fill & Stroke pane, 557
`fill` method, 420
`filterAngle`, 437
`filterCenter`, 437
`FilterName` property, 372
filters
  animating, 456–458
  applying to image, 436–437
  controller interfaces, 435–436
  keys, 434–435
  setting up, 432–433

filterSize, 437
filterTypes, 238
filterWithName: method, 433
@finally directive, 233–234
Find Text in Documentation, 123, 129
Finder, renaming or moving project folder with, 92
Finder application, 10
finishEncoding: method, 306
Firefox, 486
First Responder
    description, 168
    error, 568
    linking button event to, 530
    methods, 204
    viewing action methods, 203
FirstAppDelegate.m, 119
FirstAppViewController.h, 532
FirstAppViewController.m, 529, 533
First-info.plist, 152
FirstNewWindow class, 172
Flash, 48, 52–53
Flash for iPhone, 76
flipButton action, 219–220
flipsideViewControllerDidFinish: method, 522
floating windows, 128
floatValue method, 325
flying button, 441
font manager, 381–382, 389
FontManager, 169
fonts
    setting, 178–179
    size, 178–179
    standardization of, 21
forKey statement, 41
format translation, 210
Foundation constants, 417
Foundation Framework, 39
Foundation Framework Reference, 101
Foundation Functions, 130–132, 417
frame, 128
Frame, 416
frame method, 133
frameworks, 24–25, 130–132, 569
fromValue: property, 446
frontimg, 558
function names, 470
FunctionBaseIn, 448
FunctionBaseOut, 448
FunctionLinear, 448

G

game object, 96
game playing field, 96
GameKit framework, 504
games
    on iOS, 504
    using MVC in, 211
garbage collection, 43–44, 307–308
Gaussian Blur filter, 431–432
GCC (GNU Compiler Collection), 76
GCD (Grand Central Dispatch), 281
gdb (GNU Project Debugger), 467
generic object, 96
gestural control, 496
get method, 98
GetInfo, 272
getLineStart: method, 376
getParagraphStart: method, 376
getters, 109
Getting Started, 60, 62–63
Global Positioning System (GPS), 496
glue code, 48
GNU Compiler Collection (GCC), 76
GNU Project Debugger (gdb), 467
GNUstep, 76
Google, 11, 247
Google groups, 571–572
GPS (Global Positioning System), 496
gradients, creating, 422–423
Grand Central Dispatch (GCD), 281
graphic identifiers, 512
graphics
    combining Cocoa and Core Graphics, 427–428
    combining Cocoa and Quartz graphics, 427–428
    combining Core Graphics with, 427–428
    combining Quartz graphics with, 427–428
    CoreImage
        adding effects in Interface Builder, 430–432
        applying filters to image, 436–437
        creating filter controller interfaces, 435–436
        data type, 403
        filter keys, 434–435
        filters, 429–430
        setting up filters for processing, 432–434
    creating and drawing gradients, 422–423
    creating Bezier paths with control points, 419
    creating path objects, 418–419
    creating shapes and colors, 418–428

graphics *(continued)*
drawing images, 426
Foundation constants, 417
Foundation functions, 417
loading images, 424–426
MultiBezier project, 428–429
overview, 415
stroking and filling paths, 420
transforming paths, 424
using calibrated colors, 422
using colors, 420–421
using geometric data structures, 416–417
Graphics and Media Layer, 23, 25
groups, 90
Groups & Files pane
groups in, 90
links to files on disk, 91
overview, 90
Growl, 11
GUI window object, 96
guides, 63–64

H

`.h` extension, 107
handlers, 560
help. *See also* project management
blogs, 570
Google groups, 571–572
mailing lists, 571
official developer forums, 570
overview, 569
resources, 570
unofficial developer forums, 571
Usenet groups, 571–572
help files, 21
`hide` function, 561
History list, 92–93
HTML document, 379
`.html` file, 551

I

iAd, 500
IB. *See* Interface Builder
`IBAction` directive, 185
`IBOutlet`, 119, 121, 184–185, 315
Icon Factory, 12
`Icon.png` file, 551–552
`id`
data collections, 200
sender, 200
using to identify objects, 202
using with casts, 201
using with class checking, 201–202
using, 200
Identity tab, Inspector window, 167, 169
image pickers, 511
images. *See also* graphics
compositing, 427
drawing, 426
loading, 424–426, 427
`imageWithBitmap:` method, 436
`imageWithData:` method, 436
implementation
defining, 110–111
defining actions in, 188
defining outlets in, 188
description, 107
`@implementationFirstAppDelegate`, 120
implicit enumeration, 298
implicit instances, 100
implicit object creation, 112
implicit question-response, 35
`#import` directive, 110–111, 568
`#import "FirstAppDelegate.h"`, 120
`#import<Cocoa/Cocoa.h>`, 119
impossible problems, solving, 573
`IndexToNameTransformer`, 347
indirect property access, 41–42
`-info.plist`, 116
`info.plist` file, 153, 390, 551–552
`InfoPlist.strings`, 116
inheritance, 105–107
`init` method, 103–104, 272, 309
`initialize` method, 341
`initWith:` method, 272
`initWith<type>` method, 380
`initWithCalibrated:` method, 50
`initWithCoder:` method, 300, 519
`initWithContentsOfFile:` method, 235, 238
`initWithContentsOfURL:` method, 235, 425
`initWithData:` method, 37
`initWithFrame:` method, 406–407
`initWithString:` method, 240
`inputCenter` key, 454, 455
`inputRadius`, 455

