
Cocoa Fundamentals Guide



2008-07-08



Apple Inc.
© 2008 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

.Mac is a registered service mark of Apple Inc.

Apple, the Apple logo, AppleScript, AppleScript Studio, Aqua, Bonjour, Carbon, Cocoa, iChat, iPod, Mac, Mac OS, Macintosh, Objective-C, OpenDoc, Quartz, QuickTime, Safari, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Finder, iPhone, Multi-Touch, and Spotlight are trademarks of Apple Inc.

NeXT and NeXTSTEP are trademarks of NeXT Software, Inc., registered in the United States and other countries.

Adobe, Acrobat, and PostScript are trademarks or registered trademarks of Adobe Systems Incorporated in the U.S. and/or other countries.

Intel and Intel Core are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

OpenGL is a registered trademark of Silicon Graphics, Inc.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction [Introduction to Cocoa Fundamentals Guide](#) 13

[Organization of This Document](#) 13

[See Also](#) 14

Chapter 1 [What Is Cocoa?](#) 15

[The Cocoa Environment](#) 15

[Introducing Cocoa](#) 15

[How Cocoa Fits into Mac OS X](#) 16

[How Cocoa Fits into iPhone OS](#) 19

[Features of a Cocoa Application](#) 20

[The Development Environment](#) 23

[Platform SDKs](#) 23

[Overview of Development Workflows](#) 24

[Xcode](#) 24

[Interface Builder](#) 27

[The iPhone OS Simulator](#) 29

[Performance Applications and Tools](#) 30

[AppleScript Studio \(Mac OS X\)](#) 32

[The Cocoa Frameworks](#) 33

[Foundation](#) 34

[Application Kit \(Mac OS X\)](#) 40

[UIKit \(iPhone OS\)](#) 47

[Comparing Application Kit and UIKit Classes](#) 51

[Other Frameworks With Cocoa API](#) 54

[A Bit of History](#) 54

Chapter 2 [Cocoa Objects](#) 57

[A Simple Cocoa Command-Line Tool](#) 57

[Object-Oriented Programming With Objective-C](#) 58

[The Objective-C Advantage](#) 59

[The Dynamism of Objective-C](#) 59

[Extensions to the Objective-C Language](#) 60

[Using Objective-C](#) 66

[The Root Class](#) 69

[NSObject](#) 69

Root Class—and Protocol	70
Overview of Root-Class Methods	70
Interface Conventions	72
Instance and Class Methods	72
Object Retention and Disposal	73
How Garbage Collection Works	74
How Memory Management Works	76
Object Creation	80
Allocating an Object	80
Initializing an Object	81
The dealloc and finalize Methods	88
Class Factory Methods	89
Introspection	90
Evaluating Inheritance Relationships	90
Method Implementation and Protocol Conformance	91
Object Comparison	92
Object Mutability	93
Why Mutable and Immutable Object Variants?	93
Programming With Mutable Objects	94
Class Clusters	98
Simple Concept, Complex Interface	98
Simple Concept, Simple Interface	99
Creating Instances	99
Class Clusters With Multiple Public Superclasses	100
Creating Subclasses Within a Class Cluster	101
Creating a Singleton Instance	106

Chapter 3 Adding Behavior to a Cocoa Program 109

Starting Up	109
What Happens in the main Function	109
Using a Cocoa Framework	111
Kinds of Framework Classes	112
Cocoa API Conventions	112
Inheriting From a Cocoa Class	115
When to Override a Method	115
When to Make a Subclass	117
Basic Subclass Design	119
The Form of a Subclass Definition	120
Overriding Superclass Methods	121
Instance Variables	122
Entry and Exit Points	123
Initialize or Decode?	125
Storing and Accessing Properties	126
Key-Value Mechanisms	131
Object Infrastructure	132

Error Handling	133
Resource Management and Other Efficiencies	134
Functions, Constants, and Other C Types	135
When the Class Is Public (Mac OS X)	136
Multithreaded Cocoa Programs	136
Multithreading and Multiprocessing Resources	137
Multithreading Guidelines for Cocoa Programs	138
Are the Cocoa Frameworks Thread Safe?	140

Chapter 4 Cocoa Design Patterns 141

What Is a Design Pattern?	141
A Solution to a Problem in a Context	141
An Example: The Command Pattern	142
How Cocoa Adapts Design Patterns	143
Abstract Factory	144
Adapter	144
Chain of Responsibility	145
Command	146
Composite	147
Decorator	149
Facade	151
Iterator	152
Mediator	152
Memento	155
Observer	157
Proxy	159
Singleton	159
Template Method	160
The Model-View-Controller Design Pattern	161
Roles and Relationships of MVC Objects	161
Types of Cocoa Controller Objects	164
MVC as a Compound Design Pattern	165
Design Guidelines for MVC Applications	167
Model-View-Controller in Cocoa (Mac OS X)	169
Object Modeling	169
Entities	170
Attributes	171
Relationships	171
Accessing Properties	172

Chapter 5 Communicating With Objects 177

Communication in Object-Oriented Programs	177
Outlets	178
Delegates and Data Sources	179

How Delegation Works	180
The Form of Delegation Messages	181
Delegation and the Cocoa Application Frameworks	182
Data Sources	183
Implementing a Delegate for a Custom Class	183
The Target-Action Mechanism	184
The Target	185
The Action	185
Target-Action in the Application Kit	186
Target-Action in UIKit	189
Bindings (Mac OS X)	190
How Bindings Work	190
How You Establish Bindings	192
Notifications	193
When and How to Use Notifications	194
The Notification Object	196
Notification Centers	196
Notification Queues	197
Ownership of Delegates, Observers, and Targets	200

Appendix A The Core Application Architecture on Mac OS X 201

The Event-Drawing Cycle, Revisited	201
The Global Application Object	202
The Main Event Loop	202
More About Event Dispatch	204
Window Management	204
Handling Apple Events	205
Windows	206
The Windows of an Application	206
NSWindow and the Window Server	207
Window Buffering	207
Window Z-Order and Levels	208
Parts of a Window	210
Window Coordinates	211
Windows and Drawing	213
Window Status	214
Windows and Event Handling	214
Panels	215
Views	216
Varieties of Views	216
The View Hierarchy	217
View Geometry and Coordinates	218
How Views Get Drawn	223
Views and Printing	227
Views and Events	227

Responders and the Responder Chain	228
Controls and Menus	230
Control and Cell Architecture	230
Menu Characteristics and Architecture	233
Represented Objects	234
Nib Files and Other Application Resources	235
Object Archives	235
Nib Files	236
Loading Application Resources	238

Appendix B	Other Cocoa Architectures on Mac OS X	241
-------------------	----------------------------------------------	------------

Document Architecture	241
Application Scriptability	243
Core Data	245

Document Revision History	247
---------------------------	-----

C O N T E N T S

Figures, Tables, and Listings

Chapter 1 What Is Cocoa? 15

Figure 1-1	Mac OS X architecture—simplified perspective	17
Figure 1-2	Cocoa in the architecture of Mac OS X	17
Figure 1-3	Zooming in on the Cocoa architecture—some major dependencies	18
Figure 1-4	Cocoa in the architecture of iPhone OS	19
Figure 1-5	The TextEdit example project in Xcode	26
Figure 1-6	The TextEdit Document Properties window in Interface Builder	28
Figure 1-7	The Interface Builder connections panel	29
Figure 1-8	The Instruments application	31
Figure 1-9	The Foundation class hierarchy	36
Figure 1-10	Application Kit class hierarchy—Objective-C	41
Figure 1-11	UIKit class hierarchy	48
Figure 1-12	UIKit classes and their Application Kit counterparts	52
Figure 1-13	Application Kit class hierarchy in 1988	55
Table 1-1	Major classes of the Application Kit and UIKit	53

Chapter 2 Cocoa Objects 57

Figure 2-1	An object's isa pointer	59
Figure 2-2	Message terminology	68
Figure 2-3	Reachable and unreachable objects	75
Figure 2-4	The life cycle of an object—simplified view	77
Figure 2-5	Retaining a received object	77
Figure 2-6	Copying a received object	78
Figure 2-7	An autorelease pool	79
Figure 2-8	Initialization up the inheritance chain	85
Figure 2-9	Interactions of secondary and designated initializers	87
Figure 2-10	A simple hierarchy for number classes	98
Figure 2-11	A more complete number class hierarchy	99
Figure 2-12	Class cluster architecture applied to number classes	99
Figure 2-13	Embedding a Cluster Object	104
Table 2-1	Important Objective-C defined types and literals	68
Listing 2-1	Output from a simple Cocoa tool	57
Listing 2-2	Cocoa code for a uniquing and sorting tool	57
Listing 2-3	An example of an initializer	83
Listing 2-4	Secondary initializers	86
Listing 2-5	An example dealloc method	88

Listing 2-6	A factory method for a singleton instance	89
Listing 2-7	Using the class and superclass methods	90
Listing 2-8	Using <code>isKindOfClass:</code>	91
Listing 2-9	Using <code>respondsToSelector:</code>	91
Listing 2-10	Using <code>conformsToProtocol:</code>	91
Listing 2-11	Using <code>isEqual:</code>	92
Listing 2-12	Overriding <code>isEqual:</code>	92
Listing 2-13	Returning an immutable copy of a mutable instance variable	96
Listing 2-14	Making a snapshot of a potentially mutable object	97
Listing 2-15	Implementation of a singleton	106

Chapter 3 Adding Behavior to a Cocoa Program 109

Figure 3-1	The main event loop (Mac OS X)	110
Figure 3-2	Invoking a framework method that messages an overridden method	116
Figure 3-3	Object composition	119
Figure 3-4	Where to put declarations in the interface file	120
Listing 3-1	The main function in a Cocoa for Mac OS X application	111
Listing 3-2	The basic structure of an interface file	120
Listing 3-3	The basic structure of an implementation file	121
Listing 3-4	Initialization helper method	125
Listing 3-5	Implementing accessors for a scalar instance variable	129
Listing 3-6	Implementing accessors for an object instance variable (garbage collection enabled)	129
Listing 3-7	Implementing accessors for an object instance variable—good technique	129
Listing 3-8	Implementing accessors for an object instance variable—better technique	130
Listing 3-9	Lazy-loading of a resource	134

Chapter 4 Cocoa Design Patterns 141

Figure 4-1	Structure diagram for the Command pattern	142
Figure 4-2	The view hierarchy, structural and visual	148
Figure 4-3	Framework object sending a message to its delegate	150
Figure 4-4	View controllers in UIKit	155
Figure 4-5	Traditional version of MVC as a compound pattern	166
Figure 4-6	Cocoa version of MVC as compound design pattern	166
Figure 4-7	Coordinating controller as the owner of a nib file	167
Figure 4-8	Employee management application object diagram	170
Figure 4-9	Employees table view	171
Figure 4-10	Relationships in the employee management application	172
Figure 4-11	Relationship cardinality	172
Figure 4-12	Object graph for the employee management application	174
Figure 4-13	Employees table view showing department name	175

Chapter 5 Communicating With Objects 177

Figure 5-1	Connecting an outlet in Interface Builder 179
Figure 5-2	The mechanism of delegation 180
Figure 5-3	A more realistic sequence involving a delegate 181
Figure 5-4	How the target–action mechanism works 187
Figure 5-5	Setting target and action in Interface Builder 188
Figure 5-6	Bindings between view, controller, and model objects 191
Figure 5-7	Establishing a binding in Interface Builder 192
Figure 5-8	Posting and broadcasting a notification 194
Figure 5-9	A notification queue and notification center 198
Table 5-1	Application Kit classes with delegates 182
Listing 5-1	Sample delegation methods with return values 181
Listing 5-2	Sample delegation methods returning <code>void</code> 182

Appendix A The Core Application Architecture on Mac OS X 201

Figure A-1	The event and drawing cycle 201
Figure A-2	Main event loop, with event source 202
Figure A-3	Main event loop, with run-loop observer 203
Figure A-4	The application's window list 205
Figure A-5	Window buffering schemes 208
Figure A-6	Window levels and Z-order 209
Figure A-7	A window's frame view and content view 210
Figure A-8	Relationships among <code>NSApp</code> , windows, and content views 211
Figure A-9	Screen coordinates 212
Figure A-10	Base coordinate system 213
Figure A-11	View hierarchy 217
Figure A-12	Relationships among objects in a view hierarchy 218
Figure A-13	Hierarchical views 219
Figure A-14	Views clipped by the superview 220
Figure A-15	Rotated view and its subview 220
Figure A-16	Flipping a view 221
Figure A-17	Relation between coordinate systems of view and superview 222
Figure A-18	Nested focused views and graphics state stack 226
Figure A-19	<code>NSResponder</code> and its direct subclasses 228
Figure A-20	The responder chain 229
Figure A-21	A control coordinating the drawing of a cell 231
Figure A-22	Interface Builder 237
Table A-1	<code>NSView</code> display methods—immediate display 224
Table A-2	<code>NSView</code> display methods—deferred display 224
Table A-3	<code>NSView</code> display methods—Immediate display of marked views 224
Listing A-1	Using a represented object 234

Appendix B **Other Cocoa Architectures on Mac OS X** 241

- Figure B-1 Ownership relationships among the document classes 242
Figure B-2 Managed object contexts and the persistence stack 246

Introduction to Cocoa Fundamentals Guide

To a developer new to it, Cocoa might seem like a vast, uncharted new world of technology. The features, tools, concepts, designs, terminology, programming interfaces, and even programming language of this development environment may all be unfamiliar. *Cocoa Fundamentals Guide* eases the initial steps to Cocoa proficiency. It provides an orientation to the technological landscape that is Cocoa. It introduces its features, basic concepts, terminology, architectures, and underlying design patterns.

You can build Cocoa applications for two platforms: the Mac OS X operating system and iPhone OS, the operating system for multi-touch devices such as iPhone and iPod touch. *Cocoa Fundamentals Guide* presents Cocoa-related information for both platforms, integrating the information as much as possible and pointing out platform differences when necessary. The intent is that, as you become familiar with Cocoa for one platform, it will become easier to transfer that knowledge to software development for the other platform.

Cocoa Fundamentals Guide is structured to lead gradually to a general understanding of what Cocoa development is all about. It starts with the most basic information—what Cocoa is in terms of its components and capabilities—and ends with an examination of the major architectures. Each chapter builds on what was explained in previous chapters. Each section gives the important details about a subject, yet describes it at only a high level. A section frequently refers the reader to another document that offers a more comprehensive description.

In the set of Cocoa developer documentation, *Cocoa Fundamentals Guide* is the conceptual entry-point document. It is prerequisite reading for other essential Cocoa guides, such as *Cocoa Drawing Guide*, *View Programming Guide for Cocoa*, and *iPhone OS Programming Guide*. *Cocoa Fundamentals Guide* assumes little in terms of prerequisite reading, but readers should be proficient C programmers and should be familiar with the capabilities and technologies of each platform. For Mac OS X, you can acquire this familiarity by reading *Mac OS X Technology Overview*; for iPhone OS, read *iPhone OS Technologies* in *iPhone OS Programming Guide*.

Organization of This Document

Cocoa Fundamentals Guide has the following chapters:

1. [“What Is Cocoa?”](#) (page 15) introduces Cocoa from a functional and broadly architectural perspective, describing its features, frameworks, and development environment.

2. [“Cocoa Objects”](#) (page 57) explains the advantages and basic usage of Objective-C, plus the common behavior, interface, and life cycle of all Cocoa objects.
3. [“Adding Behavior to a Cocoa Program”](#) (page 109) describes what it's like to write a program using a Cocoa framework and explains how to create a subclass.
4. [“Cocoa Design Patterns”](#) (page 141) describes the Cocoa adaptations of design patterns, especially Model-View-Controller and object modeling.
5. [“Communicating With Objects”](#) (page 177) discusses the programming interfaces and mechanisms for communication between Cocoa objects, including delegation, notification, and bindings.

The book includes two appendixes that present information specific to the Mac OS X version of Cocoa:

- [“The Core Application Architecture on Mac OS X”](#) (page 201) examines the network of application objects Cocoa puts into place to facilitate drawing and event handling.
- [“Other Cocoa Architectures on Mac OS X”](#) (page 241) summarizes major architectures of Cocoa that assist application development and expand application capabilities.

See Also

You can find several excellent third-party introductions to Cocoa in technical book stores. You can use these books to supplement what you learn in *Cocoa Fundamentals Guide*. In addition, there are a few other Apple publications that you should also read when starting out as a Cocoa developer:

- *The Objective-C 2.0 Programming Language* describes the Objective-C programming language and runtime environment.
- *Model Object Implementation Guide* discusses basic issues of subclass design and implementation.
- *Cocoa Application Tutorial* shows you how to build a simple Cocoa application for Mac OS X using the Xcode development environment, the Cocoa frameworks, and Objective-C.
- *iPhone OS Programming Guide* presents much information specific to the Cocoa frameworks and the development tools used to develop applications for devices running iPhone OS.

What Is Cocoa?

Cocoa is an application environment for both the Mac OS X operating system and iPhone OS, the operating system used on multi-touch devices such as iPhone and iPod touch. It consists of a suite of object-oriented software libraries, a runtime, and an integrated development environment.

This chapter expands on this definition, describing the purpose, capabilities, and components of Cocoa on both platforms. Reading this functional description of Cocoa is an essential first step toward understanding Cocoa as a developer.

The Cocoa Environment

Cocoa is a set of object-oriented frameworks that provides a runtime environment for applications running on Mac OS X and iPhone OS. It is also part of a development environment that helps you efficiently bring these applications from design stage to deployment. Cocoa is the preeminent application environment for Mac OS X and the *only* application environment for iPhone OS. (Carbon is an alternative environment on Mac OS X, but it is a compatibility framework with procedural programmatic interface intended to support existing Mac OS X code bases.) Most of the applications you see on Mac OS X and iPhone OS, including Mail and Safari, are Cocoa applications. An integrated development environment called Xcode supports application development for both platforms. The combination of this development environment and Cocoa makes it easy to create a well-factored, full-featured application.

Introducing Cocoa

As with all application environments, Cocoa presents two faces; it has a runtime aspect and a development aspect. In its runtime aspect, Cocoa applications present the user interface and are tightly integrated with the other visible portions of the operating system; on Mac OS X, these include the Finder, the Dock, and other applications from all environments.

But it is the development aspect that is the more interesting one to programmers. Cocoa is an integrated suite of object-oriented software components—classes—that enables you to rapidly create robust, full-featured Mac OS X applications. These classes are reusable and adaptable software building blocks; you can use them as-is or extend them for your specific requirements. Cocoa classes exist for just about every conceivable development necessity, from user-interface objects to data formatting, and where a need hasn't been anticipated, you can easily create a subclass of an existing class that answers that need.

Cocoa has one of the most distinguished pedigrees of any object-oriented development environment. From its introduction as NeXTSTEP in 1989 to the present day, it has been continually refined and tested (see [“A Bit of History”](#) (page 54)). Its elegant and powerful design is ideally suited for the rapid development of software of all kinds, not only applications but command-line tools, plug-ins, and various types of bundles. Cocoa gives your application much of its behavior and appearance “for free,” freeing up more of your time to work on those features that are distinctive. (For details on what Cocoa offers, see [“Features of a Cocoa Application”](#) (page 20).)

iPhone OS Note: Cocoa for iPhone OS supports only application development and not any other kind of executable.

You may use several programming languages when developing Cocoa software, but the essential, required language is Objective-C. Objective-C is a superset of ANSI C that has been extended with certain syntactical and semantic features (derived from Smalltalk) to support object-oriented programming. The few added conventions are simple and easy to learn and use. Because Objective-C rests on a foundation of ANSI C, you can freely intermix straight C code with Objective-C code. Moreover, your code can call functions defined in non-Cocoa programmatic interfaces, such as the BSD library interfaces in `/usr/include`. You can even mix C++ code with your Cocoa code and link them into the same executable.

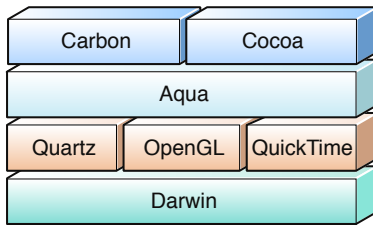
Mac OS X Note: On Mac OS X, you can also program in Cocoa using scripting bridges such as PyObjC (the Python–Objective-C bridge) and RubyCocoa (the Ruby–Cocoa bridge). Both bridged languages let you write Cocoa applications in the respective scripting languages, Python and Ruby. Both of these are interpreted, interactive, and object-oriented programming languages that make it possible for Python or Ruby objects to message Objective-C objects as if they're Python or Ruby objects, and also for Objective-C objects to message Python or Ruby objects. For more information, see *Ruby and Python Programming Topics for Mac OS X*.

The core Cocoa class libraries come packaged in two core frameworks for each platform: Foundation and Application Kit for Mac OS X and Foundation and UIKit for iPhone OS. As with all frameworks, these contain not only a dynamically sharable library (or sometimes several compatibility versions of libraries), but header files, API documentation, and related resources. The duo of both Application Kit and Foundation and UIKit and Foundation reflect the division of the Cocoa programmatic interfaces into those classes that have some bearing on a graphical user interface and those that don't. For each platform, its two core frameworks are essential to any Cocoa project whose end product is an application.

Mac OS X also ships with several other frameworks that vend Cocoa programmatic interfaces, such as the Core Data framework, the Screen Saver and Address Book frameworks, and more will be added to the operating system over time. See [“The Cocoa Frameworks”](#) (page 33) for further information.

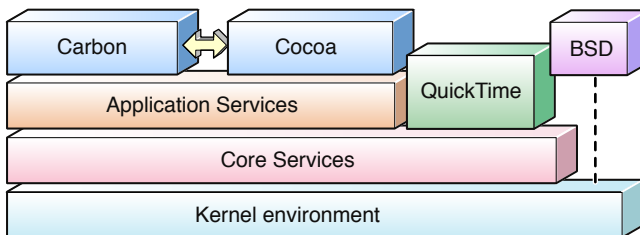
How Cocoa Fits into Mac OS X

Figure 1-1 shows a simplified diagram of the Mac OS X system architecture.

Figure 1-1 Mac OS X architecture—simplified perspective

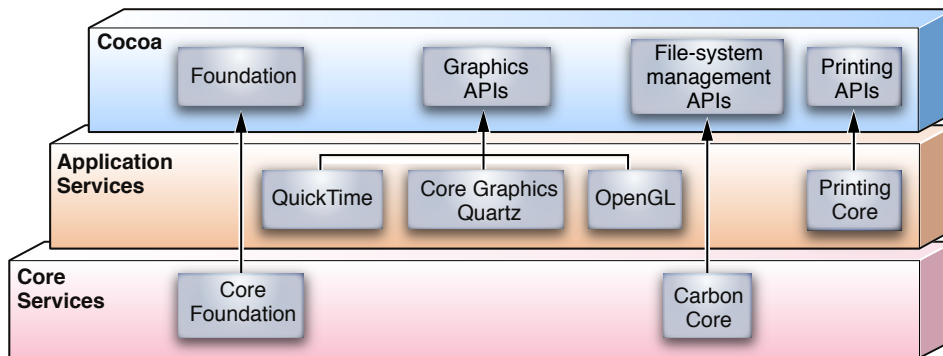
This diagram is simple for a purpose: to depict unambiguously to those unfamiliar with Mac OS X some of its major components and dependencies. But in its simplicity it omits important details and blurs others. These details fill in an important part of the picture showing how Cocoa fits into the rest of Mac OS X.