inRange: parameter, 294
Inspector pane, 353
Inspector window, Interface Builder
  Attributes tab, 164
  Bindings tab, 166–167
  Connections tab, 167, 168, 189
  description, 156
  Effects tab, 164–165
  functions, 163
  Identity tab, 167, 169
  opening, 173
  Size tab, 166
  using in Dashcode, 557
instance methods, 65, 104
instance variables, 108
instances, 97, 99
  creating, 100
  explicit, 100
  implicit, 100
  naming, 101
instruments, 478–485
int, 101
interface, 5, 98
Interface Builder (IB)
  adding CoreImage effects in, 430–432
  creating links in
    to actions, 191–192
    to outlets, 190–191
    using selector, 192–194
    windows, 189
  defining outlets and actions
    in implementation, 188
    in interface, 184–187
  designing project in
    adding objects to nib, 176–178
    aligning objects, 179–181
    centering and aligning objects, 182–183
    selecting cell objects, 181–182
    setting fonts and font sizes, 178–179
    windows, 173
    workflow, 174–175
  File's Owner, 168–170
  First Responder, 168–170
  launching, 154
  nib files
    adding objects to, 176–183
    editing, 154
    locating objects from, 153–154
  overview, 16–17, 18
  previewing orientation in, 510
  project management, 569
  setting classes in, 170–172
  using IBOutlet and IBAction, 184–185
  using NSTimer
    start and stop button methods, 197–198
    timer method, 195–196
  UT techniques
    id sender, 200–202
    loose typing, 200–202
    placing outlets and actions, 202–205
  views, 46
  windows
    Document, 156–159
    Edit, 159–160
    Inspector, 156, 163–168
    Library, 156, 160–163
    Main, 156
    overview, 155–156
  workflow, 174–175
@interface directive, 107–109
interface file, 107–109
@interfaceFirstAppleDelegate, 119
international taxation, 542
internationalization, support for, 20
ints, 98
invalidate method, 195
iOS
  adding navigation and control features, 511–514
  app design goals, 503–504
  app switching, 498
  building simple application
    adding view controller subclasses, 526
    creating animated view swap, 534–535
    creating new project in Xcode, 525
    creating views, 528–532
    handling events with protocol messaging, 532–534
    implementing view controllers, 526–528
  built-in applications, 498
  coding differences, 505
  developing for
    Xcode Simulator, 516–517
    Xcode templates, 517–524
  frameworks, 30
  games on, 504
  gestural control, 496
  handling touch events, 514–515

iOS *(continued)*
hardware compatibility, 505
location sensing, 496
managing rotation, 509–511
mobile camera input, 498
moving from OS X, 502–505
multitasking, 498
multi-touch screen control, 496
nib file, 507–508
OpenGL on, 504
orientation sensing, 496
project templates, 86
tilt and motion sensing, 497
view controllers, 507–508
views, 507–508
windows and views on iPad, 515
windows and views on iPhone, 505–508
iOS 3.2, 505
iOS 4, 505
iOS apps, vs. OS X apps, 495–496
iPad
app design goals, 503–504
app development, 501–502
app market, 501
backward compatibility, 494
Cocoa on, 30
description, 494
navigation controllers, 513
popovers, 515
simulating, 516–517
split views, 513, 515
total sales, 501
views, 515
windows, 515
iPad Simulator, 525
iPhone
camera, 498
description, 494
features, 494
home screen, 493
iOS 4 compatibility, 505
models, 494
navigation controllers, 513
orientation sensing, 509–511
simulating, 516–517
total sales, 499
using standard event messages on, 204–205
views, 505–508
windows, 505–508
iPhone apps
adding navigation and control features to, 511–514
building simple application
adding view controller subclasses, 526
creating animated view swap, 534–535
creating new project in Xcode, 525
creating views, 528–532
handling events with protocol messaging, 532–534
implementing view controllers, 526–528
business model, 498–502
custom switching, 514
design goals
constrained data sharing, 504
full-screen operation, 503
instant on, instant off, 503
loss of focus without notice, 503
no explicit data save and restore, 504
orientation sensing, 509–511
restricted battery life, 504
handling touch events, 514–515
vs. Mac apps, 13
market, 499
master/root view controller, 507
vs. OS X apps, 30, 495–496
packaging for App Store, 541–542
prices, 499
profiting from, 12–14
selling in App Store
certification process, 538–540
income territories, 499
online assistant, 539–540
packaging apps for, 541–542
permissions, 538–540
ratings grid, 543
reasons for refusing apps, 536–537
selling in, 82, 536–537
uploading apps to, 542–543
uploading to App Store, 542–543
X-code only policy for, 76
iPhone Developer Program, 81, 503, 538
iPhone OS. *See* iOS
iPhone OS 4, 75
iPhone OS Reference Library, 58
iPhone Program, 82
iPhone SDK, 502
iPhone Simulator, 525
iPhone templates, 85

iPod Touch
description, 494
models, 494
orientation sensing, 509–511
total sales, 499
`isEqual:` method, 40
`isEqualToArray:` method, 40
`isEqualToString:` method, 40, 376
`isfileReferenceURL` method, 240
`isfileURL` method, 240
`isObserving:` method, 327–328
iTunes, 25
iTunes Connect, 543
ivars, 108

J

jailbreaking, 537
Java, 53–54
JavaScript
DOM object access, 559–560
loading images, 561
working with events and handlers, 560
working with time, 561
working with widget events, 561
JavaScript Object Notation (JSON), 255, 257, 556
jEdit code editor, 54, 76
Jobs, Steve, 5, 7
JSON (JavaScript Object Notation), 255, 257, 556

K

Kay, Allan, 4
`kCAMediaTiming`, 448
`kCAMediaTimingFunction`, 448
Kernel layer, 23, 26
keyboard events, 49
keypath limitations, 321
`keyPath` property, 446
Key-Value Coding. *See* KVC
Key-Value Observing. *See* KVO
key-value pairs, 41–42
KVC (Key-Value Coding)
avoiding sources, 224
data access, 214
description, 41
issues, 222
objectifying values, 222–223
using `nil`, 224
using `NSNull`, 224
using `NSNumber`, 223
KVO (Key-Value Observing)
implementing, 288–290
limitations, 228
making assignments compliant, 226–227
managing bindings with, 326–328
monitoring changes to preferences with, 341
`nil/null` values, 228
overview, 224–225
supported constants, 289
using, 227–228

L

layers
animating with, 452–453
Application Framework, 22
blending, 426
code-level overview, 26
developing features across, 26
toll-free bridged, 27
User Experience, 22
`length:` method, 376
Library window, Interface Builder, 156, 160–163, 173, 556
line numbers, 470
`lineToPoint:` method, 419
linking, 65
links
creating, 189–195
to actions, 191–192
to outlets, 190–191
using selector, 192–194
description, 184
errors, 568
localization
defined, 395–398
nib file, 395–396
strings, 396–399
location sensing, 496
looping, 35
loops, creating objects inside, 43
Love, Tom, 4
`lowercaseString:` method, 376

M

.m extension, 107
Mac Dev Center, 77–78
Mac Developer Program, 80–81
Mac legacy, 379
Mac OS X Reference Library, 58
Macs, market share, 9
mailing lists, 571
main method, 272
mainFrame property, 264
main.m, 116, 117
MainMenu.xib file, 120, 154–155
MainWindow.xib file, 521
makeObjectsPerformSelector: method, 298, 414
managed object context, 365
managedObjectContext, 363
manual pagination, creating, 394
maximize method, 149–150
maxLine Beizer paths, 428
Media Layer, 23, 27
Media tab, Library window, 161–163
Medialets, 500
memory leaks, 104
memory management
  code, 309
  creating standard init method, 309
  creating standard setter method, 310
  garbage collection, 307–308
  heuristics, 311
  implementing, 308–311
  in iOS coding, 505
  methods, 308
  in Objective-C, 43–44
  using dealloc, 310
  with view controllers, 526
messages
  forwarding, 33
  loose links between object and, 113
  nested, 37
  receiving message from OS X with NSResponder, 142–144
  selectors, 113
messaging
  Cocoa messaging and notifications, 36
  in Objective-C, 32–33
  question-response, 34
method code, 100
method interface, 100
methods
  adding, 569
  description, 97
  implementing, 139–141
  naming, 101
  objects and, 96
Metrics pane, 557
migration, 363
missing capitalization, 567
missing colon, 567
missing curly bracket, 567
missing parameter, 567
missing semicolon, 567
missing square bracket, 567
mobile apps
  adding navigation and control features to, 511–514
  building simple application
    adding view controller subclasses, 526
    creating animated view swap, 534–535
    creating new project in Xcode, 525
    creating views, 528–532
    handling events with protocol messaging, 532–534
    implementing view controllers, 526–528
  business model
    making money from iPad development, 501–502
    making money from iPhone and iPod touch development, 499
  custom switching, 514
  design goals
    constrained data sharing, 504
    full-screen operation, 503
    instant on, instant off, 503
    loss of focus without notice, 503
    no explicit data save and restore, 504
    orientation sensing, 509–511
    restricted battery life, 504
  handling touch events, 514–515
  vs. Mac apps, 13
  market, 499
  master/root view controller, 507
  vs. OS X apps, 30, 495–496
  packaging for App Store, 541–542
  prices, 499
  profiting from, 12–14
  selling in App Store
    certification process, 538–540
    income territories, 499
    online assistant, 539–540

packaging apps for, 541–542
permissions, 538–540
ratings grid, 543
reasons for refusing apps, 536–537
selling in, 82, 536–537
uploading apps to, 542–543
uploading to App Store, 542–543
X-code only policy for, 76
mobile camera input, 498
modal view swaps, 508
modal views, 508
`modalTransitionsStyle` property, 508
Model Key Path, 349
Model-View-Controller. *See* MVC
Mono, 51–52, 76
MonoDevelop environment, 51
motion sensing, 497
mouse events, 414–415
`mouse: inRect:` method, 414
`mouseDown:` method, 148, 414, 444
`mouseDragged:` method, 408, 414–415
`mouseEntered:` method, 415
`mouseExited:` method, 415
`mouseMoved:` method, 415
`mouseUp:` method, 415
`moveToPoint:` method, 419
`MovieClip`, ActionScript, 53
MS Word importer, 380
`MSMakeRange()`, 377
`MSMutableSet`, 286
multi-alphabet support, 20
MultiBezier, 428–429
MultiButtons project, 411
Multicore instrument, 483
MultiDocument
default nib file, 387–388
document structure, 386
implementing save and open code, 391–393
printing documents, 393–394
setting document types, 388–391
template, 385
multi-language support, 20
multiple classes, 128–129
multiple users, 20
multiple windows, 20–21
multitasking, 498
multi-touch screen control, 496
mutable collections, 299
MVC (Model-View-Controller)
controller layer, 210
creating custom controllers, 213
description, 46–48
outline form, 210
overview, 209
using
in browser, 212
with Cocoa controller objects, 212–213
in database management, 211
in games, 211
`MyDocument` class, 386
`MyLog`, 472
`MyNewClass` class, 108
`myObject`, 37

N

names
changing, 86
convention, 101–102, 145
descriptive, 100
saving, 85
Nanopad text editor
creating, 381–384
implementing Open Recent menu, 384
saving and loading rich text, 383–384
template, 381
using `NSFontManager`, 381–382
navigation controllers, 508, 513–514, 518
Navigation-based Application template, 518
nesting, 103–104
.Net, 50
network support, 45
`new_hash` field, 256
`newRect`, 427
`nextObject` method, 297
NeXTStep, 5–7, 29
nib files
adding objects to, 176–178
creating, 387–388
default, 387–388
defined, 46, 151
description, 116
editing, 154
iOS, 507–508
key features, 152
loading, 151–153