Figure 1-2 situates Cocoa more accurately in an architectural setting. This diagram shows Mac OS X as a series of software layers going from the foundation of Darwin to the various application environments; the intervening layers represent the system software contained in the two major umbrella frameworks, Core Services and Application Services. The diagram suggests that a component at one layer generally has dependencies on the layer beneath it.

Figure 1-2 Cocoa in the architecture of Mac OS X

In some ways, this diagram is similar to the previous diagram. For example, the system component that is largely responsible for rendering the Aqua user interface, Quartz (implemented in the Core Graphics framework), is part of the Application Services layer. And at the base of the architectural stack is Darwin; everything in Mac OS X, including Cocoa, ultimately depends on Darwin to function.

But if you look closer, at individual or groups of Cocoa classes and at the particular subframeworks of the umbrella frameworks, you begin to see where Cocoa either has specific dependencies on other parts of Mac OS X or where it exposes underlying technology with its interfaces. Figure 1-3 charts a few of these dependencies or exposures.

Figure 1-3 Zooming in on the Cocoa architecture—some major dependencies

Apple has carefully designed Cocoa so that some of its programmatic interfaces give access to the capabilities of underlying technologies that applications typically need. But if you require some capability that is not exposed through the programmatic interfaces of Cocoa, or if you need some finer control of what happens in your application, you may be able to use an underlying framework directly. (A prime example is Core Graphics; by calling its functions or those of OpenGL, your code can draw more complex and nuanced images than is possible with the Cocoa drawing methods.) Fortunately, using these lower-level frameworks is not a problem because the programmatic interfaces of most dependent frameworks are written in standard ANSI C, of which Objective-C language is a superset.

Note: The intent of this architectural overview is not to itemize every particular dependency or interface that Cocoa has on or to other parts of Mac OS X. Instead, it surveys the more interesting ones in order to give you a general idea of the architectural context of the framework.

The major underlying frameworks on which Cocoa depends or which it exposes through its classes and methods are Core Foundation, Carbon, Core Graphics (Quartz), Launch Services, and Print Core (the printing subsystem):

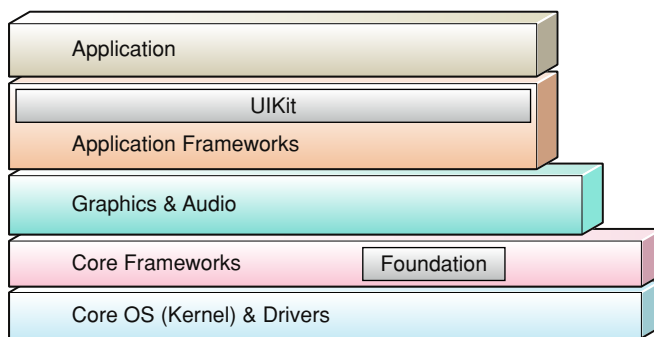
- **Core Foundation.** Many classes of the Foundation framework are based on equivalent Core Foundation opaque types. This close relationship is what makes “toll-free bridging”—cast-conversion between compatible Core Foundation and Foundation types—possible. Some of the implementation of Core Foundation, in turn, is based on the BSD part of the Darwin layer.
- **Carbon.** Cocoa taps into Carbon for some of the services it provides. This is because various Carbon frameworks are situated as system-wide services in the Core Services and Application Services layer. Carbon Core is a particularly important one of these frameworks; for example, it has the File Manager, which Cocoa uses for conversions between various file-system representations.
- **Core Graphics.** The Cocoa drawing and imaging classes are (quite naturally) closely based on the Core Graphics framework, which implements Quartz and the window server.
- **Launch Services.** The `NSWorkspace` class exposes the underlying capabilities of Launch Services. Cocoa also uses the application-registration feature of Launch Services to get the icons associated with applications and documents.
- **Print Core.** The Cocoa printing classes present an object-oriented interface to the printing subsystem.

Further Reading: *Mac OS X Technology Overview* gives an overview of the frameworks, services, technologies, and other components of Mac OS X. *Apple Human Interface Guidelines* specifies how the Aqua human interface should appear and behave.

How Cocoa Fits into iPhone OS

Although the iPhone OS infrastructure on which Cocoa depends is similar to that for Mac OS X, there are some significant differences. Compare Figure 1-4, which depicts the architectural setting of iPhone OS, to the diagram in [Figure 1-2](#) (page 17). The iPhone OS diagram also shows the software supporting its platform as a series of layers going from a Core OS foundation to a set of application frameworks, the most critical (for applications) being the UIKit framework. As in the Mac OS X diagram, the iPhone OS diagram has middle layers consisting of core-services frameworks and graphics and audio frameworks and libraries. Here also, a component at one layer often has dependencies on the layer beneath it.

Figure 1-4 Cocoa in the architecture of iPhone OS



Generally, the system libraries and frameworks of iPhone OS that ultimately support UIKit are a subset of the libraries and frameworks on Mac OS X. For example, there is no Carbon application environment on iPhone OS, there is no command-line access (the BSD environment), there are no printing frameworks and services, and QuickTime is absent from the platform. However, because of the nature of the devices supported by iPhone OS, there are some frameworks, both public and private, that are specific to iPhone OS.

The following summarizes some of the frameworks found at each layer of the iPhone OS stack, starting from the foundation layer.

- **Core OS**—This level contains the kernel, the file system, networking infrastructure, security, power management, and a number of device drivers. It also has the libSystem library, which supports the POSIX/BSD 4.4/C99 API specifications and includes system-level APIs for many services.
- **Core Services**—The frameworks in this layer provide core services, such as string manipulation, collection management, networking, URL utilities, contact management, and preferences. This layer includes Core Foundation, a framework that provides abstractions for common data types (such as strings and collections); they allows a large degree of integration between object-oriented and procedural code.

- **Media**—The frameworks and services in this layer depend on the Core Services layer and provide graphical and multimedia services to the Cocoa Touch layer. They include Core Graphics, OpenGL ES, Core Animation, Core Audio, and video playback.
- **Cocoa Touch**—The frameworks in this layer directly support applications based on iPhone OS. They include two Objective-C frameworks that are particularly important for developing applications for iPhone OS:
 - The UIKit framework provides the objects an application displays in its user interface and defines the structure for application behavior, including event handling and drawing. For a description of UIKit, see “[UIKit \(iPhone OS\)](#)” (page 47).
 - The Foundation framework defines the basic behavior of objects, establishes mechanisms for their management, and provides objects for primitive data types, collections, and operating-system services. Foundation is essentially an object-oriented cover to the Core Foundation framework; see “[Foundation](#)” (page 34) for a discussion of the Foundation framework.

Note: For detailed descriptions of the components in the layers of iPhone OS, see iPhone OS Technologies in *iPhone OS Programming Guide*.

As with Cocoa on Mac OS X, the programmatic interfaces of Cocoa on iPhone OS give your applications access to the capabilities of underlying technologies. Usually there will be a Foundation or UIKit method or function that can tap into a lower-level framework to do what you want. But, as with Cocoa on Mac OS X, if you require some capability that is not exposed through API, or if you need some finer control of what happens in your application, you may choose to use an underlying framework directly. For example, UIKit uses the Web Kit to draw text; however, you may decide to use Core Graphics to draw text because that gives you the control you need for a particular task. Again, using these lower-level frameworks is not a problem because the programmatic interfaces of most dependent frameworks are written in standard ANSI C, of which Objective-C language is a superset.

Further Reading: To learn more about the frameworks, services, and other aspects of the iPhone OS platform, see iPhone OS Technologies in *iPhone OS Programming Guide*.

Features of a Cocoa Application

On Mac OS X it is possible to create a Cocoa application without adding a single line of code. Make a new Cocoa application project using Xcode and then build the project. That's it. Of course, this application won't do much, or at least much that's interesting. But this extremely simple application still launches when double-clicked, displays its icon in the Dock, displays its main menu and window (entitled “Window”), hides itself on command, behaves nicely with other running applications, and quits on command. You can move, resize, minimize, and close the window. You can even print the emptiness contained by the window.

Imagine what you could do with a little code.

iPhone OS Note: The features and behavior of an application running on iPhone OS are considerably different from a Mac OS X application. Because an application runs in a more constrained environment on iPhone OS, it can only display a single window of content at a time, must save state frequently (as it cannot be expected to run in the background), and generally has a more limited set of capabilities than a Mac OS X application. For discussions of application capabilities and constraints on iPhone OS, see *iPhone OS Programming Guide*.

In terms of programming effort, Cocoa gives you, the developer, much that is free and much that is low-cost. Of course, to become a productive Cocoa developer means becoming familiar with possibly new concepts, design patterns, programming interfaces, and development tools, and this effort is not negligible. But familiarity yields greater productivity. Programming becomes largely an exercise in assembling the programmatic components that Cocoa provides along with the custom objects and code that define your program's particular logic, then fitting the whole assemblage together.

What follows is a short list of how Cocoa adds value to an application with only a little (and sometimes no) effort on your part:

- **Basic application framework**—Cocoa provides the infrastructure for event-driven behavior and for application-, window-, and (in the case of Mac OS X) workspace-management. In most cases, you won't have to handle events directly or send any drawing commands to a rendering library.
- **User-interface objects**—Cocoa offers a rich collection of ready-made objects for your application's user interface. Most of these objects are available on palettes of Interface Builder, a development application for creating user interfaces; you simply drag an object from a palette onto the surface of your interface, configure its attributes, and connect it to other objects. (And, of course, you can always instantiate, configure, and connect these objects programmatically.) Here is a sampling of Cocoa user-interface objects:

windows	text fields	image views	date pickers
sheets and dialogs	segmented controls	table views	progress indicators
buttons	sliders	radio buttons (Mac OS X)	color wells (Mac OS X)
drawers (Mac OS X)	page controls (iPhone OS)	navigation bars (iPhone OS)	switch controls (iPhone OS)

Cocoa on Mac OS X also features technologies that support user interfaces, including those that promote accessibility, perform validation, and facilitate the connections between objects in the user interface and custom objects.

- **Drawing and imaging**—Cocoa enables efficient drawing of custom views with a framework for locking graphical focus and marking views (or portions of views) as “dirty.” On Mac OS X, it includes programmatic tools for drawing Bezier paths, performing affine transforms, compositing images, and creating various representations of images.

iPhone OS Note: The UIKit framework provides rudimentary support for drawing rectangles and lines. For drawing of any greater complexity, applications must use the facilities of Core Graphics framework or the OpenGL ES framework.

- **System interaction**—On Mac OS X, Cocoa gives your application ways to interact with (and use the services of) the file system, the workspace, and other applications. On iPhone OS, you can pass system applications URLs to have them handle the referenced resource (for example, email or websites).
- **Performance**—To enhance the performance of your application, Cocoa provides programmatic support for multithreading, idle-time processing, lazy loading of resources, memory management, and run-loop manipulation.
- **Internationalization**—Cocoa provides a rich architecture for internationalizing applications, making it possible for you to support localized resources such as text, images, and even user interfaces. The Cocoa approach is based on users' lists of preferred languages and puts localized resources in bundles of the application. Based on the settings it finds, Cocoa automatically selects the localized resource that best matches the user's preferences. It also provides tools and programmatic interfaces for generating and accessing localized strings. Moreover, text manipulation in Cocoa is based on Unicode by default, and is thus an asset for internationalization.
- **Text**—On Mac OS X, Cocoa provides a sophisticated text system that allows you to do things with text ranging from the simple (for example, displaying a text view with editable text) to the more complex, such as control of kerning and ligatures, spell checking, and embedding images. On iPhone OS, Cocoa has no native text system (it uses Web Kit for string drawing) and text capabilities are more limited.
- **Preferences**—The user defaults system is based on a system-wide database in which you can store global and application-specific preferences. The procedure for specifying application preferences is different on the two platforms.
- **Networking**—Cocoa also offers programmatic interfaces for communicating with servers using standard Internet protocols, communicating via sockets, and incorporating Bonjour capabilities in your application.

On Mac OS X, Cocoa includes a distributed objects architecture that allows one Cocoa process to communicate with another process on the same computer or on a different one.

Cocoa on Mac OS X has several other additional features:

- **Data exchange**—Cocoa simplifies the exchange of data within an application and between applications using the copy-paste and drag-and-drop models and through the Services menu.
- **Document-based applications**—Cocoa specifies an architecture for applications composed of a potentially unlimited number of documents, with each contained in its own window (a word processor, for example). Indeed, if you choose the “Document-based application” project type, many of the components of this sort of application are created for you.
- **Scripting**—Through application scriptability information and a suite of supporting Cocoa classes, you can make your application scriptable; that is, it can respond to commands emitted by AppleScript scripts. Applications can also execute scripts or use individual Apple events to send commands to, and receive data from, other applications. As a result, every scriptable application can supply services to both users and other applications.
- **Undo management**—You can register user actions that occur with an undo manager, and it will take care of undoing them (and redoing them) when users choose the appropriate menu items. The manager maintains undo and redo operations on separate stacks.

- **Printing**—In a fashion similar to the text system, the printing architecture lets you print documents and other application content along a range of control and sophistication. At the simplest level, you can print the contents of any view by default. At a more complicated level, you can define the content and format of printed content, control how a print job is performed, and add an accessory view to the print panel.
- **Multimedia**—On Mac OS X, Cocoa provides support for QuickTime video and basic audio capabilities.

The Development Environment

To say that Cocoa has its own development environment wouldn't quite be an accurate statement. For one thing, programmers can use Apple's major development applications, Xcode and Interface Builder, to develop software for Mac OS X applications based on Carbon. Moreover, it is possible to develop Cocoa applications without using Xcode and Interface Builder at all. For example, you could write code using a text editor such as Emacs, build the application from the command line using make files, and debug the application from the command line using the `gdb` debugger.

Note: "Xcode" is sometimes used to refer to the complete suite of development tools and frameworks, and other times specifically to the IDE application that allows you to manage projects and build executable code.

But Xcode and Interface Builder are the preferred applications to use for developing Cocoa software. Their origins coincide with the origins of Cocoa itself, and consequently there is a high degree of compatibility between tools and frameworks. Together, Xcode and Interface Builder make it extraordinarily easy to design, manage, build, and debug Cocoa software projects.

When you install the development tools and documentation, you may select the installation location. Traditionally that location has been `/Developer`, but it can be anywhere in the file system you wish. As a shorthand for designating this installation location, the documentation uses `<Xcode>`. Thus, the development applications are installed in `<Xcode>/Applications`.

Platform SDKs

Beginning with Xcode 3.1 and the introduction of iPhone OS, when you create a software project you must choose a platform SDK. The SDK enables you to build an executable that is targeted for a particular release of a particular operating system, or platform: Mac OS X or iPhone OS.

The platform SDK contains everything that is required for developing software for a given platform and operating-system release. A Mac OS X SDK consists of frameworks, libraries, header files, and system tools. The SDK for iPhone OS has the same components, but includes a platform-specific compiler and other tools. There is also a separate SDK for the Simulator environment for iPhone OS (see ["The iPhone OS Simulator"](#) (page 29)). All SDKs include build settings and project templates appropriate to their platform.

Further reading: For more on platform SDKs, see *Xcode Overview*.

Overview of Development Workflows

Application development differs for Mac OS X and iPhone OS, not only the tools used but in the development workflow. On Mac OS X, the typical development workflow is the following:

1. In Xcode, create a project using a template from the Mac OS X SDK.
2. Write code and, using Interface Builder, construct your application's user interface.
3. Define the targets and executable environment for your project.
4. Test and debug the application using the Xcode debugging facilities.

As part of debugging, you can check the system logs in the Console window.

5. Measure application performance using one or more of the available performance tools.

For iPhone OS development, the workflow when developing an application is a bit more complex. Before you can develop for iPhone OS, you must register as a developer for the platform. Thereafter, building an application that's ready to deploy should go through the following steps:

1. Configure the remote device. This results in the required tools, frameworks, and other components being installed on the device.
2. In Xcode, create a project using a template from the iPhone OS SDK.
3. Write code, and construct your application's user interface.
4. Define the targets and executable environment for the project.
5. Build the application (locally).
6. Test and debug the application, either in the iPhone OS Simulator or remotely in the device. (If remotely, your debug executable is downloaded to the device.)

As you debug, you can check the system logs for the device in the Console window.

7. Measure application performance using one or more of the available performance tools.

Further reading: For more on the development workflow, see Development Environment in *iPhone OS Programming Guide*.

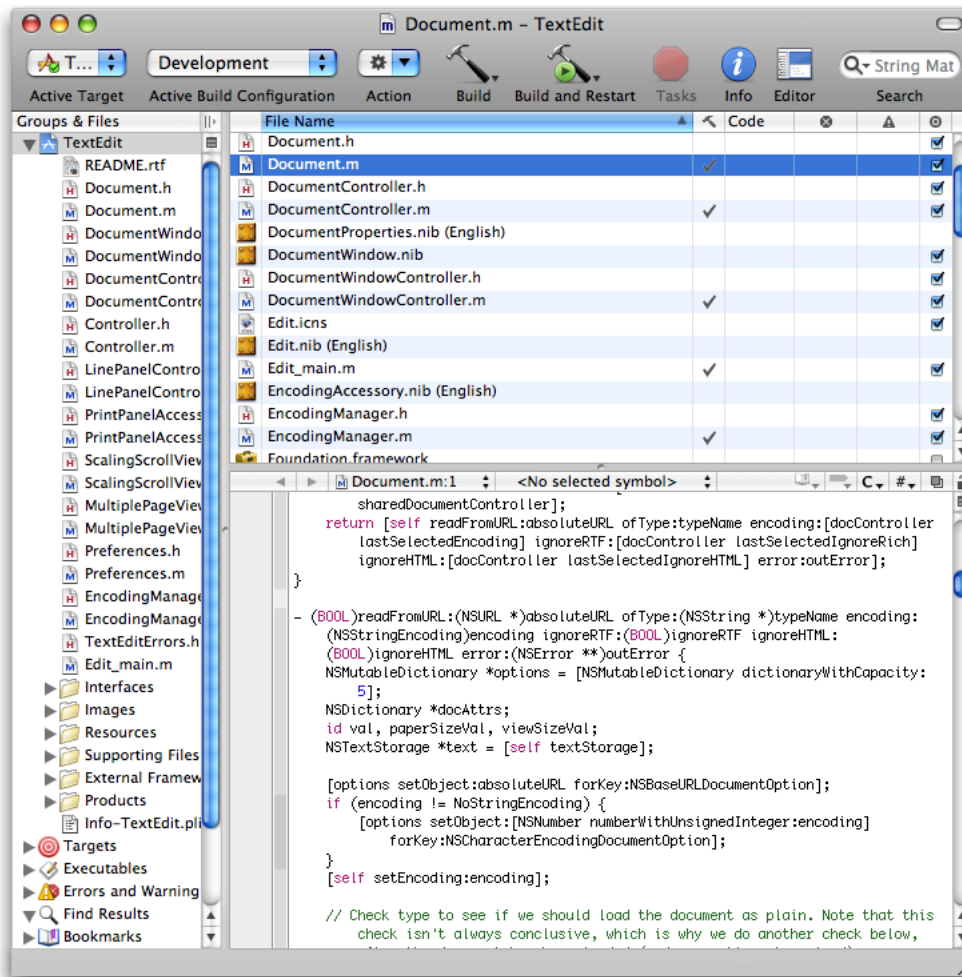
Xcode

Xcode is the engine that powers Apple's integrated development environment (IDE) for Mac OS X and iPhone OS. It is also an application that takes care of most project details from inception to deployment. It allows you to

- Create and manage projects, including specifying platforms, target requirements, dependencies, and build configurations.
- Write source code in editors with features such as syntax coloring and automatic indenting.
- Navigate and search through the components of a project, including header files and documentation.
- Build the project.
- Debug the project locally, in the iPhone OS simulator, or remotely, in a graphical source-level debugger.

Xcode builds projects from source code written in C, C++, Objective-C, and Objective-C++. It generates executables of all supported types on Mac OS X, including command-line tools, frameworks, plug-ins, kernel extensions, bundles, and applications. (For iPhone OS, only application executables are possible.) Xcode permits almost unlimited customization of build and debugging tools, executable packaging (including information property lists and localized bundles), build processes (including copy-file, script-file, and other build phases), and the user interface (including detached and multi-view code editors). It also supports several source-code management systems—namely CVS, Subversion and Perforce—allowing you to add files to a repository, commit changes, get updated versions, and compare versions.

Figure 1-5 shows an example of a project in Xcode.

Figure 1-5 The TextEdit example project in Xcode

Xcode is especially suited for Cocoa development. When you create a project, Xcode sets up your initial development environment using project templates corresponding to Cocoa project types: application, document-based application, Core Data application, tool, bundle, framework, and others. For compiling Cocoa software for Mac OS X, Xcode uses the GNU C compiler (`gcc`), and for debugging that software, it uses the GNU source-level debugger (`gdb`). Both `gcc` and `gdb` have been used in Cocoa development since Cocoa was NeXTSTEP (see “A Bit of History” (page 54)), and over the years have been refined, extended, and tuned to support the compilation and debugging of Cocoa binaries.

Xcode is well integrated with the other major development application, Interface Builder. See “Interface Builder” (below) for details.

Further Reading: *Xcode Overview* gives an overview of Xcode and provides links to additional development-tools documentation.

Interface Builder

The second major development application for Cocoa projects built for Mac OS X is Interface Builder. As its name suggests, Interface Builder is a graphical tool for creating user interfaces. Interface Builder has been around almost since the inception of Cocoa as NeXTSTEP. Not surprisingly, its integration with Cocoa is airtight. Moreover, you can also use it to create user interfaces for Carbon applications for Mac OS X.