nib files *(continued)*
loading objects from, 100, 153–154
loading options, 153
localized, 395–396
for Picopad, 244
for simple animation, 444
`nil`, 224, 229, 311
`nodeForXPath`, 261
non-modal view swaps, 508
notifications
`aNotification`, 139
Cocoa messaging and, 36
delegates and, 230–231
flowchart, 229
overview, 228
posting, 230
registering object for, 229
triggering events with, 229
`NSAffineTransform`, 418
`NSAlert`, 232–233
`NSAnimationContext`, 446
`NSApplication`, 104, 169
`NSApplicationDelegate`, 119, 135, 137
`NSApplicationDirectory`, 237
`NSApplicationName` key, 339
`NSApplicationPath` key, 340
`NSApplicationProcessIdentifier` key, 340
`NSArray`
data sources, 291
description, 38, 286, 298
features, 285
id data type, 200
searching, 293–294
`setValue:forKey`, 289
sorting, 294–295
using, 291–292
`NSArrayController`
adding, 333
Controller Content property, 334
description, 330
predefined keys, 345
using MVC with, 212
`NSASCIIStringEncoding`, 238
`NSAttributedString`, 378–379
`NSBezierPath`, 403, 418
`NSBlockOperation`
passing parameters to, 280–281
using, 279–280
`NSBundle`, 153
`NSButton`, 53, 106–107
`NSButtonCell` objects, 185–186
`NSCell`, 162
`NSClass`, 37
`NSCoder`
conversion methods, 300
creating class with, 302–304
description, 45
implementation methods, 300
saving and loading composite objects to disk, 301
uses for, 235, 242
`NSColor`, 49–50, 420–421
`NSColorSpace`, 418
`NSContext`, 437
`NSControl`, 406
`NSCopying` method, 40
`NSData`, 259, 300, 380
`NSDate`, 97, 104, 268
`NSDesktopDirectory`, 237
`NSDictionary`
accessing file attributes, 296
description, 38, 298
key-value pairs, 39
using, 296
using `ID` with data collections, 200
`NSDictionaryController`, 212, 330
`NSDocument`, 153, 385, 394
`NSDownloadsDirectory`, 237
`NSEntityDescription`, 367
`NSEnumerator`, 297
`NSEqualPoints`, 417
`NSEqualRect`, 417
`NSError`, 232
`NSEvent`, 408
`NSException`, 233–234
`NSFetchRequest`, 367
`NSFileHandle`, 239, 281
`NSFontManager`, 381–382
`NSForm`, 302, 305
`NSGradient`, 418, 422–423
`NSGraphicsContext`, 418
`NSHomeDirectory()` function, 236
`NSImage`, 403, 418, 424–426
`NSIndexSet`, 287, 338
`NSInteger`, 101, 294–295
`NSIntegralRect`, 417
`NSIntersectionRect`, 417

NSInvocation, 221
NSISOLatin1StringEncoding, 377
NSKeyedArchiveFromDatraTransformerName, 349
NSKeyedArchiver, 300, 306
NSKeyValueChangeInsertion constant, 289
NSKeyValueChangeRemoval constant, 289
NSKeyValueChangeReplacement constant, 289
NSLog. *See also* debugging
  Console window, 467
  format options, 468–469
  implementing delegation method, 140
  listing running apps, 331
  overview, 466–467
  reporting events with, 467–469
  reporting line numbers and function names with, 470
  reporting values with, 468–469
  selectively enabling, 471–472
NSMakePoint, 403, 417
NSMakeReact, 130, 417
NSManagedObject, 367
NSManagedObjectContext, 366
NSManagedObjectModel, 366
NSMidX, 417
NSMidY, 417
NSMutableArray
  description, 286
  searching, 293–294
  sorting, 294–295
  using, 292–293
NSMutableAttributedString, 379
NSMutableData, 300, 306
NSMutableDictionary
  description, 286, 298
  key-value pairs, 39
  setValue:forKey: method, 288
NSMutableSet, 297
NSMutableString, 39
NSNib, 153
NSNotFound, 294
NSNull, 224
NSNumber, 223, 270, 287, 290–291
NSNumberFormatter, 328–330
NSObject
  as blank generic subclass, 146–147
  description, 105
  key-value observing, 225
  performSelector: in, 268
  running on delegates, 120
  subclass hierarchy, 106
  subclassing, 108, 119, 304, 386, 406
  uses for, 97
NSObjectController, 330
NSOpenGLContext, 458
NSOpenGLView, 458
NSOpenPane, 240, 241, 245
NSOperation
  advantages, 271–272
  creating new object, 272–273
  disadvantage, 277
  Grand Central Dispatch, 281
  running object, 274
  using, 275–276
NSOperationQueue, 274–277, 280, 281
NSOrderedAscending, 294–295
NSOrderedDescending, 294–295
NSOrderedSame, 294–295
NSPanel, 345, 405
NSPersistentStoreCoordinator, 366
NSPoint, 290, 403, 416, 428
NSPointFromCGPoint, 428
NSPointInRect, 417
NSPointToCGPoint, 428
NSPredicate, 367
NSRange, 291, 376–377
NSRect, 129–130, 291, 403, 416
NSRectFromCGrect, 403
NSRectToCGrect, 403
NSResponder
  description, 406
  handling keyboard events, 49
  handling mouse events, 414–415
  placing events in OS X, 204
  receiving message from OS X, 142–144
NSSavePane, 240, 242, 243, 245
NSScreen, 133
NSScrollView, 243, 244
NSSet
  adding to nib, 162
  description, 38, 286, 298
  id data type, 200
  methods, 297
  touch events, 515
NSSize, 291, 416
NSSlider, 317