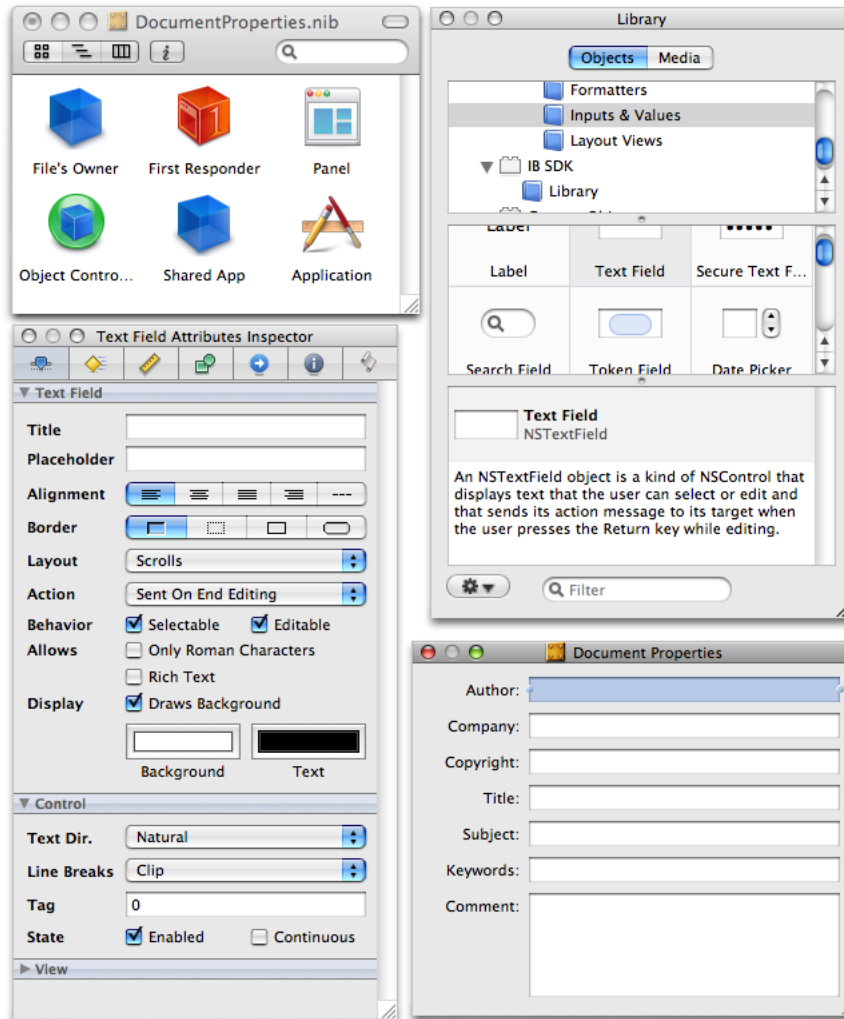
Interface Builder (version 3.0) is centered around four main design elements:

- **Nib files.** A nib file is actually a file wrapper (an opaque directory) that contains the objects appearing on a user interface in an archived form. Essentially, this archive is an object graph that contains information on each object, including its size and location, on the connections between objects, and on proxy references for custom classes. When you create and save a user interface in Interface Builder, all information necessary to re-create the interface is stored in the nib file.

Nib files offer a way to easily localize user interfaces. Interface Builder stores a nib file in a localized directory inside a Cocoa project; when that project is built, the nib file is copied to a corresponding localized directory in the created bundle.

Interface Builder presents the contents of a nib file in a nib document window (also called a nib file window). The nib document window gives you access to the important objects in a nib file, especially top-level objects such as windows, menus, and controller objects that have no parent in their object graph. (Controller objects mediate between user-interface objects and the model objects that represent application data; they also provide overall management for an application.)

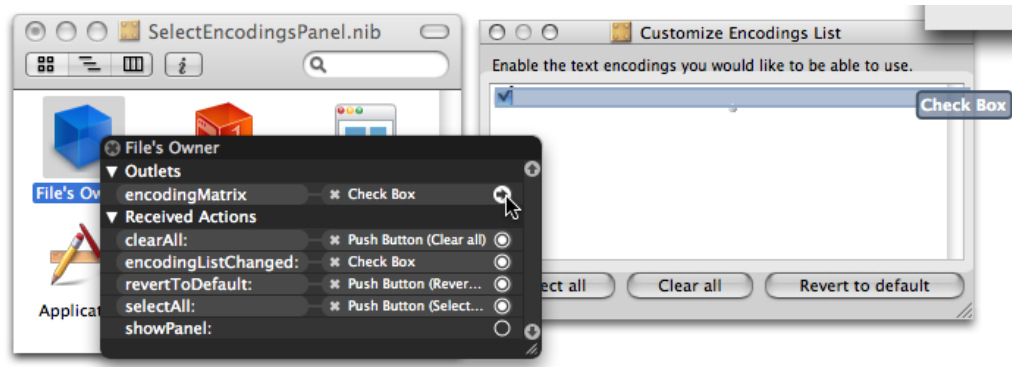
Figure 1-6 shows a nib file opened in Interface Builder and displayed in a nib document window, along with supporting windows.

Figure 1-6 The TextEdit Document Properties window in Interface Builder

- **Object library.** The Library window of Interface Builder contains objects that you can place on a user interface. They range from typical UI objects—for example, windows, controls, menus, text views, and outline views—to controller objects, custom view objects, and framework-specific objects, such as the Image Kit browser view. The Library groups the objects by categories and lets you browse through them and search for specific objects. When an object is dragged from the Library onto an interface, Interface Builder instantiates a default instance of that object; You can resize, configure, and connect the object to other objects using the Inspector window (or, simply, the inspector).
- **Inspector.** Interface Builder has an Inspector window (or simply, the inspector) for objects on a user interface. The inspector has a number of selectable panes for setting the initial runtime configuration of objects (although size and some attributes can be set by direct manipulation). The inspector in Figure 1-6 shows the primary attributes for a text field; note that different collapsible slices of the pane reveal attributes at various levels of the inheritance hierarchy (text field, control, and view). In addition to primary attributes and size, the inspector features panes for animation effects, AppleScript event handlers, and bindings and target-action connections between objects

- **Connections panel**. The connections panel is a context-sensitive display that shows the current outlet and action connections for a selected object and lets you manage those connections. To get the connections panel to appear, right-click on the target object (Control-click on single-button mice). Figure 1-7 shows what the connections panel looks like.

Figure 1-7 The Interface Builder connections panel



Interface Builder uses momentary blue lines to show the compliance of each positioned object, when moved or resized, to the Aqua human interface guidelines. This compliance includes recommended size, alignment, and position relative to other objects on the user interface and to the boundaries of the window.

Interface Builder is tightly integrated with Xcode. It “knows” about the outlets, actions, and bindable properties of your custom classes. When you add, remove, or modify any of these things, Interface Builder detects those changes and updates its presentation of them.

Further Reading: For further information on Interface Builder, see *Interface Builder User Guide*. In addition, “[Nib Files](#)” (page 236) gives more information about nib files and their use in an application. Also refer to “[Communicating With Objects](#)” (page 177) for overviews of outlets, the target-action mechanism, and the Cocoa bindings technology.

The iPhone OS Simulator

For iPhone OS projects, you can select the iPhone OS Simulator as the platform SDK for the project. When you build and run the project, Xcode runs the Simulator, which presents your application as it would appear on the device and allows you to manipulate parts of the user interface. You can use the Simulator to help you debug the application prior to loading it onto the device.

You should always perform the final phase of debugging on the device. The Simulator does not perfectly simulate the device. For example, you must use the mouse pointer instead of finger touches, and so manipulations of the interface required multiple fingers are not possible. In addition, the Simulator does not use versions of the OpenGL framework that are specific to iPhone OS, and it uses the Mac OS X versions of the Foundation, Core Foundation, and CFNetwork frameworks, as well as the Mac OS X version of libSystem.

More importantly, you should not assume that the performance of your application on the Simulator is the same as it would be on the device. The Simulator is essentially running your iPhone application as a "guest" Mac OS X application. As such, it has a 4GB memory partition and swap space available to it as it runs on a processor that is more powerful than the one on the device.

Further reading: For more on the Simulator, see Development Environment in *iPhone OS Programming Guide*.

Performance Applications and Tools

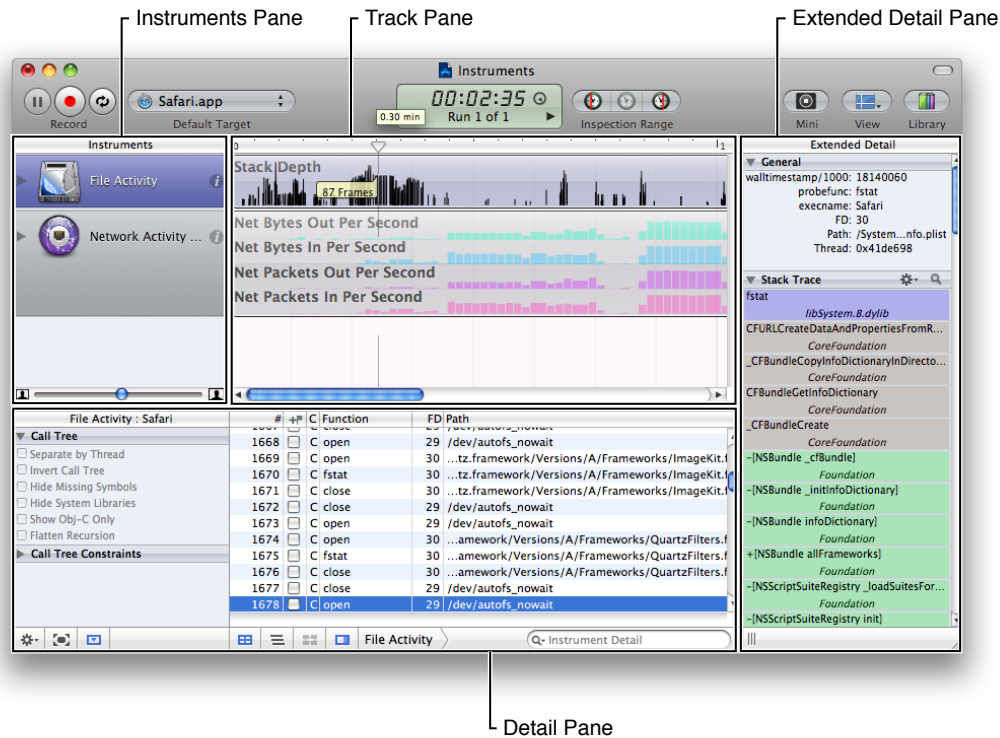
Although Xcode and Interface Builder are the major tools you use to develop Cocoa applications, there are dozens of other applications at your disposal. Many of these tools are performance applications.

Instruments

Instruments is an application introduced in Xcode 3.0 that lets you run multiple performance-testing tools simultaneously and view the results in a timeline-based graphical presentation. It can show you CPU usage, disk reads and writes, memory statistics, thread activity, garbage collection, network statistics, directory and file usage, and other measurements—individually or in different combinations—in the form of graphs tied to time. This simultaneous presentation of instrumentation data helps you to discover the relationships between what is being measured. It also displays the specific data behind the graphs.

Further Reading: See the *Instruments User Guide* for complete information about the Instruments application.

Figure 1-8 The Instruments application



Shark

Shark is a performance-analysis application that creates a time-based profile of your program's execution; over a given period it traces function calls and graphs memory allocations. You can use Shark to track information for a single program or for the entire system, which on Mac OS X includes kernel components such as drivers and kernel extensions. Shark also monitors file-system calls, traces system calls and memory allocations, performs static analyses of your code, and gathers information about cache misses, page faults, and other system metrics. Shark supports the analysis of code written in C, Objective-C, C++, and other languages.

Other Performance Applications (Mac OS X)

Many applications are used in measuring and analyzing aspects of a Mac OS X program's performance. They are located in <Xcode>/Applications/Performance Tools.

- **Thread Viewer** displays activity among a process's threads. It shows time lines of activity on each thread, which is color-coded with the action. By clicking a time line, you can get a sample backtrace of activity at that point.
- **BigTop** graphs performance trends over time, providing a real-time display of memory usage, page faults, CPU usage, and other data.

- **Spin Control** automatically samples unresponsive applications. You leave Spin Control running in the background while you launch and test your applications. If applications become unresponsive to the point where the spinning cursor appears, Spin Control automatically samples your application to gather information about what your application was doing during that time.
- **MallocDebug** shows all currently allocated blocks of memory in your program, organized by the call stack at the time of allocation. At a glance you can see how much allocated memory your application consumes, where that memory was allocated from, and which functions allocated large amounts of memory. MallocDebug can also find allocated memory that is not referenced elsewhere in the program, thus helping you find leaks and track down exactly where the memory was allocated.
- **QuartzDebug** is a tool to help you debug how your application displays itself. It is especially useful for applications that do significant amounts of drawing and imaging. QuartzDebug has several debugging options, including the following:
 - Auto-flush drawing, which flushes the contents of graphics contexts after each drawing operation)
 - A mode that paints regions of the screen in yellow just before they're updated
 - An option that takes a static snapshot of the system-wide window list, giving the owner of each window and how much memory each window consumes.

For performance analysis, you can also use command-line tools such as:

- `top`, which shows a periodically sampled set of statistics on currently running processes
- `gprof`, which produces an execution profile of a program
- `fs_usage`, which displays file-system access statistics

Many other command-line tools for performance analysis and other development tasks are available. Some are located in `/usr/bin` and `/usr/sbin`, and some Apple-developed command-line tools are installed in `<Xcode>/Tools`. For many of these tools you can consult their manual page for usage information. (To do this, either choose **Help > Open man page in Xcode** or type `man` followed by the name of the tool in a Terminal shell.)

Further Reading: For more on the performance tools and applications you can use in Cocoa application development, as well as information on concepts, techniques, guidelines, and strategy related to performance, see *Performance Overview. Cocoa Performance Guidelines* covers the performance guidelines for Cocoa.

AppleScript Studio (Mac OS X)

A defining feature of the Mac OS for years has been the capability for users to control applications with scripts written in the AppleScript language. Many users find this feature indispensable, for it allows them to string together complicated series of related operations involving multiple applications. AppleScript capabilities have been carried forward into Mac OS X. AppleScript Studio is a development technology for creating Cocoa applications that use AppleScript scripts to control complex user interfaces.

AppleScript Studio combines elements from AppleScript, Xcode, Interface Builder, and Cocoa to provide a sophisticated environment for creating AppleScript solutions. It allows you to build applications that can do the following:

- Execute AppleScript scripts.
- Control the user interface of the application.
- Control scriptable applications or scriptable parts of the operating system.

Because AppleScript Studio integrates AppleScript with Xcode, Interface Builder, and Cocoa, scripters can take advantage of their particular strengths and capabilities. They can drag a rich set of user-interface objects off Interface Builder palettes and customize them to their liking. They get built-in support for the Aqua human interface guidelines. And they are able to build and maintain complex projects with multiple targets and build steps.

The development environment makes possible scripting capabilities that go well beyond those available in the Script Editor application traditionally used for creating AppleScript scripts. These include:

- creation of arbitrarily large scripts
- search and replace in scripts
- single-step script debugging with variable execution
- easy access to handlers and properties in scripts
- a flexible dictionary viewer for working with application scripting terminologies

Further Reading: For more information, see the *AppleScript Studio Programming Guide*.

The Cocoa Frameworks

What makes a program a Cocoa program? It's not really the language, because you can use a variety of languages in Cocoa development. It's not the development tools, because you could create a Cocoa application from the command line (although that would be a complex, time-consuming task). No, what all Cocoa programs have in common—what makes them distinctive—is that they are composed of objects that inherit ultimately from the root class, `NSObject`, and that are ultimately based upon the Objective-C runtime. This statement is also true of all Cocoa frameworks.

Note: That statement also needs to be qualified a bit. First, the Foundation framework supplies another root class, `NSProxy`; however, `NSProxy` is rarely used in Cocoa programming. Second, you could create your own root class, but this would be a lot of work (entailing the writing of code that interacts with the Objective-C runtime) and probably not worth your time.

Mac OS X includes several Cocoa frameworks, and Apple and third-party vendors are releasing more frameworks all the time. Despite this abundance of Cocoa frameworks, two of them stand out on each platform as core frameworks:

- On Mac OS X: Foundation and Application Kit
- On iPhone OS: Foundation and UIKit

The Foundation, Application Kit, and UIKit frameworks are essential to Cocoa development, and all other frameworks are secondary and elective. You cannot develop a Cocoa application for Mac OS X unless you link against (and use the classes of) the Application Kit, and you cannot develop a Cocoa application for iPhone OS unless you link against (and use the classes of) UIKit. Moreover, you cannot develop Cocoa software of any kind unless you link against and use the classes of the Foundation framework. Linking against the right frameworks on Mac OS X happens automatically when you link against the Cocoa umbrella framework for Mac OS X.) Classes, functions, data types, and constants in Foundation and the Application Kit have a prefix of “NS”; classes, functions, data types, and constants in UIKit have a prefix of “UI”.

Note: In Mac OS X version 10.5 the Cocoa frameworks have been ported to support 64-bit addressing. iPhone OS also supports 64-bit addressing. As part of this effort, various general changes have been made to the Cocoa API, most significantly the introduction of the `NSInteger` and `NSUInteger` types (replacing `int` and `unsigned int` where appropriate) and the `CGFloat` type (replacing most instances of `float`). Most Cocoa applications have no immediate need to make the transition to 64-bit, but for those that do, porting tools and guidelines are available. *64-Bit Transition Guide for Cocoa* discusses these matters in detail.

The following sections surveys the features and classes of the three core Cocoa frameworks and briefly describes some of the secondary frameworks. To make these large frameworks more approachable, the introductions to the Foundation, Application Kit, and UIKit frameworks break down the dozens of classes in each hierarchy into functional groupings. Although these groupings have a strong logical basis, one can plausibly group classes in other ways.

Foundation

The Foundation framework defines a base layer of classes that can be used for any type of Cocoa program. The criterion separating the classes in Foundation from those in the Application Kit is the user interface. If an object doesn't either appear in a user interface or isn't *exclusively* used to support a user interface, then its class belongs in Foundation. You can create Cocoa programs that use Foundation and no other framework; examples of these are command-line tools and Internet servers.

The Foundation framework was designed with certain goals in mind:

- Define basic object behavior and introduce consistent conventions for such things as memory management, object mutability, and notifications.
- Support internationalization and localization with (among other things) bundle technology and Unicode strings.
- Support object persistence.
- Support object distribution.
- Provide some measure of operating-system independence to support portability.
- Provide object wrappers or equivalents for programmatic primitives, such as numeric values, strings, and collections. Also include utility classes for accessing underlying system entities and services, such as ports, threads, and file systems.

Cocoa applications, which by definition link either against the Application Kit or the UIKit frameworks, invariably must link against the Foundation framework as well. The class hierarchies share the same root class, `NSObject`, and many if not most of the Application Kit and UIKit methods and functions

have Foundation objects as parameters or return values. Some Foundation classes may seem designed for applications—`NSUndoManager` and `NSUserDefaults`, to name two—but they are included in Foundation because there can be uses for them that do not involve a user interface.

Foundation Paradigms and Policies

Foundation introduces several paradigms and policies to Cocoa programming to ensure consistent behavior and expectations among the objects of a program in certain situations. These include:

- **Object retention and object disposal.** The Objective-C runtime and Foundation give Cocoa programs two ways to ensure that objects persist when they're needed and are freed when they are no longer needed. Garbage collection, which was introduced in Objective-C 2.0, automatically tracks and disposes of objects that your program no longer needs, thus freeing up memory. Foundation also still offers the traditional approach of memory management. It institutes a policy of object ownership that specifies that objects are responsible for releasing other objects that they have created, copied, or explicitly retained. `NSObject` (class and protocol) defines methods for retaining and releasing objects. Autorelease pools (defined in the `NSAutoreleasePool` class) implement a delayed-release mechanism and enable Cocoa programs to have a consistent convention for returning objects for which the caller is not responsible. For more about garbage collection and explicit memory management, see [“Object Retention and Disposal”](#) (page 73).

iPhone OS Note: For performance reasons, garbage collection is not allowed in Cocoa applications running on iPhone OS.

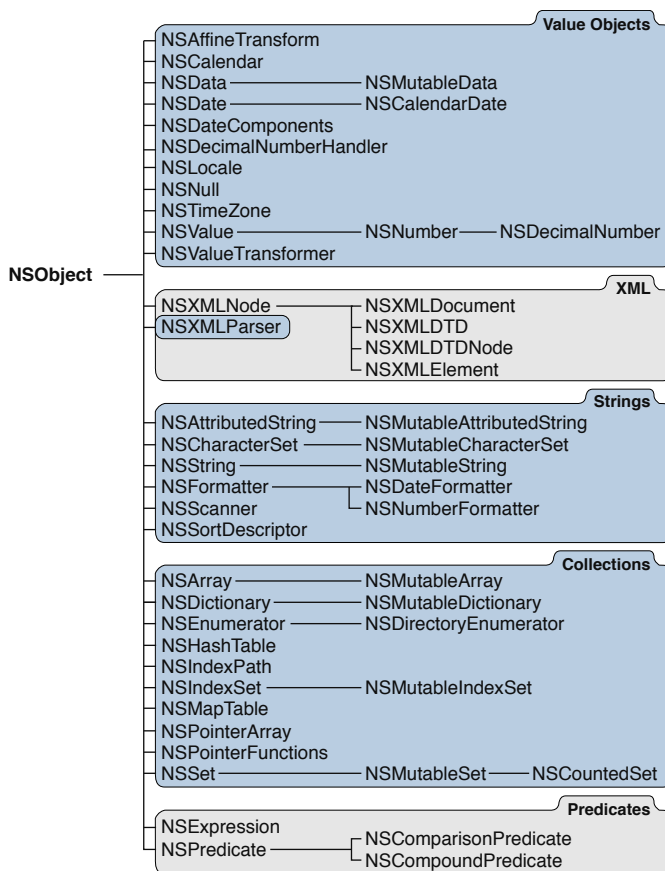
- **Mutable class variants.** Many value and container classes in Foundation have a mutable variant of an immutable class, with the mutable class always being a subclass of the immutable one. If you need to dynamically change the encapsulated value or membership of such an object, you create an instance of the mutable class. Because it inherits from the immutable class, you can pass the mutable instance in methods that take the immutable type. For more on object mutability, see [“Object Mutability”](#) (page 93).
- **Class clusters.** A class cluster is an abstract class and a set of private concrete subclasses for which the abstract class acts as an umbrella interface. Depending on the context (particularly the method you use to create an object), an instance of the appropriate optimized class is returned to you. `NSString` and `NSMutableString`, for example, act as brokers for instances of various private subclasses optimized for different kinds of storage needs. Over the years the set of concrete classes has changed several times without breaking applications. For more on class clusters, see [“Class Clusters”](#) (page 98).
- **Notifications.** Notification is a major design pattern in Cocoa. It is based on a broadcast mechanism that allows objects (called observers) to be kept informed of what another object is doing or is encountering in the way of user or system events. The object originating the notification can be unaware of the existence or identity of the observers of the notification. There are several types of notifications: synchronous, asynchronous, and distributed. The Foundation notification mechanism is implemented by the `NSNotification`, `NSNotificationCenter`, `NSNotificationQueue`, and `NSDistributedNotificationCenter` classes. For more on notifications, see [“Notifications”](#) (page 193).