NSSortDescriptor, 367
NSString
  creating paths with, 236
  description, 39
  methods, 235, 376
  overview, 375
NSStringstringWithFormat: method, 196
NSTask, 281–283
NSTextField, 185, 196
NSTextFieldDelegate protocol, 249
NSTextView, 243, 244, 302, 381
NSThread
  description, 270
  handling UO and thread interactions, 271
  managing thread memory, 271
  pausing thread, 270
NSTimeInterval, 267
NSTimer, 195–199, 221, 267–268, 441
NSTrackingArea, 415
NSTreeController, 330
NSUInteger, 338
NSUnarchiveFromDataTransformerName, 349
NSUndoManager, 394–395
NSUnionRect, 417
NSURL, 240
NSURLConnection object, 260
NSURLRequest object, 260
NSUserDefaults, 341
NSUserDefaultsController, 330, 340–341, 345–346
NSUTF8StringEncoding, 238, 377–378
NSValue, 39, 287, 290–291
NSValueTransformer, 347
NSView
  description, 46, 405
  drawInRect: method, 105–106
  handling mouse events, 414–415
  subclassing, 108
  template, 406
NSViewController, 153, 406
NSWindow
  application, 405
  class references, 121–122
  creating category on, 149–150
  in document-based application, 385
  extending with maximize method, 149–150
  guide, 129
  setting classes and subclasses, 170–172
  subclassing, 144–149
NSWindow window object, 36
NSWindow*window;, 119
NSWindowController, 153
NSWorkspace, 331
NSXMLDocument, 261, 378
NSXMLElement, 260
NSXMLNode, 260
NSXMLParse, 261
NSZeroPoint, 416
NSZeroRect, 416, 426
NSZeroSize, 416
null, 229
numTaps, 515

O

Object, ActionScript, 53
Object Allocations, 484–485
object explorer, 478
object names, finding, 186
object property, 32
<object>WithContentsOfFile: method, 238
objectAtIndex: method, 294
objectForKey, 287–288
Objective-C
  asterisks in, 37–38
  blocks
    code, 277
    description, 267
    NSBlockOperation, 279–281
    overview, 277
    pointer, 277
    syntax, 277–278
    values, 278–279
    variables, 278–279
  classes
    description, 97–98
    getters, 109
    implementation, 110–111
    interface, 107–109
    public methods, 111–112
    public properties, 111
    self, 110
    setters, 109
  Cocoa messaging and notifications, 36
  features, 112
  history, 4–5
  implicit question-response, 35

instances, 98–100
memory management, 43–44
messaging, 32–33
moving to, 31
objectification with @ character, 38
objects
constructors, 102–103
designing, 100–107
inheritance, 105–106
naming, 101–102
overview, 31–32
pointers, 102–103
subclassing, 105–106
question-response messaging, 34
subclassing in, 32
syntax, 36
using objects in, 113–114
object-oriented architecture, 112
object-oriented programming, 95–96
objects
abstraction, 95
adding to nib, 176–178
aligning, 179–181, 182–183
applications, 96
archiving, 304–306
cell, selecting, 181–182
centering, 182–183
changing class of, 170–172
conditionals and, 44
constructors, 102–103
creating, 33
data dependency management, 95
de-archiving, 304–306
designing, 100–107
encapsulation, 95
event management, 95
finding names of, 186
flow control, 95
getting property, 32
initializing, 33
looping, 95
loose links between messages and, 113
naming, 101–102
Objective-C, 31–32
path, 418–419
pointers, 102–103
posting, 230
posting status of, 33
private features, 98
receiver, 99
registering for notifications, 229
releasing, 33
setting property, 32
using, 230–231
using `id` sender to identify, 202
using in Objective-C, 113–114
vs. values, 287
Objects tab, Library window, 160–161
`ObjectToWatch`, 225
observedProperty key, 225
`observeValue` method, 226
`observeValueForKeyPath:` method, 225, 328
`onclick` message, 34
online assistant, 539–540
`onload` message, 34
opacity, 426
opaque objects, 5
Open Office documents, 379
Open Recent menu, 384
`openADocument:` method, 244
OpenDocumentText, 379
OpenGL. *See also* animations
controlling animation, 462–463
creating animation, 459–462
description, 439, 440
functions, 459
overview, 457–458
Utility Toolkit, 459
OpenGL ES, 504
OpenGL ES Application template, 519
OpenStep, 6
`orderFrontPanel:` method, 381
orientation sensing, 496, 509–511
OS Layer, 28
OS X 10.0, 7
OS X application
creating, 29
vs. iOS apps, 495–496
OS X documentation
articles, 61–62
download site, 57
Featured, 61
flowchart, 73
Getting Started, 60, 62–63
guides, 63–64
references, 64–65

OS X documentation *(continued)*
release notes, 66
Required Reading, 60
sample code, 66–67
sorting, 71
source code, 72
technical notes, 67–68
technical Q&As, 68–69
Tools & Languages section, 70
Topics breakdown, 69–70
OS X layers, 28
OS X project
creating, 84–89
naming, 85
renaming, 86
saving, 87
selecting items for editing, 92
windows
resizing, 87
switching between, 88
tiling, 88
Other References group, 101
outlets, 184
creating links to, 190–191
defining in implementation, 188
defining in interface, 185–187
placing, 202–203
`outputArray`, 237

P

Palo Alto Research Center, 4
parameter errors, 567
parameters lists, defining, 112
parts, 556
path animation, 450–452
path objects. *See also* graphics
Bezier paths with control points, 419
composite Bezier paths, 419
creating, 418–419
filling, 420
stroking, 420
transforming, 424
paths
autocompletion feature, 237–238
bundle, 236–237
creating with `NSString`, 236
standard directories, 237
URLs and, 45
using, 238
PCs (personal computers), 9
`performSelector:`
description, 215, 221
implement pause method, 269
messaging across threads, 269
running selector in separate thread, 269
using, 268
`performSelectorInBackground:` method, 269
`performSelectorOnMainThread:` method, 269, 271, 275
`performSelector:withObject:afterDelay:` method, 373
personal computers (PCs), 9
Picopad
nib files for, 244
saving files for, 247
`ping`, 281
Placement buttons, 183
platform comparison
abstraction, 48–49
Assembler, 54–55
C programming language, 54
Flash, 52–53
Java, 53–54
object orientation, 48–49
setup and teardown, 48–49
Windows, 50–52
plists, 48
`.png` files, 551–552
pointers, 102–103
popovers, 515
POSIX, 24
`postFilterImage`, 437
pre-defined keys, 42
predicates, 367–368
`predicateWithFormat:`, 367–368
preferences. *See also* bindings
creating application with, 343–346
implementing with bindings, 340–346
initializing, 341–342
reading values, 342–343
setting keys, 342
Preferences Controller, 345
`preFilterImage`, 437
`Prefix.pch` file, 117, 118
`presentModalViewController:` method, 508, 522
preset keyboard shortcuts, 20
`print:` method, 384
`printDocument:` method, 393–394