Foundation Classes

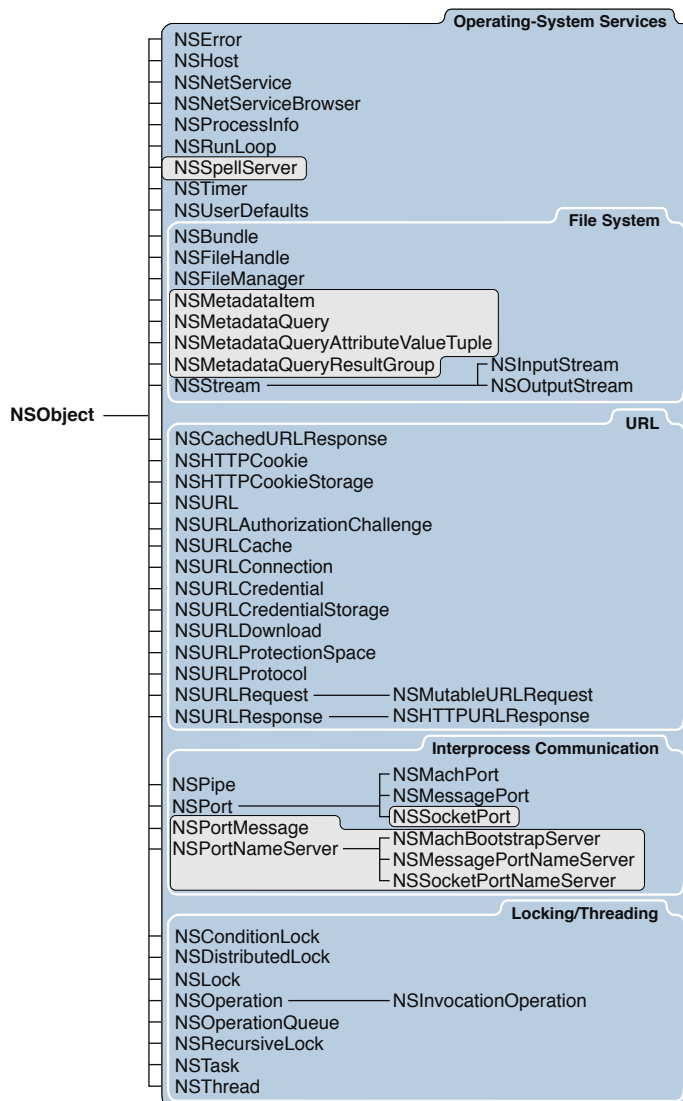
The Foundation class hierarchy is rooted in the `NSObject` class, which (along with the `NSObject` and `NSCopying` protocols) define basic object attributes and behavior. For further information on `NSObject` and basic object behavior, see [“The Root Class”](#) (page 69).

The remainder of the Foundation framework consists of several related groups of classes as well as a few individual classes. There are classes representing basic data types such as strings and byte arrays, collection classes for storing other objects, classes representing system information such as dates, and classes representing system entities such as ports, threads, and processes. The class hierarchy charts in Figure 1-9 (for printing purposes, in three parts) depict the logical groups these classes form as well as their inheritance relationships. Classes in blue-shaded areas are present in both the Mac OS X and iPhone OS versions of Foundation.

Figure 1-9 The Foundation class hierarchy



Objective-C Foundation Continued



```

graph LR
    NSObject --- NSNotification
    NSObject --- NSNotificationCenter
    NSObject --- NSNotificationQueue
    NSObject --- NSDistributedNotificationCenter
    NSObject --- NSCoder
    NSObject --- NSPropertyListSerialization
    NSObject --- NSArchiver
    NSObject --- NSKeyedArchiver
    NSObject --- NSKeyedUnarchiver
    NSObject --- NSPortCoder
    NSObject --- NSUnarchiver
    NSObject --- NSAssertionHandler
    NSObject --- NSAutoreleasePool
    NSObject --- NSClassDescription
    NSObject --- NSException
    NSObject --- NSGarbageCollector
    NSObject --- NSInvocation
    NSObject --- NSMethodSignature
    NSObject --- NSUndoManager
    NSObject --- NSAppleEventDescriptor
    NSObject --- NSAppleEventManager
    NSObject --- NSAppleScript
    NSObject --- NSPositionalSpecifier
    NSObject --- NSScriptCoercionHandler
    NSObject --- NSScriptCommand
    NSObject --- NSScriptCommandDescription
    NSObject --- NSScriptExecutionContext
    NSObject --- NSScriptObjectSpecifier
    NSObject --- NSScriptSuiteRegistry
    NSObject --- NSScriptWhoseTest
    NSObject --- NSScriptClassDescription
    NSObject --- NSCloneCommand
    NSObject --- NSCloseCommand
    NSObject --- NSCountCommand
    NSObject --- NSCreateCommand
    NSObject --- NSDeleteCommand
    NSObject --- NSExistsCommand
    NSObject --- NSGetCommand
    NSObject --- NSMoveCommand
    NSObject --- NSQuitCommand
    NSObject --- NSSetCommand
    NSObject --- NSIndexSpecifier
    NSObject --- NSMiddleSpecifier
    NSObject --- NSNameSpecifier
    NSObject --- NSPropertySpecifier
    NSObject --- NSRandomSpecifier
    NSObject --- NSRangeSpecifier
    NSObject --- NSRelativeSpecifier
    NSObject --- NSUniqueIDSpecifier
    NSObject --- NSWhoseSpecifier
    NSObject --- NSLogicalTest
    NSObject --- NSSpecifierTest
    NSObject --- NSConnection
    NSObject --- NSDistantObjectRequest
    NSObject --- NSDistantObject
    NSObject --- NSProtocolChecker
  
```

NSObject

- Notifications**
 - NSNotification
 - NSNotificationCenter
 - NSDistributedNotificationCenter
 - NSNotificationQueue
- Archiving and Serialization**
 - NSCoder
 - NSArchiver
 - NSKeyedArchiver
 - NSKeyedUnarchiver
 - NSPortCoder
 - NSUnarchiver
 - NSPropertyListSerialization
- Objective-C Language Services**
 - NSAssertionHandler
 - NSAutoreleasePool
 - NSClassDescription
 - NSException
 - NSGarbageCollector
 - NSInvocation
 - NSMethodSignature
 - NSUndoManager
- Scripting**
 - NSAppleEventDescriptor
 - NSAppleEventManager
 - NSAppleScript
 - NSPositionalSpecifier
 - NSScriptCoercionHandler
 - NSScriptCommand
 - NSScriptClassDescription
 - NSCloneCommand
 - NSCloseCommand
 - NSCountCommand
 - NSCreateCommand
 - NSDeleteCommand
 - NSExistsCommand
 - NSGetCommand
 - NSMoveCommand
 - NSQuitCommand
 - NSSetCommand
 - NSScriptCommandDescription
 - NSIndexSpecifier
 - NSMiddleSpecifier
 - NSNameSpecifier
 - NSPropertySpecifier
 - NSRandomSpecifier
 - NSRangeSpecifier
 - NSRelativeSpecifier
 - NSUniqueIDSpecifier
 - NSWhoseSpecifier
 - NSScriptExecutionContext
 - NSScriptObjectSpecifier
 - NSLogicalTest
 - NSSpecifierTest
 - NSScriptSuiteRegistry
 - NSScriptWhoseTest
- Distributed Objects**
 - NSConnection
 - NSDistantObjectRequest
 - NSDistantObject
 - NSProtocolChecker

- **Value objects.** Value objects encapsulate data of various types, giving access to the data and offering various manipulations of it. Because they are objects, they (and their contained values) can be archived and distributed. `NSData` provides object-oriented storage for streams of bytes whereas `NSNumber` and `NSValue` provide object-oriented storage for arrays of simple scalar values. The `NSDate`, `NSCalendarDate`, `NSTimeZone`, `NSCalendar`, `NSDateComponents`, and `NSLocale` classes provide objects that represent times, dates, calendar, and locales. They offer methods for calculating date and time differences, for displaying dates and times in many formats, and for adjusting times and dates based on location in the world.
- **Strings.** `NSString` is another type of value object that provides object-oriented storage for a null-terminated array of bytes in a particular encoding. It includes support for converting string encodings among UTF-16, UTF-8, and many other encodings. `NSString` also offers methods for searching, combining, and comparing strings and for manipulating file-system paths. You can

use an `NSString` object to parse numbers and words from an `NSString` object. `NSCharacterSet` (shown as a collection class in the diagram) represents a set of characters that are used by various `NSString` and `NSString` methods.

- **Collections.** Collections are objects that store and vend other (usually value) objects in a particular ordering scheme. `NSArray` uses zero-based indexing, `NSDictionary` uses key-value pairs, and `NSSet` provides unordered storage of objects (`NSCountedSet` “uniques” the collection). With an `NSEnumerator` object, you can access in sequence the elements of a collection. Collection objects are essential components of property lists and, like all objects, can be archived and distributed.
- **Operating-system services.** Many Foundation classes facilitate access of various lower-level services of the operating system and, at the same time, insulate you from operating-system idiosyncrasies. For example, `NSProcessInfo` lets you query the environment in which an application runs and `NSHost` yields the names and addresses of host systems on a network. You can use an `NSTimer` object to send a message to another object at specific intervals, and `NSRunLoop` lets you manage the input sources of an application or other type of program. `NSUserDefaults` provides a programmatic interface to a system database of global (per-host) and per-user default values (preferences).
 - **File system and URL.** `NSFileManager` provides a consistent interface for file operations such as creating, renaming, deleting, and moving files. `NSFileHandle` permits file operations at a lower level (for example, seeking within a file). `NSBundle` finds resources stored in bundles and can dynamically load some of them (for example, nib files and code). You use `NSURL` and related `NSURL` . . . classes to represent, access, and manage URL sources of data.
 - **Multithreading, operations, and subtasks.** `NSThread` lets you create multithreaded programs, and various lock classes offer mechanisms for controlling access to process resources by competing threads. You can use `NSOperation` and `NSOperationQueue` to perform multiple operations (concurrent or non-concurrent) in priority and dependence order. With `NSTask`, your program can fork off a child process to perform work and monitor its progress.
 - **Interprocess communication.** Most of the classes in this category represent various kinds of system ports, sockets, and name servers and are useful in implementing low-level IPC. `NSPipe` represents a BSD pipe, a unidirectional communications channel between processes.

iPhone OS Note: The name server classes are not present in the iPhone OS version of Foundation.

- **Networking.** The `NSNetService` and `NSNetServiceBrowser` classes support the zero-configuration networking architecture called Bonjour. Bonjour is a powerful system for publishing and browsing for services on an IP network.
- **Notifications.** See the summary of the notification classes in [“Foundation Paradigms and Policies”](#) (page 35).
- **Archiving and serialization.** The classes in this category make object distribution and persistence possible. `NSCoder` and its subclasses, along with the `NSCoding` protocol, represent the data an object contains in an architecture-independent way by allowing class information to be stored along with the data. `NSKeyedArchiver` and `NSKeyedUnarchiver` offer methods for encoding objects and scalar values and decoding them in a way that is not dependent on the ordering of encoding messages.
- **Objective-C language services.** `NSException` and `NSAssertionHandler` provide an object-oriented way of making assertions and handling exceptions in code. An `NSInvocation` object is a static representation of an Objective-C message that your program can store and later use to invoke a

message in another object; it is used by the undo manager (`NSUndoManager`) and by the Distributed Objects system. An `NSMethodSignature` object records the type information of a method and is used in message forwarding. `NSClassDescription` is an abstract class for defining and querying the relationships and properties of a class.

The Foundation framework for iPhone OS has a subset of the classes for Mac OS X. The following categories of classes are present only in the Mac OS X version of Foundation:

- **XML processing.** The `NSXML` classes of Foundation process XML data efficiently, either in an event-driven way or by representing an XML document as a hierarchical tree structure. The latter approach lets you to query this structure and manipulate its nodes. The `NSXML` classes support several XML-related technologies and standards, such as XQuery, XPath, XInclude, XSLT, DTD, and XHTML.
- **Expressions and predicates.** The predicate classes—`NSPredicate`, `NSCompoundPredicate`, and `NSComparisonPredicate`—encapsulate the logical conditions to constrain a fetch or filter object. `NSEvaluation` objects represent expressions in a predicate.
- **Spotlight queries.** The `NSMetadataItem`, `NSMetadataQuery` and related query classes encapsulate file-system metadata and make it possible to query that metadata.
- **Scripting.** The classes in this category help to make your program responsive to AppleScript scripts and Apple event commands.
- **Distributed objects.** You use the distributed object classes for communication between processes on the same computer or on different computers on a network. Two of these classes, `NSDistantObject` and `NSProtocolChecker`, have a root class (`NSProxy`) different from the root class of the rest of Cocoa.

Application Kit (Mac OS X)

The Application Kit is a framework containing all the objects you need to implement your graphical, event-driven user interface: windows, dialogs, buttons, menus, scrollers, text fields—the list goes on. The Application Kit handles all the details for you as it efficiently draws on the screen, communicates with hardware devices and screen buffers, clears areas of the screen before drawing, and clips views. The number of classes in the Application Kit may seem daunting at first. However, most Application Kit classes are support classes that you use indirectly. You also have the choice at which level you use the Application Kit:

- Use Interface Builder to create connections from user-interface objects to your application's controller objects, which manage the user interface and coordinate the flow of data between the user interface and internal data structures. For this, you might use off-the-shelf controller objects (for Cocoa bindings) or you may need to implement one or more custom controller classes—particularly the action and delegate methods of those classes. For example, you would need to implement a method that is invoked when the user chooses a menu item (unless it has a default implementation that is acceptable).
- Control the user interface programmatically, which requires more familiarity with Application Kit classes and protocols. For example, allowing the user to drag an icon from one window to another requires some programming and familiarity with the `NSDragging...` protocols.
- Implement your own objects by subclassing `NSView` or other classes. When subclassing `NSView`, you write your own drawing methods using graphics functions. Subclassing requires a deeper understanding of how the Application Kit works.

The Application Kit consists of more than 125 classes and protocols. All classes ultimately inherit from the Foundation framework's `NSObject` class. The diagrams in Figure 1-10 show the inheritance relationships of the Application Kit classes.

```

graph LR
    NSObject* --- NSController
    NSObject* --- NSAlert
    NSObject* --- NSCell
    NSObject* --- NSCollectionViewItem
    NSObject* --- NSDockTile
    NSObject* --- NSEvent
    NSObject* --- NSMenu
    NSObject* --- NSMenuItem
    NSObject* --- NSPredicateEditorRowTemplate
    NSObject* --- NSResponder
    NSObject* --- NSRulerMarker
    NSObject* --- NSStatusBar
    NSObject* --- NSStatusItem
    NSObject* --- NSTabViewItem
    NSObject* --- NSTableColumn
    NSObject* --- NSToolbar
    NSObject* --- NSToolBarItem
    NSObject* --- NSTrackingArea

    NSObject* --- NSApplication
    NSObject* --- NSDrawer
    NSObject* --- NSWindow
    NSObject* --- NSWindowController
    NSObject* --- NSView
    NSObject* --- NSViewController
    NSObject* --- NSToolbarItemGroup

    NSObject* --- NSObjectController
    NSObject* --- NSTreeNode
    NSObject* --- NSUserDefaultsController

    NSObject* --- NSArrayController
    NSObject* --- NSTreeController
    NSObject* --- NSDictionaryController

    NSObject* --- NSButtonCell
    NSObject* --- NSFormCell
    NSObject* --- NSPathCell
    NSObject* --- NSSegmentedCell
    NSObject* --- NSSliderCell
    NSObject* --- NSStepperCell
    NSObject* --- NSTextFieldCell

    NSObject* --- NSMenuItemCell
    NSObject* --- NSPopUpButtonCell

    NSObject* --- NSComboBoxCell
    NSObject* --- NSPathComponentCell
    NSObject* --- NSSearchFieldCell
    NSObject* --- NSSecureTextFieldCell
    NSObject* --- NSTableHeaderCell

    NSObject* --- NSPanel
    NSObject* --- NSColorPanel
    NSObject* --- NSFontPanel
    NSObject* --- NSSavePanel
    NSObject* --- NSOpenPanel

    NSObject* --- NSBox
    NSObject* --- NSClipView
    NSObject* --- NSCollectionView
    NSObject* --- NSControl
    NSObject* --- NSMenuView
    NSObject* --- NSMovieView
    NSObject* --- NSOpenGLView
    NSObject* --- NSProgressIndicator
    NSObject* --- NSQuickDrawView
    NSObject* --- NSRulerView
    NSObject* --- NSScrollView
    NSObject* --- NSSplitView
    NSObject* --- NSTabView
    NSObject* --- NSTableHeaderView

    NSObject* --- NSBrowser
    NSObject* --- NSButton
    NSObject* --- NSColorWell
    NSObject* --- NSImageView
    NSObject* --- NSMatrix
    NSObject* --- NSPathControl
    NSObject* --- NSRuleEditor
    NSObject* --- NSScroller
    NSObject* --- NSSegmentedControl
    NSObject* --- NSSlider
    NSObject* --- NSStepper
    NSObject* --- NSTableView
    NSObject* --- NSTextField

    NSObject* --- NSText
    NSObject* --- NSTextView

    NSObject* --- NSGlyphInfo
    NSObject* --- NSGlyphGenerator
    NSObject* --- NSLayoutManager
    NSObject* --- NSParagraphStyle
    NSObject* --- NSTextAttachment
    NSObject* --- NSTextBlock
    NSObject* --- NSTextContainer
    NSObject* --- NSTextList
    NSObject* --- NSAttributedString*
    NSObject* --- NSMutableAttributedString*
    NSObject* --- NSTextStorage
    NSObject* --- NSTextTab
    NSObject* --- NSTypesetter
    NSObject* --- NSATSTypesetter

    NSObject* --- NSFont
    NSObject* --- NSFontManager
    NSObject* --- NSFontDescriptor

    subgraph User_Interface [User Interface]
        NSAlert
        NSCell
        NSCollectionViewItem
        NSDockTile
        NSEvent
        NSMenu
        NSMenuItem
        NSPredicateEditorRowTemplate
        NSResponder
        NSRulerMarker
        NSStatusBar
        NSStatusItem
        NSTabViewItem
        NSTableColumn
        NSToolbar
        NSToolBarItem
        NSTrackingArea
    end

    subgraph Cocoa_Bindings [Cocoa Bindings]
        NSObjectController
        NSTreeNode
        NSUserDefaultsController
        NSArrayController
        NSTreeController
        NSDictionaryController
    end

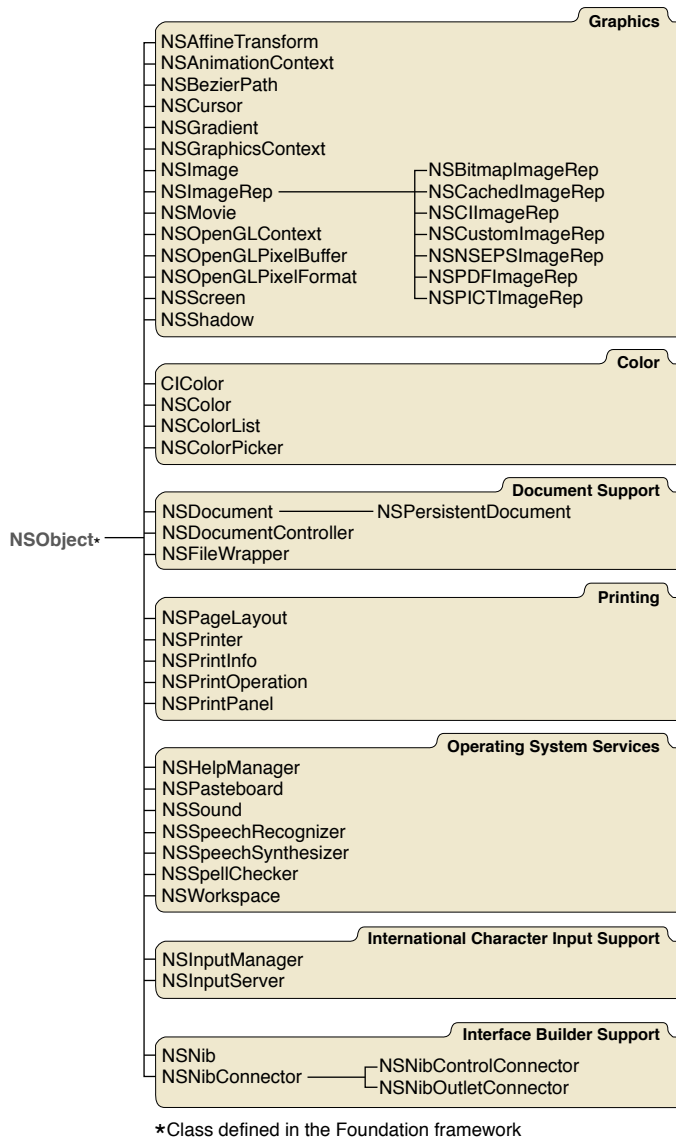
    subgraph Text [Text]
        NSText
        NSTextView
    end

    subgraph Fonts [Fonts]
        NSFont
        NSFontManager
        NSFontDescriptor
    end
  
```

The diagram illustrates the hierarchy of Cocoa framework classes. At the top is the **User Interface** category, which includes classes like `NSController`, `NSAlert`, `NSCell`, `NSCollectionViewItem`, `NSDockTile`, `NSEvent`, `NSMenu`, `NSMenuItem`, `NSPredicateEditorRowTemplate`, `NSResponder`, `NSRulerMarker`, `NSStatusBar`, `NSStatusItem`, `NSTabViewItem`, `NSTableColumn`, `NSToolbar`, `NSToolBarItem`, and `NSTrackingArea`. Below this is the **Cocoa Bindings** category, which includes `NSObjectController`, `NSTreeNode`, `NSUserDefaultsController`, `NSArrayController`, `NSTreeController`, and `NSDictionaryController`. The **Text** category includes `NSText` and `NSTextView`. The **Fonts** category includes `NSFont`, `NSFontManager`, and `NSFontDescriptor`. The **NSObject*** category includes `NSApplication`, `NSDrawer`, `NSWindow`, `NSWindowController`, `NSView`, `NSViewController`, `NSToolbarItemGroup`, `NSPanel`, `NSColorPanel`, `NSFontPanel`, `NSSavePanel`, `NSOpenPanel`, `NSBox`, `NSClipView`, `NSCollectionView`, `NSControl`, `NSMenuView`, `NSMovieView`, `NSOpenGLView`, `NSProgressIndicator`, `NSQuickDrawView`, `NSRulerView`, `NSScrollView`, `NSSplitView`, `NSTabView`, `NSTableHeaderView`, `NSBrowser`, `NSButton`, `NSColorWell`, `NSImageView`, `NSMatrix`, `NSForm`, `NSPathControl`, `NSRuleEditor`, `NSPredicateEditor`, `NSScroller`, `NSSegmentedControl`, `NSSlider`, `NSStepper`, `NSTableView`, `NSTextField`, `NSGlyphInfo`, `NSGlyphGenerator`, `NSLayoutManager`, `NSParagraphStyle`, `NSTextAttachment`, `NSTextBlock`, `NSTextContainer`, `NSTextList`, `NSAttributedString*`, `NSMutableAttributedString*`, `NSTextStorage`, `NSTextTab`, and `NSTypesetter`.

*Class defined in the Foundation framework

Objective-C Application Kit Continued



As you can see, the hierarchy tree of the Application Kit is broad but fairly shallow; the classes deepest in the hierarchy are a mere five superclasses away from the root class and most classes are much closer than that. Some of the major branches in this hierarchy tree are particularly interesting.

At the root of the largest branch in the Application Kit is the `NSResponder` class. This class defines the responder chain, an ordered list of objects that respond to user events. When the user clicks the mouse button or presses a key, an event is generated and passed up the responder chain in search of an object that can respond to it. Any object that handles events must inherit from the `NSResponder` class. The core Application Kit classes—`NSApplication`, `NSWindow`, and `NSView`—inherit from `NSResponder`. You can find out more about these responder classes by reading [“The Core Application Architecture on Mac OS X”](#) (page 201).

The second largest branch of classes in the Application Kit descend from `NSCell`. The noteworthy thing about this group of classes is that they roughly mirror the classes that inherit from `NSControl`, which inherits from `NSView`. For its user-interface objects that respond to user actions, the Application Kit uses an architecture that divides the labor between control objects and cell objects. The `NSControl` and `NSCell` classes, and their subclasses, define a common set of user-interface objects such as buttons, sliders, and browsers that the user can manipulate graphically to control some aspect of your application. Most control objects are associated with one or more cell objects that implement the details of drawing and handling events. For example, a button comprises both an `NSButton` object and an `NSButtonCell` object. See [“Control and Cell Architecture”](#) (page 230) for further information.

Controls and cells implement a mechanism that is based on an important design pattern of the Application Kit: the target-action mechanism. A cell can hold information that identifies the message that should be sent to a particular object when the user clicks (or otherwise acts upon) the cell. When a user manipulates a control (by, for example, clicking the mouse pointer over it), the control extracts the required information from its cell and sends an action message to the target object. Target-action allows you to give meaning to a user action by specifying what the target object and invoked method should be. You typically use Interface Builder to set these targets and actions by Control-dragging from the control object to your application or other object. You can also set targets and actions programmatically.

Another important design-pattern based mechanism of the Application Kit is delegation. Many objects in a user interface, such as text fields and table views, define a delegate. A delegate is an object that acts on behalf of, or in coordination with, the delegating object. It is thus able to impart application-specific logic to the operation of the user interface. For more on delegation, target-action, and other paradigms and mechanisms of the Application Kit, see [“Communicating With Objects”](#) (page 177). For a discussion of the design patterns on which these paradigms and mechanisms are based, see [“Cocoa Design Patterns”](#) (page 141).

One of the general features of Mac OS X version 10.5 and later systems is resolution independence: the resolution of the screen is decoupled from the drawing done by code. The system automatically scales content for rendering on the screen. The Application Kit classes support resolution independence in its user-interface objects. However, for your own applications to take advantage of resolution independence, you might have supply images at a higher resolution or make minor adjustments in your drawing code that take the current scaling factor into account.

Note: In Mac OS X version 10.5 resolution independence is a developer feature, and is not yet available to end users.

The following sections briefly describe some of the capabilities and architectural aspects of the Application Kit and its classes and protocols. It groups classes according to the class hierarchy diagrams shown in [Figure 1-10](#) (page 41) and [“Application Kit class hierarchy—Objective-C \(part two\)”](#) (page 42).

General User-Interface Classes

For the overall functioning of a user interface, the Application Kit provides the following classes:

- **The global application object.** Every application uses a singleton instance of `NSApplication` to control the main event loop, keep track of the application’s windows and menus, distribute events to the appropriate objects (that is, itself or one of its windows), set up top-level autorelease pools, and receive notification of application-level events. An `NSApplication` object has a delegate (an object that you assign) that is notified when the application starts or terminates, is hidden or

activated, should open a file selected by the user, and so forth. By setting the `NSApplication` object's delegate and implementing the delegate methods, you customize the behavior of your application without having to subclass `NSApplication`. [“The Core Application Architecture on Mac OS X”](#) (page 201) discusses this singleton application object in detail.

- **Windows and views.** The window and view classes, `NSWindow` and `NSView`, also inherit from `NSResponder`, and so are designed to respond to user actions. An `NSApplication` object maintains a list of `NSWindow` objects—one for each window belonging to the application—and each `NSWindow` object maintains a hierarchy of `NSView` objects. The view hierarchy is used for drawing and handling events within a window. An `NSWindow` object handles window-level events, distributes other events to its views, and provides a drawing area for its views. An `NSWindow` object also has a delegate allowing you to customize its behavior.

Beginning with Mac OS X version 10.5, the window and view classes of the Application Kit support enhanced animation features.

`NSView` is the superclass for all objects displayed in a window. All subclasses implement a drawing method using graphics functions; `drawRect:` is the primary method you override when creating a new `NSView`.

[“The Core Application Architecture on Mac OS X”](#) (page 201) describes `NSView` and `NSWindow` objects.

- **Controller classes for Cocoa bindings.** The abstract `NSController` class and its concrete subclasses `NSObjectController`, `NSArrayController`, `NSDictionaryController`, and `NSTreeController` are part of the implementation of Cocoa bindings. This technology automatically synchronizes the application data stored in objects and the presentation of that data in a user interface. See [“The Model-View-Controller Design Pattern”](#) (page 161) for a description of these types of controller objects.
- **Panels (dialogs).** The `NSPanel` class is a subclass of `NSWindow` that you use to display transient, global, or pressing information. For example, you would use an instance of `NSPanel`, rather than an instance of `NSWindow`, to display error messages or to query the user for a response to remarkable or unusual circumstances. The Application Kit implements some common dialogs for you such as the Save, Open and Print dialogs, used to save, open, and print documents. Using these dialogs gives the user a consistent look and feel across applications for common operations.
- **Menus and cursors.** The `NSMenu`, `NSMenuItem`, and `NSCursor` classes define the look and behavior of the menus and cursors that your application displays to the user.
- **Grouping and scrolling views.** The `NSBox`, `NSScrollView`, and `NSSplitView` classes provide graphic “accessories” to other view objects or collections of views in windows. With the `NSBox` class, you can group elements in windows and draw a border around the entire group. The `NSSplitView` class lets you append views vertically or horizontally, apportioning to each view some amount of a common territory; a sliding control bar lets the user redistribute the territory among views. The `NSScrollView` class and its helper class, `NSClipView`, provide a scrolling mechanism as well as the graphic objects that let the user initiate and control a scroll. The `NSRulerView` class allows you to add a ruler and markers to a scroll view.
- **Table views and outline views.** The `NSTableView` class displays data in rows and columns. `NSTableView` is ideal for, but not limited to, displaying database records, where rows correspond to each record and columns contain record attributes. The user can edit individual cells and rearrange the columns. You control the behavior and content of an `NSTableView` object by setting its delegate and data source objects. Outline views (instances of `NSOutlineView`, a subclass of `NSTableView`) offer another approach to displaying tabular data. With the `NSBrowser` class you can create an object with which users can display and navigate hierarchical data.

Text and Fonts

The Cocoa text system is based on the Core Text framework, which was introduced in Mac OS X version 10.5. The Core Text framework provides a modern, low-level, high-performance technology for laying out text. If you use the Cocoa text system, you should rarely have reason to use Core Text directly.

The `NSTextField` class implements a simple editable text-input field, and the `NSTextView` class provides more comprehensive editing features for larger text bodies.

`NSTextView`, a subclass of the abstract `NSText` class, defines the interface to the extended text system. `NSTextView` supports rich text, attachments (graphics, file, and other), input management and key binding, and marked text attributes. `NSTextView` works with the Fonts window and Font menu, rulers and paragraph styles, the Services facility, and the pasteboard (Clipboard). `NSTextView` also allows customizing through delegation and notifications—you rarely need to subclass `NSTextView`. You rarely create instances of `NSTextView` programmatically either, since objects on Interface Builder’s palettes, such as `NSTextField`, `NSForm`, and `NSScrollView`, already contain `NSTextView` objects.

It is also possible to do more powerful and more creative text manipulation (such as displaying text in a circle) using `NSTextStorage`, `NSLayoutManager`, `NSTextContainer`, and related classes. The Cocoa text system also supports lists, tables, and non-contiguous selections.

The `NSFont` and `NSFontManager` classes encapsulate and manage font families, sizes, and variations. The `NSFont` class defines a single object for each distinct font; for efficiency, these objects, which can represent a lot of data, are shared by all the objects in your application. The `NSFontPanel` class defines the Fonts window that’s presented to the user.

Graphics and Colors

The classes `NSImage` and `NSImageRep` encapsulate graphics data, allowing you to easily and efficiently access images stored in files on the disk and displayed on the screen. `NSImageRep` subclasses each know how to draw an image from a particular kind of source data. The `NSImage` class provides multiple representations of the same image, and also provides behaviors such as caching. The imaging and drawing capabilities of Cocoa are integrated with the Core Image framework.

Color is supported by the classes `NSColor`, `NSColorSpace`, `NSColorPanel`, `NSColorList`, `NSColorPicker`, and `NSColorWell`. `NSColor` and `NSColorSpace` support a rich set of color formats and representations, including custom ones. The other classes are mostly interface classes: They define and present panels and views that allow the user to select and apply colors. For example, the user can drag colors from the Color window to any color well.

The `NSGraphicsContext`, `NSBezierPath`, and `NSAffineTransform` classes help you with vector drawing and support graphical transformations such as scaling, rotation, and translation.

Printing and Faxing

The `NSPrinter`, `NSPrintPanel`, `NSPageLayout`, and `NSPrintInfo` classes work together to provide the means for printing and faxing the information that your application displays in its windows and views. You can also create a PDF representation of an `NSView`.

Document and File-System Support

Use the `NSFileWrapper` class to create objects that correspond to files or directories on disk. `NSFileWrapper` holds the contents of the file in memory so that it can be displayed, changed, or transmitted to another application. It also provides an icon for dragging the file or representing it as an attachment. Or use the `NSFileManager` class in the Foundation framework to access and enumerate file and directory contents. The `NSOpenPanel` and `NSSavePanel` classes also provide a convenient and familiar user interface to the file system.

The `NSDocumentController`, `NSDocument`, and `NSWindowController` classes define an architecture for creating document-based applications. (The `NSWindowController` class is shown in the User Interface group of classes in the class hierarchy charts). Such applications can generate identically contained but uniquely composed sets of data that can be stored in files. They have built-in or easily acquired capabilities for saving, opening, reverting, closing, and managing these documents.

Internationalization and Character Input Support

If an application is to be used in more than one part of the world, its resources may need to be customized, or localized, for language, country, or cultural region. For example, an application may need to have separate Japanese, English, French, and German versions of character strings, icons, nib files, or context help. Resource files specific to a particular language are grouped together in a subdirectory of the bundle directory (the directories with the `.lproj` extension). Usually you set up localization resource files using Interface Builder. See [“Nib Files and Other Application Resources”](#) (page 235) for more information on the Cocoa internationalization facilities.

The `NSInputServer` and `NSInputManager` classes, along with the `NSTextInput` protocol, give your application access to the text input management system. This system interprets keystrokes generated by various international keyboards and delivers the appropriate text characters or Control-key events to text view objects. (Typically the text classes deal with these classes and you won't have to.)

Operating-System Services

The Application Kit provides operating-system support for your application through classes that implement the following features:

- **Sharing data with other applications.** The `NSPasteboard` class defines the pasteboard, a repository for data that's copied from your application, making this data available to any application that cares to use it. `NSPasteboard` implements the familiar cut-copy-paste operation.
- **Dragging.** With very little programming on your part, custom view objects can be dragged and dropped anywhere. Objects become part of this dragging mechanism by conforming to `NSDragging...` protocols; draggable objects conform to the `NSDraggingSource` protocol, and destination objects (receivers of a drop) conform to the `NSDraggingDestination` protocol. The Application Kit hides all the details of tracking the cursor and displaying the dragged image.
- **Spell checking.** The `NSSpeller` class lets you define a spell-checking service and provide it as a service to other applications. To connect your application to a spell-checking service, you use the `NSSpellerChecker` class. The `NSIgnoreMisspelledWords` and `NSChangeSpelling` protocols support the spell-checking mechanism.

Interface Builder Support

The abstract `NSNibConnector` class and its two concrete subclasses, `NSNibControlConnector` and `NSNibOutletConnector`, represent connections in Interface Builder. `NSNibControlConnector` manages an action connection in Interface Builder and `NSNibOutletConnector` manages an outlet connection.

UIKit (iPhone OS)

The UIKit framework on iPhone OS is the sister framework of the Application Kit on Mac OS X. Its purpose is essentially the same: to provide all the classes that an application needs to construct and manage its user interface. However, there are significant differences in how the frameworks realize this purpose.

One of the greatest differences is that, on iPhone OS, the objects that appear in the user interface of a Cocoa application look and behave differently from their counterparts in a Cocoa application running on Mac OS X. Some common examples are text views, table views, and buttons. In addition, the event-handling and drawing models for Cocoa applications on the two platforms are significantly different. The following sections explain the reasons for these and other differences.

You can add UIKit objects to your application's user interface in three ways:

- Use the Interface Builder development application to drag-and-drop windows, views, and other objects from an object library.
- Create, position, and configure framework objects programmatically.
- Implement custom user-interface objects by subclassing `UIView` or classes that inherit from `UIView`.

Further Reading: To gain a thorough understanding of UIKit, read *UIKit Framework Reference* and *iPhone OS Programming Guide*.

Overview of UIKit Classes

As with the Application Kit, the classes of the UIKit framework ultimately inherit from `NSObject`. Figure 1-11 presents the classes of the UIKit framework in their inheritance relationship.

[illegible]

The Cocoa Frameworks
2008-07-08 | © 2008 Apple Inc. All Rights Reserved.

scrolls a table view with his or her finger or types characters in a virtual keyboard, UIKit generates an event and that event is passed up the responder chain until an object handles it. The corresponding core objects—application (`UIApplication`), window (`UIWindow`), and view (`UIView`)—all directly or indirectly inherit from `UIResponder`.

Unlike the Application Kit, UIKit does not make use of cells. Controls in UIKit—that is, all objects that inherit from `UIControl`—do not require cells to carry out their primary role: sending action messages to a target object. Yet the way UIKit implements the target-action mechanism is different from the design in the Application Kit. The `UIControl` class defines a set of event types for controls; if, for example, you want a button (`UIButton`) to send an action message to a target object, you call `UIControl` methods to associate the action and target with one or more of the control event types. When one of those events happens, the control sends the action message.

The UIKit framework makes considerable use of delegation, another design pattern of the Application Kit. Yet the UIKit implementation of delegation is different. Instead of using informal protocols, UIKit uses formal protocols with possibly some protocol methods marked optional.

Note: For a complete description of the target-action mechanism in UIKit and the Application Kit, see [“The Target-Action Mechanism”](#) (page 184). To learn more about delegation and protocols, both formal and informal, see [“Delegates and Data Sources”](#) (page 179) and [“Protocols”](#) (page 61).

Application Coordination

Each application running on iPhone OS is managed by a singleton application object, and this object has a job that is almost identical to that for the global `NSApplication` object. A `UIApplication` object controls the main event loop, keeps track of the applications windows and views, and dispatches incoming events to the appropriate responder objects.

The `UIApplication` object also receives notification of system-level and application-level events. Many of these it passes to its delegate, allowing it to inject application-specific behavior when the application launches and terminates, and to respond to low-memory warnings and changes in time.

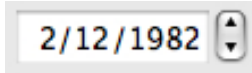
Differences in Event and Drawing Models

In Mac OS X, the mouse and keyboard generate most user events. The Application Kit uses `NSEvent` objects to encapsulate these events. On iPhone OS, however, a user’s finger movements on the screen are what originate events. UIKit also has a class, `UIEvent`, to represent these events. But finger touches are different in nature from mouse clicks; two or more touches occurring over a period could comprise a discrete event—a pinch gesture, for example. Thus a `UIEvent` object contains one or more objects representing finger touches (`UITouch`). The model for distributing and dispatching events to objects that can handle them on the two platforms is almost identical. However, to handle an event, an object must take into consideration the sequence of touches specific to that event.

The drawing models are very similar for the Application Kit and UIKit. In UIKit, however, a `UIView` object has an animation layer associated with it to enable the animation of changes within views and transitions between views. That said, the programmatic support that UIKit directly provides for drawing is limited (specifically, simple line and rectangle drawing through functions declared in `UIGraphics.h`). Therefore, for drawing of any sophistication, an application must use the Core Graphics or OpenGL framework. You use `UIColor` objects to set colors in the current graphics context.

General User-Interface Classes

Objects on an iPhone OS user interface are visibly different than objects on a Mac OS X user interface. Because of the nature of the device—specifically the smaller screen size and the use of fingers instead of the mouse and keyboard for input—user-interface objects on iPhone OS must typically be larger (to be an adequate target for touches) while at the same time make as efficient use of screen real estate as possible. These objects are sometimes based on visual and tactile analogues of an entirely different sort. As an example, consider the date picker object, which is instantiated from a class that both the UIKit and Application Kit frameworks define. On Mac OS X, the date picker looks like the following:



This style of date picker has a two tiny areas for incrementing date components, and thus is suited to manipulation by a mouse pointer. Contrast this with the date picker seen on iPhone applications:



This style of date picker is more suited for finger touches as an input source; users can swipe a month, day, or year column to spin it to a new value.

As with the Application Kit, many of UIKit classes fall into functional groups:

- **Controls.** The subclasses of `UIControl` instantiate objects that let users communicate their intent to an application. In addition to the standard button object (`UIButton`) and slider object (`UISlider`), there is a control that simulate off/on switches (`UISwitch`), a spinning-wheel control for selecting from multidimensional sets of values (`UIPickerView`), a control for paging through documents (`UIPageControl`), and other controls.
- **Modal views.** The two classes inheriting from `UIModalView` are for displaying messages to users either in the form of “sheets” attached to specific views or windows (`UIActionSheet`) or as unattached alert dialogs (`UIAlertView`).
- **Scroll views.** The `UIScrollView` class enables instances of its subclasses to respond to touches for scrolling within large views. As users scroll, the scroll view displays transient indicators of position within the document. The subclasses of `UIScrollView` implement table views, text views, and web views.
- **Toolbars, navigation bars, and view controllers.** The `UIViewController` class is a base class for managing a view. A view controller provides methods for creating and observing views, overlaying views, handling view rotation, and responding to low-memory warnings. UIKit includes concrete subclasses of `UIViewController` for managing toolbars, navigation bars, and image pickers.

Applications use both toolbars and navigation bars to manage behavior related to the “main” view on the screen; typically, toolbars are placed beneath the main view and navigation bars above it. You use toolbar (`UIToolbar`) object to switch between modes or views of an application; you can also used them to display a set of functions that perform some action related to the current main view. You use navigation bars (`UINavigationController`) to manage sequences of windows or

views in an application and, in effect, to “drill down” a hierarchy of objects defined by the application; the Mail application for example, uses a navigation bar to navigate from accounts to mailbox folders and from there to individual email messages.

Text and Images

Users can enter text in an iPhone application either through a text view (`UITextView`) or a text field (`UITextField`). These classes adopt the `UInputTraits` protocol to specify the appearance and behavior of the virtual keyboard that is presented when users touch the text-entry object; any subclasses that enable entry of text should also conform to this protocol. Applications can draw text in views using `UStringDrawing` methods, a category on the `NSString` class. And with the `UIFont` class you can specify the font characteristics of text in all objects that display text, including table cells, navigation bars, and labels.

UIKit uses `UIImage` objects to represent and encapsulate images.

Comparing Application Kit and UIKit Classes

The Application Kit and UIKit are Cocoa application frameworks that are designed for different platforms, one for Mac OS X and the other for iPhone OS. Because of this affinity, it is not surprising that many of the classes in each framework have similar names; in most cases, the prefix (“NS” versus “UI”) is the only name difference. These similarly named classes fulfill mostly similar roles, but there are differences. These differences can be a matter of scope, of inheritance, or of design. Generally, UIKit classes have fewer methods than their Application Kit counterparts, largely because an iPhone application is running in a more restricted environment.

Figure 1-12 shows UIKit classes with corresponding classes in the Application Kit. [Table 1-1](#) (page 53) describes the differences between the major classes in each framework.

Figure 1-12 UIKit classes and their Application Kit counterparts

Table 1-1 Major classes of the Application Kit and UIKit

Classes	Comparison
NSApplication UIApplication	The classes are strikingly similar in their primary roles. They provide a singleton object that sets up the application's display environment and event loop, distributes events, and notifies a delegate when application-specific events occur (such as launch and termination). However, the <code>NSApplication</code> class performs functions (for example, managing application suspension, reactivation, and hiding) that are not available to an iPhone application.
NSResponder UIResponder	These classes also have nearly identical roles. They are abstract classes that define an interface for responding to events and managing the responder chain. The main difference is that the <code>NSResponder</code> event-handling methods are defined for the mouse and keyboard, while the <code>UIResponder</code> methods are defined for the Multi-Touch event model.
NSWindow UIWindow	The <code>UIWindow</code> class occupies a different place in the class hierarchy than does <code>NSWindow</code> ; it is a subclass of <code>UIView</code> , whereas the Application Kit class inherits directly from <code>NSResponder</code> . <code>UIWindow</code> has a much more restricted role in an application than does <code>NSWindow</code> . It also provides an area for displaying views, dispatches events to those views, and converts between window and view coordinates.
NSView UIView	These classes are very similar in purpose and in their basic sets of methods. They allow you to move and resize views, manage the view hierarchy, draw view content, and convert view coordinates. The design of <code>UIView</code> , however, makes view objects inherently capable of animation.
NSControl UIControl	Both classes define a mechanism for objects such as buttons and sliders so that, when manipulated, the control object sends an action message to a target object. The classes implement the target-action mechanism in different ways, largely because of the difference between event models. See “The Target-Action Mechanism” (page 184) for information.
NSViewController UIViewController	The role of both of these classes is, as their names suggest, to manage views. How they accomplish this task is different. The management provided by an <code>NSViewController</code> object is dependent on bindings, which is a technology supported only on Mac OS X. <code>UIViewController</code> objects are used in the iPhone OS application model for modal and navigation user interfaces (for example, the views controlled by navigation bars).
NSTableView UITableView	<code>NSTableView</code> inherits from <code>NSControl</code> , but <code>UITableView</code> does not inherit from <code>UIControl</code> . More importantly, <code>NSTableView</code> objects support multiple columns of data; <code>UITableView</code> objects display only a single column of data at a time, and thus function more as lists than presentations of tabular data.

Among the minor classes you can find some differences too. For example, UIKit has the `UITextField` and `UILabel` classes, the former for editable text fields and the latter for non-editable text fields used as labels; with the `NSTextField` class you can also create both kinds of objects simply by setting text-field attributes. Similarly, the `NSProgressIndicator` class can create objects in styles that correspond to instances of the `UIProgressIndicator` and `UIProgressBar` classes.