printf, 140
printInfo, 393–394
problem solving, 566–568, 573
project management
  adding frameworks, 569
  adding new properties or methods, 569
  creating new subclass, 568
  error checklist, 566
  help resources, 569–572
  including other classes in class header, 568
  reference documentation, 565
  solving problems, 566–568, 573
  source code management in, 565
  techniques, 565–566
  version control in, 565
  working in Interface Builder, 569
project windows
  closing, 89
  hiding, 89
  position, 89
  resizing, 87
  restoring, 89
  size, 89
  switching between, 88
  tiling, 88
projects
  copying, 487–489
  creating snapshots manually, 487–489
  designing in Interface Builder
    adding objects to nib, 176–178
    aligning objects, 179–181
    centering and aligning objects, 182–183
    selecting cell objects, 181–182
    setting fonts and font sizes, 178–179
    windows, 173
    workflow, 174–175
  naming, 85
  renaming, 86
  saving, 87
  selecting items for editing, 92
Projects folder
  adding to Developer directory, 84
  creating, 83
properties, 96
Properties section, 65
@property, 119
property
  access, 41–42
  adding, 355–356, 569
  Core Data, 355–356
  defined, 97
  getting, 33
  naming, 101
@property directive, 108
Property pane, 353
protocol messaging, 532–534
Protocol Reference page, 137–138
protocols, 135–136
protocols field, 108
Provisioning Profile, 538–539
pseudo-properties, 42
public method, defining, 111–112
public properties, defining, 111
Pure Data audio synthesizer, 15

Q

quad-monitor system, 77
Quartz 2D, 427–428
Quartz 2D graphics library, 131
Quartz Composer, 53
QuartzCore, 445, 504
question-response messaging, 34
Quick Start page, 122

R

range, 376–377
rangeOfString: method, 258
readFromData: method, 391
receiver, 99
RectsView, 420
reference counting, 43–44
reference documentation, 565
references, 64–65
refreshLink: method, 370
refreshList: method, 369, 372
registerAsObserver: method, 225
registering as developer, 77–80
relationships
  creating, 356–359
  reciprocal, 359
  to-many, 357
RelativeLineToPoint: method, 419
RelativeMoveToPoint: method, 419
Release Notes, 66
reminder field, setting, 126
remove function, 561

removeAllObjects, 296
removedOnCompletion:, 446
removeFromSuperview: method, 413, 508, 529
removeObjectForKey, 296
removeObjectsForKeys, 296
repeating counter, creating, 196
replaceSubview: method, 435
resetCount: method, 191, 198
Resources group, 154–155
responder chain, 33, 531
Return key, 126
return values, defining, 112
returnXML, 255, 256
rich text, 383
rich text editor. *See* Nanopad text editor
Rich Text Format (RTF), 379
root class, 107
root view, 406–409
RootViewController, 518
RTF (Rich Text Format), 379
RTFD files, 379
RTFDFromRange: method, 380
RTFFromRange: method, 380
runningApps array, 334–336
runningApps property, 331

S

Safari
  layout, 21
  overview, 17
Safari Developer Program, 17
Sample Code, 66–67
saveRecord: method, 304
saveTheDocument: method, 244
screen method, 133
searchReturnArray, 370, 371
seconds counter, creating, 196
SEL variable, 216
selectedName text box, 372
selectedObjects, 338
Selection, 338
selectionIndex, 338
selectionIndexes, 338
selector data type, 113
selectors
  applications, 221
  creating links, 192–194
  defining, 215–216
  defining in Interface Builder, 217–218
  limitations, 216–217
  running in separate thread, 269
  using, 216
self, 110
self-messaging, 113
semicolon (;), 111, 567
Services Layer, 28, 29
set method, 98
setArguments: method, 282
setAttributedTitle: method, 380
setDictionary: method, 296
setEnabled: property, 198
setFrame:, 128
setFrameOrigin:, 128, 442
setFrameSize:, 442
setObject: method, 341
setters, 109, 310
setTitle: link, 124
setTitle: method, 125
setValue: forKey: method, 287–288, 289
shared property, 275
SharedInstance property, 275
Shark, 485–486
shouldAutorotateToInterfaceOrientation:
    method, 509
show function, 561
showAController method, 533
showBView: method, 526, 529, 534, 535
showFront(event) routine, 561
showPanels, 394
showPreferences method, 345
showRear(event) routine, 561
signatures, 109, 125–126
sign-up page, 79
Size tab, Inspector window, 166
size_t integer, 104
Sizing Windows task, 128
sleepUntilDate: method, 270
sliderCount property, 315–316, 320
Smalltalk, 4
Smoaktalk, 6
Snapshots, 487, 489–491
Snow Leopard, 8
sort descriptors, 368
source code, 72
source code management, 565
Source Control, 487, 491–492
sourceURL, 249
Split icon, 92