Other Frameworks With Cocoa API

As part of a standard installation, Apple includes, in addition to the core frameworks for both platforms, several frameworks that vend Cocoa programmatic interfaces. You can use these secondary frameworks to give your application capabilities that are desirable, if not essential. Some notable secondary frameworks include:

- **Core Data**—(Mac OS X only) The Core Data framework helps a program to manage graphs of model objects through their life cycles, including the persistent storage of their data in relational databases or flat files. It includes features such as undo and redo management, automatic validation of values, change propagation, and integration with Cocoa bindings. See [“Other Cocoa Architectures on Mac OS X”](#) (page 241) and *Core Data Programming Guide* for more information.
- **Sync Services**—(Mac OS X only) Using Sync Services you can sync existing contacts, calendars and bookmarks schemas as well as your own application data. You can also extend existing schemas. See *Sync Services Programming Guide* for more information.
- **Address Book**—This framework implements a centralized database for contact and other personal information. Applications that use the Address Book framework can share this contact information with other applications, including Apple’s Mail and iChat. See *Address Book Programming Guide* for more information.

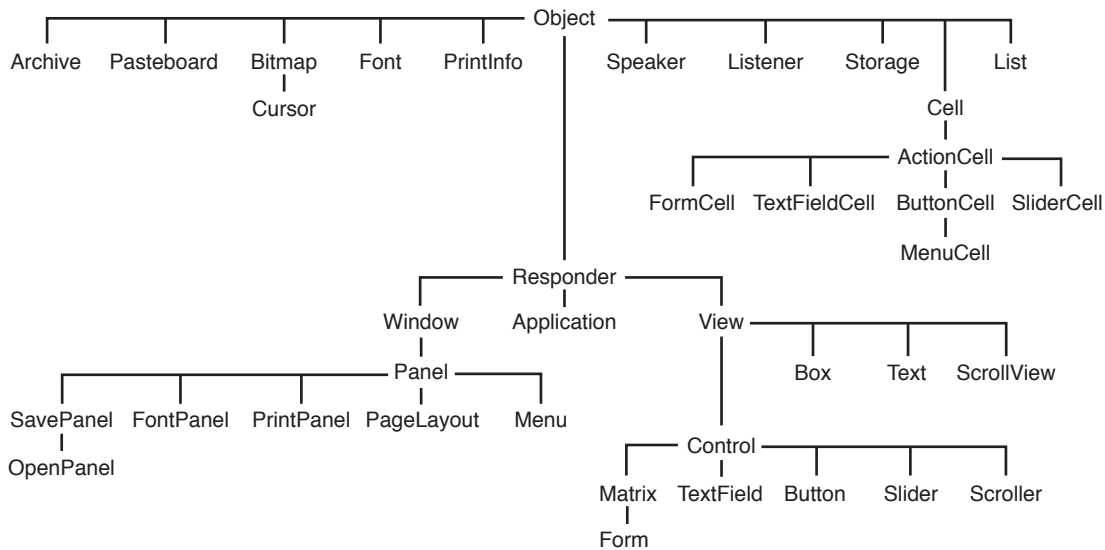
iPhone OS Note: There are different versions of this framework on Mac OS X and iPhone OS. In addition, the Address Book framework on iPhone OS has only ANSI C (procedural) programmatic interfaces.

- **Preference Panes**—(Mac OS X only) With this framework you can create plug-ins that your application can dynamically load to obtain a user interface for recording user preferences, either for the application itself or system-wide. See *Preference Panes* for more information.
- **Screen Saver**—(Mac OS X only) The Screen Saver framework helps you create Screen Effects modules, which can be loaded and run via System Preferences. See *Screen Saver Framework Reference* for more information.
- **Web Kit**—(not public on iPhone OS) The Web Kit framework provides a set of core classes to display web content in windows, and by default, implements features such as following links clicked by the user. See *Web Kit Objective-C Programming Guide* for more information.

A Bit of History

Many years ago Cocoa was known as NeXTSTEP. NeXT Computer developed and released version 1.0 of NeXTSTEP in September of 1989, and versions 2.0 and 3.0 followed not far behind (in 1990 and 1992, respectively). In this early phase, NeXTSTEP was more than an application environment; the term referred to the entire operating system, including the windowing and imaging system (which was based on Display PostScript), the Mach kernel, device drivers, and so on.

Back then, there was no Foundation framework. Indeed, there were no frameworks; instead, the software libraries (dynamically shared) were known as kits, the most prominent of them being the Application Kit. Much of the role that Foundation now occupies was taken by an assortment of functions, structures, constants, and other types. The Application Kit itself had a much smaller set of classes than it does today. Figure 1-13 shows a class hierarchy chart of NeXTSTEP 0.9 (1988).

Figure 1-13 Application Kit class hierarchy in 1988

In addition to the Application Kit, the early NeXTSTEP included the Sound Kit and the Music Kit, libraries containing a rich set of Objective-C classes that provided high-level access to the Display Postscript layer for audio and music synthesis.

In early 1993 NeXTSTEP 3.1 was ported to (and shipped on) Intel, Sparc, and Hewlett-Packard computers. NeXTSTEP 3.3 also marked a major new direction, for it included a preliminary version of Foundation. Around this time (1993), the OpenStep initiative also took form. OpenStep was a collaboration between Sun and NeXT to port the higher levels of NeXTSTEP (particularly the Application Kit and Display PostScript) to Solaris. The “Open” in the name referred to the open API specification that the companies would publish jointly. The official OpenStep API, published in September of 1994, were the first to split the API between Foundation and Application Kit and the first to use the “NS” prefix.

By June 1996 NeXT had ported and shipped versions of OpenStep 4.0 that could run Intel, Sparc, and Hewlett-Packard computers as well as an OpenStep runtime that could run on Windows systems. Sun also finished their port of OpenStep to Solaris and shipped it as part of their Network Object Computing Environment. OpenStep, however, never became a significant part of Sun’s overall strategy.

When Apple acquired NeXT Software (as it was then called) in 1997, OpenStep became the Yellow Box and was included with Mac OS X Server (also known as Rhapsody) and Windows. Then, with the evolution of the Mac OS X strategy, it was finally renamed to “Cocoa.”

Cocoa Objects

To say that Cocoa is object-oriented is to invite the question: What is a Cocoa object? This section describes what is distinctive about Objective-C objects and what advantages the language brings to software development. It also shows you how to use Objective-C to send messages to objects and how to handle return values from those messages. (Objective-C is an elegantly simple language, so this is not too hard to do.) This section also describes the root class, `NSObject`, and explains how to use its programmatic interface to create objects, introspect them, and manage object life cycles

A Simple Cocoa Command-Line Tool

Let's begin with a simple command-line program created using the Foundation framework for Mac OS X. Given a series of arbitrary words as arguments, the program removes redundant occurrences, sorts the remaining list of words in alphabetical order, and prints the list to standard output. Listing 2-1 shows a typical execution of this program.

Listing 2-1 Output from a simple Cocoa tool

```
localhost> SimpleCocoaTool a z c a l q m z
a
c
l
m
q
z
```

Listing 2-2 shows the code for an Objective-C version of this program.

Listing 2-2 Cocoa code for a uniquing and sorting tool

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

int main (int argc, const char * argv[]) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSArray *args = [[NSProcessInfo processInfo] arguments];
    NSCountedSet *cset = [[NSCountedSet alloc] initWithArray:args];
    NSArray *sorted_args = [[cset allObjects]
        sortedArrayUsingSelector:@selector(compare:)];
    NSEnumerator *enm = [sorted_args objectEnumerator];
    id word;
```

```

while (word = [enm nextObject]) {
    printf("%s\n", [word UTF8String]);
}

[ccset release];
[pool release];
return 0;
}

```

This code creates and uses several objects: an autorelease pool for memory management, collection objects (arrays and a set) for “uniquing” and sorting the specified words, and an enumerator object for iterating through the elements in the final array and printing them to standard output.

The first thing you probably notice about this code is that it is short, perhaps much shorter than a typical ANSI C version of the same program. Although much of this code might look strange to you, many of its elements are familiar ANSI C. These include assignment operators, control-flow statements (`while`), calls to C-library routines (`printf`), primitive scalar types, and so on. Objective-C obviously has ANSI C underpinnings.

The rest of this chapter examines the Objective-C elements of this code, using them as examples in discussions on subjects ranging from the mechanics of message-sending to the techniques of memory management. If you haven’t seen Objective-C code before, the code in the example might seem formidably convoluted and obscure, but that impression will melt away soon. Objective-C is actually a simple, elegant programming language that is easy to learn and intuitive to program with.

Object-Oriented Programming With Objective-C

Cocoa is pervasively object-oriented, from its paradigms and mechanisms to its event-driven architecture. Objective-C, the development language for Cocoa, is thoroughly object-oriented too, despite its grounding in ANSI C. It provides runtime support for message dispatch and specifies syntactical conventions for defining new classes. Objective-C supports most of the abstractions and mechanisms found in other object-oriented languages such as C++ and Java. These include inheritance, encapsulation, reusability, and polymorphism.

But Objective-C is different from these other object-oriented languages, often in important ways. For example, Objective-C, unlike C++, doesn’t allow operator overloading, templates, or multiple inheritance.

Although Objective-C doesn’t have these features, its strengths as an object-oriented programming language more than compensate. What follows is an exploration of the special capabilities of Objective-C.

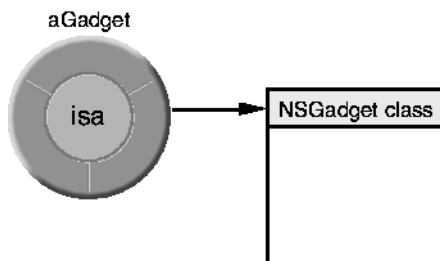
Further Reading: Much of this section summarizes information from the definitive guide to Objective-C, *The Objective-C 2.0 Programming Language*. Consult this document for a detailed and comprehensive description of Objective-C.

The Objective-C Advantage

If you're a procedural programmer new to object-oriented concepts, it might help at first to think of an object as essentially a structure with functions associated with it. This notion is not too far off the reality, particularly in terms of runtime implementation.

Every Objective-C object hides a data structure whose first member—or instance variable—is the “isa pointer.” (Most remaining members are defined by the object's class and superclasses.) The isa pointer, as the name suggests, points to the object's class, which is an object in its own right (see Figure 2-1) and is compiled from the class definition. The class object maintains a dispatch table consisting essentially of pointers to the methods it implements; it also holds a pointer to its superclass, which has its own dispatch table and superclass pointer. Through this chain of references, an object has access to the method implementations of its class and all its superclasses (as well as all inherited public and protected instance variables). The isa pointer is critical to the message-dispatch mechanism and to the dynamism of Cocoa objects.

Figure 2-1 An object's isa pointer



This peek behind the object facade gives a highly simplified view of what happens in the Objective-C runtime to enable message-dispatch, inheritance, and other facets of general object behavior. But this information is essential to understanding the major strength of Objective-C, its dynamism.

The Dynamism of Objective-C

Objective-C is a very dynamic language. Its dynamism frees a program from compile-time and link-time constraints and shifts much of the responsibility for symbol resolution to runtime, when the user is in control. Objective-C is more dynamic than other programming languages because its dynamism springs from three sources:

- Dynamic typing—determining the class of an object at runtime
- Dynamic binding—determining the method to invoke at runtime
- Dynamic loading—adding new modules to a program at runtime

For dynamic typing, Objective-C introduces the `id` data type, which can represent any Cocoa object. A typical use of this generic object type is shown in this part of the code example from [Listing 2-2](#) (page 57):

```
id word;
while (word = [enm nextObject]) {
    // etc....
}
```

The `id` data type makes it possible to substitute any type of object at runtime. You can thereby let runtime factors dictate what kind of object is to be used in your code. Dynamic typing permits associations between objects to be determined at runtime rather than forcing them to be encoded in a static design. Static type checking at compile time may ensure stricter data integrity, but in exchange for that integrity, dynamic typing gives your program much greater flexibility. And through object introspection (for example, asking a dynamically typed, anonymous object what its class is) you can still verify the type of an object at runtime and thus validate its suitability for a particular operation. (Of course, you can always statically check the types of objects when you need to.)

Dynamic typing gives substance to dynamic binding, the second kind of dynamism in Objective-C. Just as dynamic typing defers the resolution of an object's class membership until runtime, dynamic binding defers the decision of which method to invoke until runtime. Method invocations are not bound to code during compilation, but only when a message is actually delivered. With both dynamic typing and dynamic binding, you can obtain different results in your code each time you execute it. Runtime factors determine which receiver is chosen and which method is invoked.

The runtime's message-dispatch machinery enables dynamic binding. When you send a message to a dynamically typed object, the runtime system uses the receiver's `isa` pointer to locate the object's class, and from there the method implementation to invoke. The method is dynamically bound to the message. And you don't have to do anything special in your Objective-C code to reap the benefits of dynamic binding. It happens routinely and transparently every time you send a message, especially one to a dynamically typed object.

Dynamic loading, the final type of dynamism, is a feature of Cocoa that depends on Objective-C for runtime support. With dynamic loading, a Cocoa program can load executable code and resources as they're needed instead of having to load all program components at launch time. The executable code (which is linked prior to loading) often contains new classes that become integrated into the runtime image of the program. Both code and localized resources (including nib files) are packaged in bundles and are explicitly loaded with methods defined in Foundation's `NSBundle` class.

This "lazy-loading" of program code and resources improves overall performance by placing lower memory demands on the system. Even more importantly, dynamic loading makes applications extensible. You can devise a plug-in architecture for your application that allows you and other developers to customize it with additional modules that the application can dynamically load months or even years after the application is released. If the design is right, the classes in these modules will not clash with the classes already in place because each class encapsulates its implementation and has its own name space.

Extensions to the Objective-C Language

Objective-C features two extensions to the base language that are powerful tools in software development: categories and protocols. Some extensions introduce different techniques for declaring methods and associating them with a class. Others offer convenient ways to declare and access object properties, enumerate quickly over collections, handle exceptions, and perform other tasks.

Categories

Categories give you a way to add methods to a class without having to make a subclass. The methods in the category become part of the class type (within the scope of your program) and are inherited by all the class's subclasses. There is no difference at runtime between the original methods and the added methods. You can send a message to any instance of the class (or its subclasses) to invoke a method defined in the category.

Categories are more than a convenient way to add behavior to a class. You can also use categories to compartmentalize methods, grouping related methods in different categories. Categories can be particularly handy for organizing large classes; you can even put different categories in different source files if, for instance, there are several developers working on the class.

You declare and implement a category much as you do a subclass. Syntactically, the only difference is the name of the category, which follows the `@interface` or `@implementation` directive and is put in parentheses. For example, say you want to add a method to the `NSArray` class that prints the description of the collection in a more structured way. In the header file for the category, you would write declaration code similar to the following:

```
#import <Foundation/NSArray.h> // if Foundation not already imported

@interface NSArray (PrettyPrintElements)
- (NSString *)prettyPrintDescription;
@end
```

Then in the implementation file you'd write code such as:

```
#import "PrettyPrintCategory.h"

@implementation NSArray (PrettyPrintElements)
- (NSString *)prettyPrintDescription {
    // implementation code here...
}
@end
```

There are some limitations to categories. You cannot use a category to add any new instance variables to the class. Although a category method can override an existing method, it is not recommended that you do so, especially if you want to augment the current behavior. One reason for this caution is that the category method is part of the class's interface, and so there is no way to send a message to `super` to get the behavior already defined by the class. If you need to change what an existing method of a class does, it is better to make a subclass of the class.

You can define categories that add methods to the root class, `NSObject`. Such methods are available to *all* instances and class objects that are linked into your code. Informal protocols—the basis for the Cocoa delegation mechanism—are declared as categories on `NSObject`. This wide exposure, however, has its dangers as well as its uses. The behavior you add to every object through a category on `NSObject` could have consequences that you might not be able to anticipate, leading to crashes, data corruption, or worse.

Protocols

The Objective-C extension called a protocol is very much like an interface in Java. Both are simply a list of method declarations publishing an interface that any class can choose to implement. The methods in the protocol are invoked by messages sent by an instance of some other class.

The main value of protocols is that they, like categories, can be an alternative to subclassing. They yield some of the advantages of multiple inheritance in C++, allowing sharing of interfaces (if not implementations). A protocol is a way for a class to declare an interface while concealing its identity. That interface may expose all or (as is usually the case) only a range of the services the class has to offer. Other classes throughout the class hierarchy, and not necessarily in any inheritance relationship (not even to the root class), can implement the methods of that protocol and so access the published services. With a protocol, even classes that have no knowledge of another's identity (that is, class type) can communicate for the specific purpose established by the protocol.

There are two types of protocols: formal and informal. Informal protocols were briefly introduced in “Categories” (page 61). These are categories on `NSObject`; as a consequence, every object with `NSObject` as its root object (as well as class objects) implicitly adopts the interface published in the category. To use an informal protocol, a class does not have to implement every method in it, just those methods it's interested in. For an informal protocol to work, the class declaring the informal protocol must get a positive response to a `respondsToSelector:` message from a target object before sending that object the protocol message. (If the target object did not implement the method there would be a runtime exception.)

Formal protocols are usually what is designated by “protocol” in Cocoa. They allow a class to formally declare a list of methods that are an interface to a vended service. The Objective-C language and runtime system supports formal protocols; the compiler can check for types based on protocols, and objects can introspect at runtime to verify conformance to a protocol. Formal protocols have their own terminology and syntax. The terminology is different for provider and client:

- A provider (which usually is a class) *declares* the formal protocol.
- A client class *adopts* a formal protocol, and by doing so agrees to implement all required methods of the protocol.
- A class is said to *conform* to a formal protocol if it adopts the protocol or inherits from a class that adopts it. (Protocols are inherited by subclasses.)

Both the declaration and the adoption of a protocol have their own syntactical forms in Objective-C. To declare a protocol you must use the `@protocol` compiler directive. The following example shows the declaration of the `NSCoding` protocol (in the Foundation framework's header file `NSObject.h`).

```
@protocol NSCoding
- (void)encodeWithCoder:(NSCoder *)aCoder;
- (id)initWithCoder:(NSCoder *)aDecoder;
@end
```

Objective-C 2.0 adds a refinement to formal protocols by giving you the option of declaring *optional* protocol methods as well as required ones. In Objective-C 1.0, the adopter of a protocol had to implement all methods of the protocol. In Objective-C 2.0 protocol methods are still implicitly required, and can be specifically marked as such using the `@required` directive. But you can also mark blocks of protocol methods for optional implementation using the `@optional` directive; all methods declared after this directive, unless there is an intervening `@required`, can be optionally implemented. Consider these declarations:

```
@protocol MyProtocol
// implementation this method is required implicitly
- (void)requiredMethod;

@optional
// implementation of these methods is optional
- (void)anOptionalMethod;
```

```
- (void)anotherOptionalMethod;

@required
// implementation of this method is required
- (void)anotherRequiredMethod;
@end
```

The class that declares the protocol methods does not need to implement them, but it should invoke them in objects of the conforming class. Before invoking optional methods, it should verify that they're implemented using the `respondsToSelector:` method.

A class adopts a protocol by specifying the protocol, enclosed by angle brackets, at the end of its `@interface` directive, just after the superclass. A class can adopt multiple protocols by delimiting them with commas. This is how the Foundation `NSData` class adopts three protocols.

```
@interface NSData : NSObject <NSCopying, NSMutableCopying, NSCoding>
```

By adopting these protocols, `NSData` commits itself to implementing all required methods declared in the protocols. It may also choose to implement methods marked with the `@optional` directive. Categories can also adopt protocols, and their adoption becomes part of the definition of their class.

Objective-C types classes by the protocols they conform to as well as the classes they inherit from. You can check if a class conforms to a particular protocol by sending it a `conformsToProtocol:` message:

```
if ([anObject conformsToProtocol:@protocol(NSCoding)]) {
    // do something appropriate
}
```

In a declaration of a type—a method, instance variable, or function—you can specify protocol conformance as part of the type. You thus get another level of type checking by the compiler, one that's more abstract because it's not tied to particular implementations. You use the same syntactical convention as for protocol adoption: Put the protocol name between angle brackets to specify protocol conformance in the type. You often see the dynamic object type, `id`, used in these declarations, for example:

```
- (void)draggingEnded:(id <NSDraggingInfo>)sender;
```

Here the object referred to in the argument can be of any class type, but it must conform to the `NSDraggingInfo` protocol.

Cocoa provides several examples of protocols other than the ones shown or so far. An interesting one is the `NSObject` protocol. Not surprisingly, the `NSObject` class adopts it, but so does the other root class, `NSProxy`. Through the protocol, the `NSProxy` class can interact with the parts of the Objective-C runtime essential to reference counting, introspection, and other basic aspects of object behavior.

Declared Properties

In the object modeling design pattern (see [“Object Modeling”](#) (page 169)) objects have properties. Properties consist of an object's attributes, such as title and color, and an object's relationships with other objects. In traditional Objective-C code, you define properties by declaring instance variables and, to enforce encapsulation, by implementing accessor methods to get and set the values of those variables. This is a tedious and error-prone task, especially when memory management is a concern (see [“Storing and Accessing Properties”](#) (page 126)).

Objective-C 2.0, which was introduced in Mac OS X version 10.5, offers a syntax for declaring properties and specifying how they are to be accessed. Declaring a property becomes a kind of shorthand for declaring a setter and getter method for the property. With properties, you no longer have to implement accessor methods. Direct access to property values is also available through a new dot-notation syntax. here are three aspects to the syntax of properties: declaration, implementation, and access.

You can declare properties wherever methods can be declared in a class, category, or protocol declarative section. The syntax for declaring properties is:

```
@property(attributes...)type propertyName
```

where *attributes* are one or more optional attributes (comma-separated if multiple) that affect how the compiler stores instance variables and synthesizes accessor methods. The *type* element specifies an object type, declared type, or scalar type, such `id`, `NSString *`, `NSRange`, or `float`. The property must be backed by an instance variable of the same type and name.

The possible attributes in a property declaration are the following:

Attribute	Effect
<code>getter=<i>getterName</i></code> <code>setter=<i>setterName</i></code>	Specifies the names of getter and setter accessor methods (see “Storing and Accessing Properties” (page 126).) You specify these attributes when you are implementing your own accessor methods and want to control their names.
<code>readonly</code>	Indicates that the property can only be read from, not written to. The compiler does not synthesize a setter accessor or allow a nonsynthesized one to be called.
<code>readwrite</code>	Indicates that the property can be read from and written to. This is the default if <code>readonly</code> is not specified.
<code>assign</code>	Specifies that simple assignment should be used in the implementation of the setter; this is the default. If properties are declared in a non-garbage-collected program, you must specify <code>retain</code> or <code>copy</code> for properties that are objects.
<code>retain</code>	Specifies that <code>retain</code> should be sent to the property (which must be of an object type) upon assignment. Note that <code>retain</code> is a no-op in a garbage-collected environment.
<code>copy</code>	Specifies that <code>copy</code> should be sent to the property (which must be of an object type) upon assignment. The object’s class must implement the <code>NSCopying</code> protocol.
<code>nonatomic</code>	Specifies that accessor methods are synthesized as nonatomic. By default, all synthesized accessor methods are atomic: a getter method is guaranteed to return a valid value, even when other threads are executing simultaneously. For a discussion of atomic versus nonatomic properties, especially with regard to performance, see “Properties” in <i>The Objective-C 2.0 Programming Language</i> .

If you specify no attributes and specify `@synthesize` for the implementation, the compiler synthesizes getter and setter methods for the property that use simple assignment and that have the forms *propertyName* for the getter and `setPropertyName:` for the setter.

In the `@implementation` blocks of a class definition, you can use the `@dynamic` and `@synthesize` directives to control whether the compiler synthesizes accessor methods for particular properties. Both directives have the same general syntax:

```
@dynamic propertyName [ , propertyName2...];
@synthesize propertyName [ , propertyName2...];
```

The `@dynamic` directive tells the compiler that you are implementing accessor methods for the property, either directly or dynamically (such as when dynamically loading code). The `@synthesize` directive, on the other hand, tells the compiler to synthesize the getter and setter methods if they do not appear in the `@implementation` block. The syntax for `@synthesize` also includes an extension that allows you to name a property differently from its instance-variable storage. Consider, for example, the following statement:

```
@synthesize title, directReports, role = jobDescrip;
```

This tells the computer to synthesize accessor methods for properties `title`, `directReports`, and `role`, and to use the `jobDescrip` instance variable to back the `role` property.

Finally, the Objective-C properties feature supports a simplified syntax for accessing (getting and setting) properties through the use of dot notation and simple assignment. A few examples suffice to show how easy it is to get the values of properties and set them using this syntax:

```
NSString *title = employee.title; // assigns employee title to local variable
employee.ID = "A542309"; // assigns literal string to employee ID
// gets last name of this employee's manager
NSString *lname = employee.manager.lastName;
```

Note that dot-notation syntax works only for attributes and simple one-to-one relationships, not to-many relationships.

Further Reading: To learn more about declared properties, read “Properties” in *The Objective-C 2.0 Programming Language*.

Fast Enumeration

Fast enumeration is language feature introduced in Objective-C 2.0 that gives you a concise syntax for efficient enumeration of collections. It is much faster than the traditional use of `NSEnumerator` objects to iterate through arrays, sets, and dictionaries. Moreover, it ensures safe enumeration by including a mutation guard to prevent modification of a collection during enumeration. (An exception is thrown if a mutation is attempted.)

The syntax for fast enumeration is similar to that used in scripting languages such as Perl and Ruby; there are two supported versions:

```
for ( type newVariable in expression ) { statements }
```

and

```
type existingVariable;
for( existingVariable in expression ) { statements }
```

expression must evaluate to an object whose class conforms to the `NSFastEnumeration` protocol. The fast-enumeration implementation is shared between the Objective-C runtime and the Foundation framework. Foundation declares the `NSFastEnumeration` protocol, and the Foundation collection classes—`NSArray`, `NSDictionary`, and `NSSet`—and the `NSEnumerator` class adopt the protocol. Other classes that hold collections of other objects, including custom classes, may adopt `NSFastEnumeration` to take advantage of this feature.

The following snippet of code illustrates how you might use fast enumeration with `NSArray` and `NSSet` objects:

```
NSArray *array = [NSArray arrayWithObjects:
    @"One", @"Two", @"Three", @"Four", nil];

for (NSString *element in array) {
    NSLog(@"element: %@", element);
}

NSSet *set = [NSSet setWithObjects:
    @"Alpha", @"Beta", @"Gamma", @"Delta", nil];

NSString *setElement;
for (setElement in set) {
    NSLog(@"element: %@", setElement);
}
```

Further Reading: To find out more about fast enumeration, including how a custom collection class can take advantage of this feature, see “Fast Enumeration” in *The Objective-C 2.0 Programming Language*.

Using Objective-C

The way work gets done in an object-oriented program is through messages; one object sends a message to another object. Through the message, the sending object requests something from the receiving object (receiver). It requests that the receiver perform some action, return some object or value, or do both things.

Objective-C adopts a unique syntactical form for messaging. Take the following statement from the `SimpleCocoaTool` code in [Listing 2-2](#) (page 57):

```
NSEnumerator *enm = [sorted_args objectEnumerator];
```

The message expression is on the right side of the assignment, enclosed by the square brackets. The left-most item in the message expression is the receiver, a variable or expression representing the object to which the message is sent. In this case, the receiver is `sorted_args`, an instance of the `NSArray` class. Following the receiver is the message proper, in this case `objectEnumerator`. (For now, we are going to focus on message syntax and not look too deeply into what this and other messages in `SimpleCocoaTool` actually do.) The message `objectEnumerator` invokes a method of the `sorted_args` object named `objectEnumerator`, which returns a reference to an object that is held by the variable `enm` on the left side of the assignment. This variable is statically typed as an instance of the `NSEnumerator` class. You can diagram this statement as:

```
NSClassName *variable = [receiver message];
```

However, this diagram is simplistic and not really accurate. A message consists of a selector name and the parameters of the message. The Objective-C runtime uses a selector name, such as `objectEnumerator` above, to look up a selector in a table in order to find the method to invoke. A selector is a unique identifier that represents a method and that has a special type, `SEL`. Because it's so closely related, the selector name used to look up a selector is frequently called a selector as well. The above statement thus is more correctly shown as:

```
NSStringName *variable = [receiver selector];
```

Messages often have parameters, or arguments. A message with a single argument affixes a colon to the selector name and puts the argument right after the colon. This construct is called a keyword ; a keyword ends with a colon, and an argument follows the colon. Thus we could diagram a message expression with a single argument (and assignment) as the following:

```
NSStringName *variable = [receiver keyword: argument];
```

If a message has multiple arguments, the selector has multiple keywords. A selector name includes all keywords, including colons, but does not include anything else, such as return type or parameter types. A message expression with multiple keywords (plus assignment) could be diagrammed as follows:

```
NSStringName *variable = [receiver keyword1: arg1 keyword2: arg2];
```

As with function parameters, the type of an argument must match the type specified in the method declaration. Take as an example the following message expression from `SimpleCocoaTool`:

```
NSCountedSet *cset = [[NSCountedSet alloc] initWithArray:args];
```

Here `args`, which is also an instance of the `NSArray` class, is the argument of the message named `initWithArray:`.

The `initWithArray:` example cited above is interesting in that it illustrates nesting. With Objective-C, you can nest one message inside another message; the object returned by one message expression is used as the receiver by the message expression that encloses it. So to interpret nested message expressions, start with the inner expression and work your way outward. The interpretation of the above statement would be:

1. The `alloc` message is sent to the `NSCountedSet` class, which creates (by allocating memory for it) an uninitialized instance of the class.

Note: Objective-C classes are objects in their own right and you can send messages to them as well as to their instances. In a message expression, the receiver of a class message is always a class object.

2. The `initWithArray:` message is sent to the uninitialized instance, which initializes itself with the array `args` and returns a reference to itself.

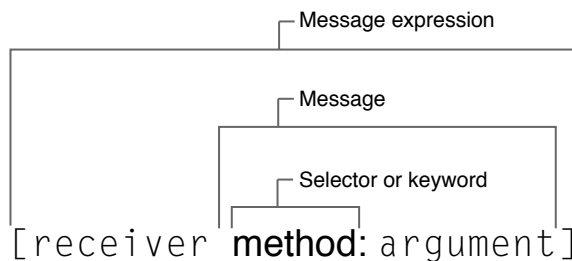
Next consider this statement from the `main` routine of `SimpleCocoaTool`:

```
NSArray *sorted_args = [[cset allObjects]
sortedArrayUsingSelector:@selector(compare:)];
```

What's noteworthy about this message expression is the argument of the `sortedArrayUsingSelector:` message. This argument requires the use of the `@selector` compiler directive to create a selector to be used as an argument.

Let's pause a moment to review message and method terminology. A method is essentially a function defined and implemented by the class of which the receiver of a message is a member. A message is a selector name (perhaps consisting of one or more keywords) along with its arguments; a message is sent to a receiver and this results in the invocation (or execution) of the method. A message expression encompasses both receiver and message. Figure 2-2 depicts these relationships.

Figure 2-2 Message terminology



Objective-C uses a number of defined types and literals that you won't find in ANSI C. In some cases, these types and literals replace their ANSI C counterparts. Table 2-1 describes a few of the important ones, including the allowable literals for each type.

Table 2-1 Important Objective-C defined types and literals

Type	Description and literal
id	The dynamic object type. Its negative literal is <code>nil</code> .
Class	The dynamic class type. Its negative literal is <code>Nil</code> .
SEL	The data type (typedef) of a selector. The negative literal of this type is <code>NULL</code> .
BOOL	A Boolean type. The literal values are <code>YES</code> and <code>NO</code> .

In your program's control-flow statements, you can test for the presence (or absence) of the appropriate negative literal to determine how to proceed. For example, the following `while` statement from the `SimpleCocoaTool` code implicitly tests the `word` object variable for the presence of a returned object (or, in another sense, the absence of `nil`):

```
while (word = [enm nextObject]) {
    printf("%s\n", [word UTF8String]);
}
```

In Objective-C, you can often send a message to `nil` with no ill effects. Return values from messages sent to `nil` are guaranteed to work as long as what is returned is typed as an object.

One final thing to note about the SimpleCocoaTool code is something that is not readily apparent if you're new to Objective-C. Compare this statement:

```
NSEnumerator *enm = [sorted_args objectEnumerator];
```

with this one:

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
```

On the surface, they seem to do identical things; both return a reference to an object. However there is an important semantic difference (for memory-managed code) that has to do with the ownership of the returned object, and hence the responsibility for freeing it. In the first statement, the SimpleCocoaTool program does not own the returned object. In the second statement, the program creates the object and so owns it. The last thing the program does is to send the `release` message to the created object, thus freeing it. The only other explicitly created object (the `NSCountedSet` instance) is also explicitly released at the end of the program. For a summary of the memory-management policy for object ownership and disposal, and the methods to use to enforce this policy, see [“How Memory Management Works”](#) (page 76).

The Root Class

Just by themselves, the Objective-C language and runtime are not enough to construct even the simplest object-oriented program, at least not easily. Something is still missing: a definition of the fundamental behavior and interface common to all objects. A root class supplies that definition.

A root class is so-called because it lies at the root of a class hierarchy—in this case, the Cocoa class hierarchy. The root class inherits from no other class, and all other classes in the hierarchy ultimately inherit from it. Along with the Objective-C language, the root class is primarily where Cocoa directly accesses and interacts with the Objective-C runtime. Cocoa objects derive the ability to behave as objects in large part from the root class.

Cocoa supplies two root classes: `NSObject` and `NSProxy`. Cocoa defines the latter class, an abstract superclass, for objects that act as stand-ins for other objects; thus `NSProxy` is essential in the distributed objects architecture. Because of this specialized role, `NSProxy` appears infrequently in Cocoa programs. When Cocoa developers refer to a root or base class, they almost always mean `NSObject`.

This section looks at `NSObject`, how it interacts with the runtime, and the basic behavior and interface it defines for all Cocoa objects. Primary among these are the methods it declares for allocation, initialization, memory management, introspection, and runtime support. These concepts are fundamental to an understanding of Cocoa.

NSObject

`NSObject` is the root class of most Objective-C class hierarchies; it has no superclass. From `NSObject`, other classes inherit a basic interface to the run-time system for the Objective-C language, and its instances obtain their ability to behave as objects.

Although it is not strictly an abstract class, `NSObject` is virtually one. By itself, an `NSObject` instance cannot do anything useful beyond being a simple object. To add any attributes and logic specific to your program, you must create one or more classes inheriting from `NSObject` or from any other class derived from `NSObject`.

`NSObject` adopts the `NSObject` protocol (see “Root Class—and Protocol” (page 70)). The `NSObject` protocol allows for multiple root objects. For example, `NSProxy`, the other root class, does not inherit from `NSObject` but adopts the `NSObject` protocol so that it shares a common interface with other Objective-C objects.

Root Class—and Protocol

`NSObject` is the name not only of a class but of a protocol. Both are essential to the definition of an object in Cocoa. The `NSObject` protocol specifies the basic programmatic interface required of *all* root classes in Cocoa; thus not only the `NSObject` class adopts the identically named protocol, but the other Cocoa root class, `NSProxy`, adopts it as well. The `NSObject` class further specifies the basic programmatic interface for any Cocoa object that is not a proxy object.

A protocol such as `NSObject` is used in the overall definition of Cocoa objects (rather than including those protocol methods in the class interface) to make multiple root classes possible. Each root class shares a common interface, as defined by the protocols they adopt.

In another sense, `NSObject` is not the only “root” protocol. Although the `NSObject` class does not formally adopt the `NSCopying`, `NSMutableCopying`, and `NSCoding` protocols, it declares and implements methods related to those protocols. (Moreover, the `NSObject.h` header file, which contains the definition of the `NSObject` class, also contains the definitions of all four protocols mentioned above.) Object copying, encoding, and decoding are fundamental aspects of object behavior. Many, if not most, subclasses are expected to adopt or conform to these protocols.

Note: Other Cocoa classes can (and do) add methods to `NSObject` through categories. These categories are often informal protocols used in delegation; they permit the delegate to choose which methods of the category to implement. However, these categories on `NSObject` are not considered part of the fundamental object interface.

Overview of Root-Class Methods

The `NSObject` root class, along with the adopted `NSObject` protocol and other “root” protocols, specify the following interface and behavioral characteristics for all non-proxy Cocoa objects:

- **Allocation, initialization, and duplication.** Some methods of `NSObject` (including some from adopted protocols) deal with the creation, initialization, and duplication of objects:
 - The `alloc` and `allocWithZone:` methods allocate memory for an object from a memory zone and set the object to point to its runtime class definition.
 - The `init` method is the prototype for object initialization, the procedure that sets the instance variables of an object to a known initial state. The class methods `initialize` and `load` give classes a chance to initialize themselves.
 - `new` is a convenience method that combines simple allocation and initialization.

- ❑ The `copy` and `copyWithZone:` methods make copies of any object that is a member of a class implementing these methods (from the `NSCopying` protocol); the `mutableCopy` and `mutableCopyWithZone:` (defined in the `NSMutableCopying` protocol) are implemented by classes that want to make mutable copies of their objects.

See “[Object Creation](#)” (page 80) for more information.

- **Object retention and disposal.** The following methods are particularly important to an object-oriented program that uses the traditional, and explicit, form of memory management:
 - ❑ The `retain` method increments an object’s retain count.
 - ❑ The `release` method decrements an object’s retain count.
 - ❑ The `autorelease` method also decrements an object’s retain count, but in a deferred fashion.
 - ❑ The `retainCount` method returns an object’s current retain count.
 - ❑ The `dealloc` method is implemented by class to release its objects’ instance variables and free dynamically allocated memory.

See the “[How Memory Management Works](#)” (page 76) for more information about explicit memory management.

- **Introspection and comparison.** Many `NSObject` methods enable you to make runtime queries about an object. These introspection methods help to discover an object’s position in the class hierarchy, determine whether it implements a certain method, and test whether it conforms to a specific protocol. Some of these are class methods only.
 - ❑ The `superclass` and `class` methods (class and instance) return the receiver’s superclass and class, respectively, as `Class` objects.
 - ❑ You can determine the class membership of objects with the methods `isKindOfClass:` and `isMemberOfClass:`; the latter method is for testing whether the receiver is an instance of the specified class. The class method `isSubclassOfClass:` tests class inheritance.
 - ❑ The `respondToSelector:` method tests whether the receiver implements a method identified by a selector. The class method `instancesRespondToSelector:` tests whether instances of a given class implement the specified method.
 - ❑ The `conformsToProtocol:` method tests whether the receiver (object or class) conforms to a given protocol.
 - ❑ The `isEqual:` and `hash` methods are used in object comparison.
 - ❑ The `description` method allows an object to return a string describing its contents; this output is often used in debugging (“print object” command) and by the “%@” specifier for objects in formatted strings.

See “[Introspection](#)” (page 90) for more information.

- **Object encoding and decoding.** The following methods pertain to object encoding and decoding (as part of the archiving process):
 - ❑ The `encodeWithCoder:` and `initWithCoder:` methods are the sole members of the `NSCoding` protocol. The first allows an object to encode its instance variables and the second enables an object to initialize itself from decoded instance variables.
 - ❑ The `NSObject` class declares other methods related to object encoding: `classForCoder`, `replacementObjectForCoder:`, and `awakeAfterUsingCoder:`.

See *Archives and Serializations Programming Guide for Cocoa* for further information.

- **Message forwarding.** The `forwardInvocation:` and related methods permit an object to forward a message to another object.
- **Message dispatch.** A set of methods beginning with `performSelector:...` allow you to dispatch messages after a specified delay and to dispatch messages (synchronously or asynchronously) from a secondary thread to the main thread.

`NSObject` has several other methods, including class methods for versioning and posing (the latter lets a class present itself to the runtime as another class). It also includes methods that let you access runtime data structures, such as method selectors and function pointers to method implementations.

Interface Conventions

Some `NSObject` methods are meant only to be invoked while others are intended to be overridden. For example, most subclasses should not override `allocWithZone:` but they should implement `init`—or at least an initializer that ultimately invokes the root-class `init` method (see “[Object Creation](#)” (page 80)). Of those methods that subclasses are expected to override, `NSObject`’s implementation either does nothing or returns some reasonable default value such as `self`. These default implementations make it possible to send basic messages such as `init` to any Cocoa object—even to an object whose class doesn’t override them—without risking a runtime exception. It’s not necessary to check (using `respondsToSelector:`) before sending the message. More importantly, the “placeholder” methods of `NSObject` define a common structure for Cocoa objects and establish conventions that, when followed by all classes, make object interactions more reliable.

Instance and Class Methods

The runtime system treats methods defined in the root class in a special way. Instance methods defined in a root class can be performed both by instances and by class objects. Therefore, all class objects have access to the instance methods defined in the root class. Any class object can perform any root instance method, provided it doesn’t have a class method with the same name.

For example, a class object could be sent messages to perform `NSObject`’s `respondsToSelector:` and `performSelector:withObject:` instance methods:

```
SEL method = @selector(riskAll:);

if ([MyClass respondsToSelector:method])
    [MyClass performSelector:method withObject:self];
```

Note that the only instance methods available to a class object are those defined in its root class. In the example above, if `MyClass` had re-implemented either `respondsToSelector:` or `performSelector:withObject:`, those new versions would be available only to instances. The class object for `MyClass` could perform only the versions defined in the `NSObject` class. (Of course, if `MyClass` had implemented `respondsToSelector:` or `performSelector:withObject:` as class methods rather than instance methods, the class would perform those new versions.)

Object Retention and Disposal

Beginning with Mac OS X version 10.5, Objective-C gives you two ways to ensure that objects persist when they are needed and are destroyed when they are no longer needed, thus freeing up memory. The preferred approach is to use the technology of garbage collection: the runtime detects objects that are no longer needed and disposes of them automatically. (It also happens to be the simpler approach in most cases.) The second approach, called memory management, is based on reference counting: an object carries with it a numerical value reflecting the current claims on the object; when this value reaches zero, the object is deallocated.

The amount of work that you, as a developer writing Objective-C code, must do to take advantage of garbage collection or memory management varies considerably.

- **Garbage Collection.** To enable garbage collection, turn on the Enable Objective-C Garbage Collection build setting (the `-fobjc-gc` flag). For each of your custom classes you might also have to implement the `finalize` method to remove instances as a notification observers and to free any resources that are not instance variables. Also, ensure that in your nib files the object acting as File's Owner maintains an outlet connection to each top-level nib object you want to persist.

iPhone OS Note:: Garbage collection is not supported on iPhone OS.

- **Memory Management.** In memory-managed code each call that makes a claim of ownership on an object—object allocation and initialization, object copying, and `retain`—must be balanced with a call that removes that claim—`release` and `autorelease`. When the object's retain count (reflecting the number of claims on it) reaches zero, the object is deallocated and the memory occupied by the object is freed.

In addition to being easier to implement, garbage-collected code has several advantages over memory-managed code. Garbage collection provides a simple, consistent model for all participating programs while avoiding problems such as retain cycles. It also simplifies the implementation of accessor methods and makes it easier to ensure thread and exception safety.

Important: Although memory-management methods are “no-ops” in a garbage-collected application, there are still intractable differences between certain programming paradigms and patterns used in the two models. Therefore it is not recommended that you migrate a memory-managed application to a source base that tries to support both memory management and garbage collection. Instead, create a new full-version release that supports garbage collection.

The following sections explore how garbage collection and memory management work by following the life cycle of objects from their creation to their destruction.

Further Reading: To learn all about the garbage-collection feature of Objective-C, read *Garbage Collection Programming Guide*. Memory management is discussed in detail in *Memory Management Programming Guide for Cocoa*.

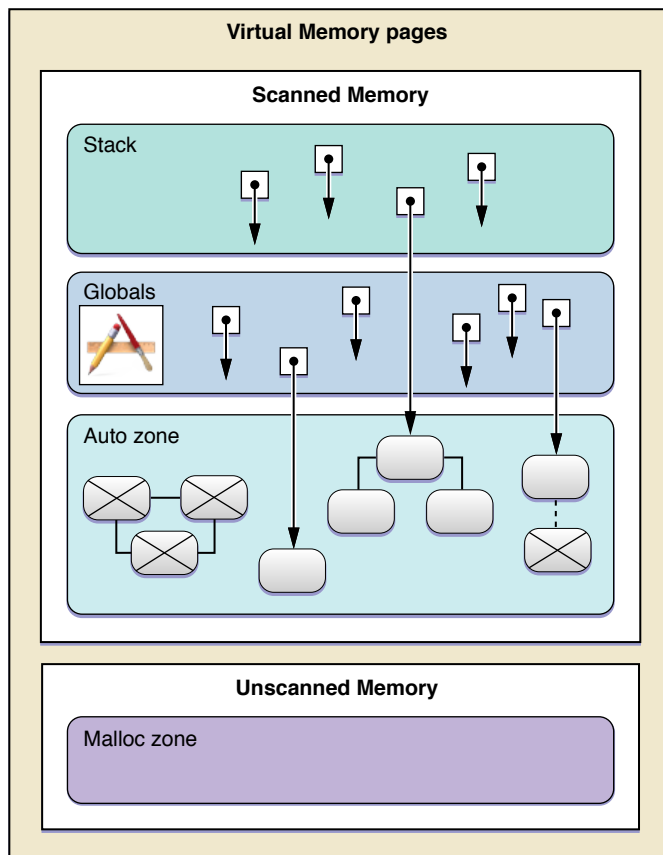
How Garbage Collection Works

To the garbage collector, objects in a program are either *reachable* or are not reachable. Periodically the collector scans through the objects and collects those that are reachable. Those objects that aren't reachable—the “garbage” objects—are finalized (that is, `finalize` is invoked). Subsequently, the memory they had occupied is freed.