Split View-based template, 520
square brackets ([]), 40, 127, 567
stack trace, 478
standard directories, finding, 237
Start button, 197–198
`startCount:` method, 191
start-up screen, 85
`stdio`, 281
StepStone, 5
Stop button, 197–198
`stopButton`, 198
`stopCount:` method, 198
string encoding, 238
`string` method, 240
`stringFromFloat` method, 325
strings
  attributed
    drawing text on path, 380
    loading, 379–380
    saving, 379–380
    using, 380
  creating buffers for UTF-8 text, 378
  default encoding, 377–378
  localized, 396–399
  `NSAttributedString`, 378–380
  `NSRange`, 376–377
  `NSString`, 375–376
`stringValue` parameter, 196
`stringValue` property, 250
`stringWithContentOfURL:` method, 376
`stringWithContentsOfURL:` method, 258
`stringWithFormat:` method, 376
`stroke` method, 420
`structs`, 96
subclasses
  creating, 568
  description, 107
  setting, 170–172
subclassing
  description, 32
  inheritance and, 105–107
  `NSWindow`, 144–149
  root view, 406–409
  views, 411–414
subpath, 237
`subStringWithRange:` method, 258, 376–377
`subtractOne:` method, 219–220
Subversion source control, 491–492
subview, 411
Sudden Termination, 483
`super` variable, 107
superclasses, 107
`superview`, 411
`suspend` method, 274
swap file, 505
Symbol list, 92–93
`sync` function, 561
`synchronizeFile` method, 239
synchronous requests, 258
syntax errors, 567
`@synthesize` directive
  adding pointers to, 188
  class implementation, 110–111
  creating setter and getter code with, 32, 310
  description, 109
  uses for, 97
`@synthesizewindow;`, 120

T

Tab Bar Application template, 521
tab bar controllers, 508
tab bars, 511, 512
Tab key, 126
table objects, 35
table views, 511
`target` property, 195
target-action
  defining selectors, 215–216, 217–218
  description, 214–215
  example application, 219–221
  limitations of selectors, 216–217
  using selectors in code, 216
`targetObject:`, 268, 394
Tasks section, 64
taxation, 542
Technical Notes, 67–68
Technical Q&As section, 68–69
template window, 85
templates, Xcode
  Navigation-based Application, 518
  OpenGL ES Application, 519
  overview, 517
  Split View-based, 520
  Tab Bar Application template, 521
  Utility Application, 521–522
  View-based Application, 523
  Window-based Application, 524

Tempverter, 324–325
testPath, 237
text. *See also* documents
  attributed strings
    drawing text on path, 380
    loading, 379–380
    saving, 379–380
    using, 380
  creating buffers for UTF-8 text, 378
  default encoding, 377–378
  localized, 396–399
  NSAttributedString, 378–380
  NSRange, 376–377
  NSString, 375–376
text editor. *See* Nanopad text editor
Text pane, 557
text parsing, 257–258
text strings, 38
theEvent, 408
theIndex, 294
theOptionalObject, 270
theRange, 394
theText, 219
TheView, 444
TheWindow, 244–246
this, ActionScript, 53
thisScreen, 134
threads. *See also* blocks
  memory management, 271
  pausing, 270
  user interface and, 271
3D graphics
  creating property animation code, 442
  creating timer for, 441
  overview, 439
  types, 440
  using animator
    creating path animation, 450–452
    creating simple proxy animation, 443–445
    creating values/keytimes animation, 449
    customizing animation object, 446–449
    setting duration, 446
    timing functions, 448–449
  using CALayer
    animatable filter, 453–456
    animating filter, 456–458
    using layers for animation, 452–453
    using drawRect:, 443
    using OpenGL, 458–463
tilt and motion sensing, 497
Time Profiler, 483–484
timer method, implementing, 195–196
timerMethod, 195, 320
timers, 267–268
timing functions, 448–449
timingFunction:, 446
toll-free bridged layer, 27
to-many fetch requests, 368–370
to-many relationship, 357
toolbar
  customizing, 93–94
  graphic identifiers, 512
  UIKit, 511–512
Tools & Languages section, 70
Topics breakdown, 69–70
touch events, 514–515
touch object, 515
touchesBegan: method, 514–515
touchesCancelled: method, 514
touchesEnded: method, 514
touchesMoved: method, 514–515
tracerroute, 281
transforms, 424
transparency, 426
triple-monitor system, 77
Trism game app, 500
@try directive, 233–234
Twitter, 11, 247, 550
Twitterific, 11, 12
2D graphics
  combining Core Graphics with, 427–428
  combining Quartz graphics with, 427–428
  CoreImage
    adding effects in Interface Builder, 430–432
    applying filters to image, 436–437
    creating filter controller interfaces, 435–436
    data type, 403
    filter keys, 434–435
    filters, 429–430
    setting up filters for processing, 432–434
  creating and drawing gradients, 422–423
  creating Bezier paths with control points, 419
  creating path objects, 418–419

creating shapes and colors in drawRect:, 418–428
drawing images, 426
loading images, 424–426
MultiBezier project, 428–429
overview, 415
stroking and filling paths, 420
transforming paths, 424
using calibrated colors, 422
using colors, 420–421
using Foundation constants, 417
using Foundation functions, 417
using geometric data structures, 416–417
<type>Value: method, 376

U

UI. *See* user interface (UI)
UIEvent, 515
UIGestureRecognizer, 496, 514–515
UIKit
framework, 494, 504, 506
lit of user events, 530
tab bars, 511
toolbars, 511
UITableView, 518
UIView, 105, 507–508
UIViewController, 507–508, 523, 526
UIWindow, 505–515, 524, 535
Universal Type Identifier (UTI), 389
UNIX Development box, 82
upButton, 219
updates, 81
uppercaseString: method, 376
<url> tags, 258
URLs
creating, 240
file paths and, 45
long, 252–253
paths, 240
reading and writing data with, 240
references, 240
sending request, 253–255
URLWithString: method, 240
"Use Core Data for storage" check box, 352
Usenet groups, 571–572
user control, 210
user experience, 23–24
User Experience layer, 22
user interface (UI)
Core Data, 359–361
creating, 199
generating, 359–361
id sender, 200
loose typing, 200
using id with data collections, 200
userInfo property, 268
<usr>, 82
UTF-8 encoding, 377, 400
UTI (Universal Type Identifier), 389
Utility Application template, 521–522

V

Value Transformer box, 349
value transformers
creating, 347–348
defined, 347
setting, 349
valueForKey: method, 287–288, 342, 370
valueForUndefinedKey: method, 224
values
getting, 33
key-value pairs, 41, 287
of preferences, 342–343
version control, 565
version.plist file, 551–552
view controllers
adding, 526
description, 507–508
implementing, 526–528
view swaps, 508, 534–535
View-based Application template, 523
viewDidLoad method, 508, 523, 526
viewForUIConfiguration, 435
views
Cocoa, 45–46
container, 404–405
creating, 528–532
handling mouse events in, 414–415
hierarchy
adding from, 409–414
description, 406
removing from, 409–414
tree structure, 410
Interface Builder, 46
iOS, 507–508

views *(continued)*
iPhone, 505–508
layer-backed, 452
objects, 405–406
with random buttons, 413
root, 406–409
subclassing, 411–414
subclassing root view, 406–409
subviews, 410–412
`visibleFrame`, 133–134
Visual Basic, 50
Visual Basic for Applications, 50
`-(void)`, 120

W

`.wdget` file, 551–552, 562
Web APIs
creating asynchronous Web requests, 258–260
creating long URL, 252–253
creating XML requests, 255–256
parsing XML returns as text, 257–258
selecting XML format, 256–257
sending URL request, 253–255
using, 247–248
WebKit, 24, 262
`WebView`, 261–264
`WebViewAppleDelegate.h` file, 262
`WebViewAppleDelegate.m` file, 263
widgets
adding ad banner space to, 550
adding donation boxes to, 550
adding mobile apps to, 550
advantages, 550
assembling manually, 552–553
building in Dashcode
creating widget graphics, 558–559
using JavaScript, 559–561
components, 551–552
deploying, 562
description, 543
dimensions, 550
disassembling manually, 551–552
displaying, 24
importing, 562
installing, 549
profiting from, 549–550
running as background processes, 548
wrapper, 550
`willRotateToInterfaceOrientation:` method, 511
Window (First) object, 170–172
Window Programming Guide for Cocoa, 128
Window-based Application template, 524
`windowControllerDidLoadNib:` method, 391
windows
Cocoa, 45–46
container, 404–405
Interface Builder
Document, 156–159
Edit, 159–160
Inspector, 156, 163–168
Library, 156, 160–163
Main, 156
overview, 155–156
iPhone, 505–508
objects, 405–406
Xcode
Edit, 92
Groups & Files, 90
template, 85
Windows operating system, 51
`withObjCType:` parameter, 291
WordDoc, 379
Workflow icon, Dashcode, 556
Workflow Steps pane, 556
`writeData:` method, 239
`writeToFileURL:` method, 376

X

X11, 24
Xcode
alternatives to, 76
customizing toolbar, 93–94
developing for iOS in
using Xcode Simulator, 516–517
Xcode templates, 517–524
installing, 82–84
iPhone Developer Program, 80–82
Mac Developer Program, 80–82
overview, 16–17, 75
registering as developer, 77–80
start-up screen, 85
windows
Edit, 92
Groups & Files, 90
template, 85
Xcode 3 vs. Xcode 4, 153

Xcode 4, 84, 153
Xcode project
  creating, 84–89
  naming, 85
  renaming, 86
  saving, 87
  selecting items for editing, 92
  windows
    resizing, 87
    switching between, 88
    tiling, 88
Xcode SDK
  download site, 57
  file size, 75
  sample code, 66–67
Xcode Simulator, 516–517
Xcode templates
  Navigation-based Application, 518
  OpenGL ES Application, 519
  overview, 517
  Split View-based, 520
  Tab Bar Application template, 521
  Utility Application, 521–522
  View-based Application, 523
  Window-based Application, 524
`Xcode.mpkg` file, 76, 82
XML
  classes, 260–261
  format, selecting, 256–257
  parsing returns as text, 257–258
  requests, creating, 255–256
XPath, 260

Y

Yahoo, 247
Yellow Box, 51

Z

Zettaboom app, 30
Zip file, 562

Everything You Need to Craft Killer Code for Apple Applications

Whether you are a seasoned developer or just getting into the Apple platform, Wiley's Developer Reference series is perfect for you. Focusing on topics that Apple developers love best, these well-designed books guide you through the most advanced and very latest Apple tools, technologies, and programming techniques. With in-depth coverage and expert guidance from skilled authors who are proven authorities in their fields, the Developer Reference series will quickly become your indispensable Apple development resource.

The Developer Reference series is available wherever books are sold.

Developer Reference

WILEY
Now you know.

Learn Cocoa — for fun and for business

Cocoa is *the* programming environment for Apple development, and this information-packed developer's guide is your key to the Cocoa libraries and the Apple developer tools. Written for developers by an experienced Mac expert and iPhone developer, this book shows you how to learn and use Xcode and Objective-C, design user interfaces, optimize your code, manage data, create animations and special effects, and package apps for the App Store.

- Master and understand the Xcode® SDK, Objective-C®, and the Cocoa API
- Create, use, profile, and debug custom objects and subclasses
- Design user interfaces with Interface Builder
- Streamline your code with design patterns such as Model-View-Controller (MVC)
- Work with text, PDFs, Web data, and Apple's Core Data API
- Create simple and advanced animation effects with Core Animation and Core Image
- Learn to develop apps for Mac OS® X and Apple devices

Access the latest information on Apple development

Visit www.wileydevreference.com for the latest on tools and techniques for Apple development, and download source code for the projects in this book.

Richard Wentk is a developer with more than fifteen years of experience in publishing, and is one of the UK's most reliable technology writers. He covers Apple products and developments for *MacWorld* and *MacFormat* magazines, and also writes about technology and business strategy for publications such as *Computer Arts* and *BBC Focus.*

WILEY
wiley.com

Reader Level: Intermediate to Advanced

Shelving Category: COMPUTERS / Programming / Apple Programming

$49.99 USA • $59.99 CANADA

ISBN 978-0-470-49589-6
54999
9 780470 495896