The critical notion behind the architecture of the Objective-C garbage collector is the set of factors that constitute a reachable object. These start with an initial root set of objects: global variables (including `NSApp`), stack variables, and objects with external references (that is, outlets). The objects of the initial root set are never treated as garbage and therefore persist throughout the runtime life of the program. The collector adds to this initial set all objects that are directly reachable through strong references as well as all possible references found in the call stacks of every Cocoa thread. The garbage collector recursively follows strong references from the root set of objects to other objects, and from those objects to other objects, until all potentially reachable objects have been accounted for. (All references from one object to another object are considered strong references by default; weak references have to be explicitly marked as such.) In other words, a nonroot object persists at the end of a collection cycle only if the collector can reach it via strong references from a root object.

Figure 2-3 illustrates the general path the collector follows when it looks for reachable objects. But it also shows a few other important aspects of the garbage collector. The collector scans only a portion of a Cocoa program's virtual memory for reachable objects. Scanned memory includes the call stacks of threads, global variables, and the auto zone, an area of memory from which all garbage collected blocks of memory are dispensed. The collector does not scan the malloc zone, which is the zone from which blocks of memory are allocated via the `malloc` function.

Figure 2-3 Reachable and unreachable objects



Another thing the diagram illustrates is that objects may have strong references to other objects, but if there is no chain of strong references that leads back to a root object, the object is considered unreachable and is disposed of at the end of a collection cycle. And indeed these references can be circular, but in garbage collection the circular references do not cause memory leaks, as do retain cycles in memory-managed code. All of these objects are disposed of when they are no longer reachable.

The Objective-C garbage collector is request-driven, not demand-driven. It initiates collection cycles only upon request, which Cocoa makes at intervals optimized for performance or when a certain memory threshold has been exceeded. You can also request collections using methods of the `NSGarbageCollector` class. The garbage collector is also generational. It makes not only exhaustive, or full, collections of program objects periodically, but it makes incremental collections based on the “generation” of objects. A object’s generation is determined by when it was allocated. Incremental collections, which are faster and more frequent than full collections, affect the more recently allocated objects. (Most objects are assumed to “die young”; if an object survives the first collection, it is likely to be intended to have a longer life.)

The garbage collector runs on one thread of a Cocoa program. During a collection cycle it will stop secondary threads to determine which objects in those threads are unreachable. But it never stops all threads at once, and it stops each thread for as short a time as possible. The collector is also conservative in that it never compacts auto-zone memory by relocating blocks of memory or updating pointers; once allocated, an object always stays in its original memory location.

How Memory Management Works

In memory-managed Objective-C code, a Cocoa object exists over a life span which, potentially at least, has distinct stages. It is created, initialized, and used (that is, other objects send messages to it). It is possibly retained, copied, or archived, and eventually it is released and destroyed. The following discussion charts the life of a typical object without going into much detail—yet.

Let's begin at the end, at the way objects are disposed of when garbage collection is turned off. In this context Cocoa and Objective-C opt for a voluntary, policy-driven procedure for keeping objects around and disposing of them when they're no longer needed.

This procedure and policy rest on the notion of reference counting. Each Cocoa object carries with it an integer indicating the number of other objects (or even procedural code sites) that are interested in its persistence. This integer is referred to as the object's *retain count* ("retain" is used to avoid overloading the term "reference"). When you create an object, either by using a class factory method or by using the `alloc` or `allocWithZone:` class methods, Cocoa does a couple of very important things:

- It sets the object's *isa* pointer—the `NSObject` class's sole public instance variable—to point to the object's class, thus integrating the object into the runtime's view of the class hierarchy. (See "Object Creation" (page 80) for further information.)
- It sets the object's retain count—a kind of hidden instance variable managed by the runtime—to one. (The assumption here is that an object's creator is interested in its persistence.)

After object allocation, you generally initialize an object by setting its instance variables to reasonable initial values. (`NSObject` declares the `init` method as the prototype for this purpose.) The object is now ready to be used; you can send messages to it, pass it to other objects, and so on.

Note: Because an initializer can return an object other than the one explicitly allocated, the convention is to nest the `alloc` message expression in the `init` message (or other initializer)—for example:

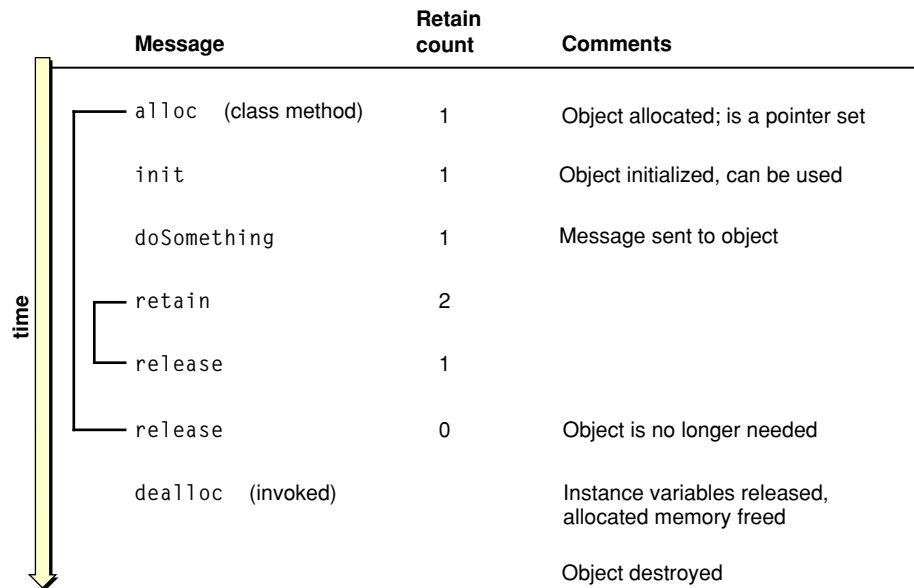
```
id anObj = [[MyClass alloc] init];
```

When you release an object—that is, send a `release` message to it—`NSObject` decrements its retain count. If the retain count falls from one to zero, the object is deallocated. Deallocation takes place in two steps. First, the object's `dealloc` method is invoked to release instance variables and free dynamically allocated memory. Then the operating system destroys the object itself and reclaims the memory the object once occupied.

Important: You should never directly invoke an object's `dealloc` method.

What if you don't want an object to go away any time soon? If after receiving an object from somewhere you send it a `retain` message, the object's retain count is incremented to two. Now two `release` messages are required before deallocation occurs. Figure 2-4 depicts this rather simplistic scenario.

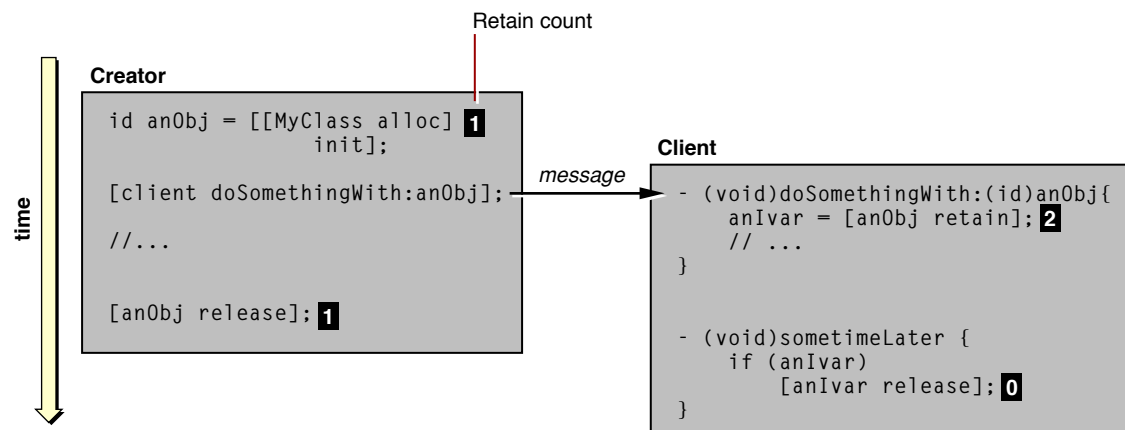
Figure 2-4 The life cycle of an object—simplified view



Of course, in this scenario the creator of an object has no need to retain the object. It owns the object already. But if this creator were to pass the object to another object in a message, the situation changes. In an Objective-C program, an object received from some other object is always assumed to be valid within the scope it is obtained. The receiving object can send messages to the received object and can pass it to other objects. This assumption requires the sending object to “behave” and not prematurely free the object while a client object has a reference to it.

If the client object wants to keep the received object around after it goes out of programmatic scope, it can retain it—that is, send it a `retain` message. Retaining an object increments its retain count, thereby expressing an ownership interest in the object. The client object assumes a responsibility to release the object at some later time. If the creator of an object releases it, but a client object has retained that same object, the object persists until the client releases it. Figure 2-5 illustrates this sequence.

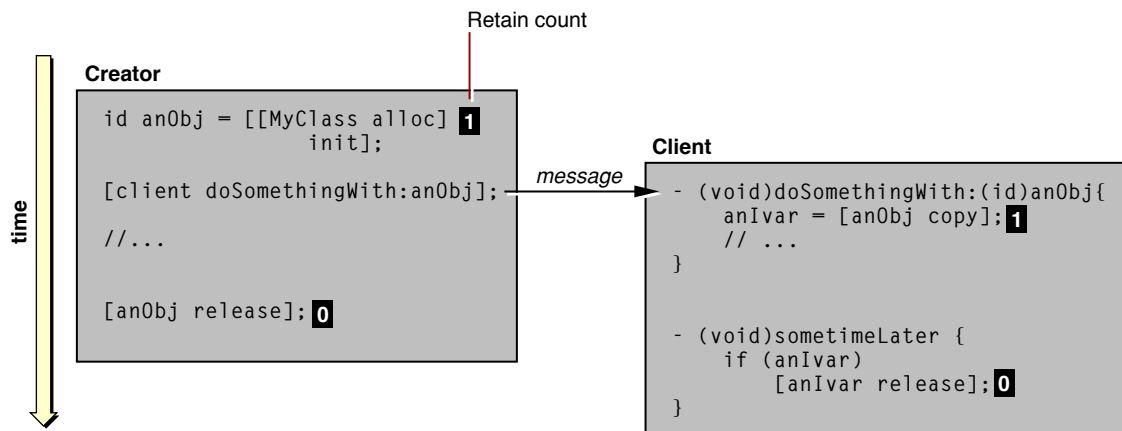
Figure 2-5 Retaining a received object



Instead of retaining an object you could copy it by sending it a `copy` or `copyWithZone:` message. (Many, if not most, subclasses encapsulating some kind of data adopt or conform to this protocol.) Copying an object not only duplicates it but almost always resets its retain count to one (see Figure 2-6). The copy can be shallow or deep, depending on the nature of the object and its intended usage. A deep copy duplicates the objects held as instance variables of the copied object while a shallow copy duplicates only the references to those instance variables.

In terms of usage, what differentiates a `copy` from a `retain` is that the former claims the object for the sole use of the new owner; the new owner can mutate the copied object without regard to its origin. Generally you copy an object instead of retaining it when it is a value object—that is, an object encapsulating some primitive value. This is especially true when that object is mutable, such as an `NSMutableString`. For immutable objects, `copy` and `retain` can be equivalent and might be implemented similarly.

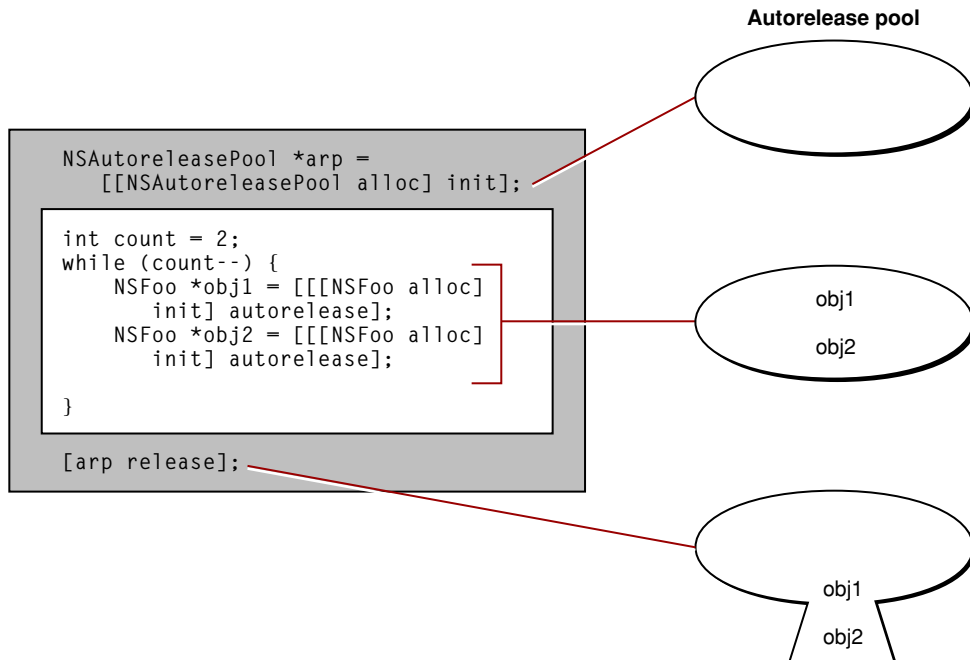
Figure 2-6 Copying a received object



You might have noticed a potential problem with this scheme for managing the object life cycle. An object that creates an object and passes it to another object cannot always know when it can release the object safely. There could be multiple references to that object on the call stack, some by objects unknown to the creating object. If the creating object releases the created object and then some other object sends a message to that now-destroyed object, the program could crash. To get around this problem, Cocoa introduces a mechanism for deferred deallocation called autoreleasing.

Autoreleasing makes use of autorelease pools (defined by the `NSAutoreleasePool` class). A autorelease pool is a collection of objects within an explicitly defined scope that are marked for eventual release. Autorelease pools can be nested. When you send an object an `autorelease` message, a reference to that object is put into the most immediate autorelease pool. It is still a valid object, so other objects within the scope defined by the autorelease pool can send messages to it. When program execution reaches the end of the scope, the pool is released and, as a consequence, all objects in the pool are released as well (see Figure 2-7). If you are developing an application you may not need to set up an autorelease pool; the Application Kit automatically sets up an autorelease pool scoped to the application's event cycle.

Figure 2-7 An autorelease pool



iPhone OS Note: Because on iPhone OS an application executes in a more memory-constrained environment, the use of autorelease pools is discouraged in methods or blocks of code (for example, loops) where an application creates many objects. Instead, you should explicitly release objects whenever possible.

So far the discussion of the object life cycle has focused on the mechanics of managing objects through that cycle. But a policy of object ownership guides the use of these mechanisms. This policy can be summarized as follows:

- If you *create* an object by allocating and initializing it (for example, `[[MyClass alloc] init]`), you own the object and are responsible for releasing it. This rule also applies if you use the `NSObject` convenience method `new`.
- If you *copy* an object, you own the copied object and are responsible for releasing it.
- If you *retain* an object, you have partial ownership of the object and must release it when you no longer need it.

Conversely,

- If you receive an object from some other object, you do not own the object and should not release it. (There are a handful of exceptions to this rule, which are explicitly noted in the reference documentation.)

As with any set of rules, there are exceptions and “gotchas”:

- If you create an object using a class factory method (such as the `NSMutableArray arrayWithCapacity:` method), assume that the object you receive has been autoreleased. You should not release the object yourself and should retain it if you want to keep it around.
- To avoid cyclic references, a child object should never retain its parent. (A parent is the creator of the child or is an object holding the child as instance variable.)

Note: “Release” in the above guidelines means sending either a `release` message or an `autorelease` message to an object.

If you do not follow this ownership policy, two bad things are likely to happen in your Cocoa program. Because you did not release created, copied, or retained objects, your program is now leaking memory. Or your program crashes because you sent a message to an object that was deallocated out from under you. And here’s a further caveat: debugging these problems can be a time-consuming affair.

A further basic event that could happen to an object during its life cycle is archiving. Archiving converts the web of interrelated objects that comprise an object-oriented program—the object graph—into a persistent form (usually a file) that preserves the identity and relationships of each object in the graph. When the program is unarchived, its object graph is reconstructed from this archive. To participate in archiving (and unarchiving), an object must be able to encode (and decode) its instance variables using the methods of the `NSCoder` class. `NSObject` adopts the `NSCoding` protocol for this purpose. For more information on the archiving of objects, see “Object Archives”.

Object Creation

The creation of a Cocoa object always takes place in two stages: allocation and initialization. Without both steps an object generally isn’t usable. Although in almost all cases initialization immediately follows allocation, the two operations play distinct roles in the formation of an object.

Allocating an Object

When you allocate an object, part of what happens is what you might expect, given the term “allocate.” Cocoa allocates enough memory for the object from a region of application virtual memory. To calculate how much memory to allocate, it takes the object’s instance variables into account—including their types and order—as specified by the object’s class.

To allocate an object, you send the message `alloc` or `allocWithZone:` to the object’s class. In return, you get a “raw” (uninitialized) instance of the class. The `alloc` variant of the method uses the application’s default zone. A zone is a page-aligned area of memory for holding related objects and data allocated by an application. See *Memory Management Programming Guide for Cocoa* for more information on zones.

An allocation message does other important things besides allocating memory:

- It sets the object’s retain count to one (as described in “How Memory Management Works” (page 76)).
- It initializes the object’s `isa` instance variable to point to the object’s class, a runtime object in its own right that is compiled from the class definition.

- It initializes all other instance variables to zero (or to the equivalent type for zero, such as `nil`, `NULL`, and `0.0`).

An object's `isa` instance variable is inherited from `NSObject`, so it is common to all Cocoa objects. After allocation sets `isa` to the object's class, the object is integrated into the runtime's view of the inheritance hierarchy and the current network of objects (class and instance) that constitute a program. Consequently an object can find whatever information it needs at runtime, such as another object's place in the inheritance hierarchy, the protocols that other objects conform to, and the location of the method implementations it can perform in response to messages.

In summary, allocation not only allocates memory for an object but initializes two small but very important attributes of any object: its `isa` instance variable and its retain count. It also sets all remaining instance variables to zero. But the resulting object is not yet usable. Initializing methods such as `init` must yet initialize objects with their particular characteristics and return a functional object.

Initializing an Object

Initialization sets the instance variables of an object to reasonable and useful initial values. It can also allocate and prepare other global resources needed by the object, loading them if necessary from an external source such as a file. Every object that declares instance variables should implement an initializing method—unless the default set-everything-to-zero initialization is sufficient. If an object does not implement an initializer, Cocoa invokes the initializer of the nearest ancestor instead.

The Form of Initializers

`NSObject` declares the `init` prototype for initializers; it is an instance method typed to return an object of type `id`. Overriding `init` is fine for subclasses that require no additional data to initialize their objects. But often initialization depends on external data to set an object to a reasonable initial state. For example, say you have an `Account` class; to initialize an `Account` object appropriately requires a unique account number, and this must be supplied to the initializer. Thus initializers can take one or more arguments; the only requirement is that the initializing method begins with the letters “init”. (The stylistic convention `init...` is sometimes used to refer to initializers.)

Note: Instead of implementing an initializer with arguments, a subclass may implement only a simple `init` method and then use “set” accessor methods immediately after initialization to set the object to a useful initial state. (Accessor methods enforce encapsulation of object data by setting and getting the values of instance variables.) Or, if the subclass uses properties and the related access syntax, it may assign values to the properties immediately after initialization.

Cocoa has plenty of examples of initializers with arguments. Here are a few (with the defining class in parentheses):

```
- (id)initWithArray:(NSArray *)array; (from NSSet)
- (id)initWithTimeInterval:(NSTimeInterval)secsToBeAdded sinceDate:(NSDate *)anotherDate; (from NSDate)
- (id)initWithContentRect:(NSRect)contentRect styleMask:(unsigned int)aStyle backing:(NSBackingStoreType)bufferingType defer:(BOOL)flag; (from NSWindow)
- (id)initWithFrame:(NSRect)frameRect; (from NSControl and NSView)
```

These initializers are instance methods that begin with “init” and return an object of the dynamic type `id`. Other than that, they follow the Cocoa conventions for multi-argument methods, often using `WithType:` or `FromSource:` before the first and most important argument.

Issues with Initializers

Although `init...` methods are required by their method signature to return an object, that object is not necessarily the one that was most recently allocated—the receiver of the `init...` message. In other words, the object you get back from an initializer might not be the one you thought was being initialized.

Two conditions prompt the return of something other than the just-allocated object. The first involves two related situations: when there must be a singleton instance or when the defining attribute of an object must be unique. Some Cocoa classes—`NSWorkspace`, for instance—allow only one instance in a program; a class in such a case must ensure (in an initializer or, more likely, in a class factory method) that only one instance is created, returning this instance if there is any further request for a new one. (See “[Creating a Singleton Instance](#)” (page 106) for information on implementing a singleton object.)

A similar situation arises when an object is required to have an attribute that makes it unique. Recall the hypothetical `Account` class mentioned earlier. An account of any sort must have a unique identifier. If the initializer for this class—say, `initWithAccountID:`—is passed an identifier that has already been associated with an object, it must do two things:

- Release the newly allocated object (in memory-managed code).
- Return the `Account` object previously initialized with this unique identifier.

By doing this, the initializer ensures the uniqueness of the identifier while providing what was asked for: an `Account` instance with the requested identifier.

Sometimes an `init...` method cannot perform the initialization requested. For example, an `initWithFile:` method expects to initialize an object from the contents of a file, the path to which is passed an argument. But if no file exists at that location the object cannot be initialized. A similar problem would happen if an `initWithArray:` initializer was passed an `NSDictionary` object instead of an `NSArray` object. When an `init...` method cannot initialize an object, it should:

- Release the newly allocated object (in memory-managed code).
- Return `nil`.

Returning `nil` from an initializer indicates that the requested object cannot be created. When you create an object, you should generally check whether the returned value is `nil` before proceeding:

```
id anObject = [[MyClass alloc] init];
if (anObject) {
    [anObject doSomething];
    // more messages...
} else {
    // handle error
}
```

Because an `init...` method might return `nil` or an object other than the one explicitly allocated, it is dangerous to use the instance returned by `alloc` or `allocWithZone:` instead of the one returned by the initializer. Consider the following code:

```
id myObject = [MyClass alloc];
[myObject init];
[myObject doSomething];
```

The `init` method in this example could have returned `nil` or could have substituted a different object. Because you can send a message to `nil` without raising an exception, nothing would happen in the former case except (perhaps) a debugging headache. But you should always rely on the initialized instance instead of the “raw” just-allocated one. It is recommended that you nest the allocation and initialization messages and test the object returned from the initializer before proceeding.

```
id myObject = [[MyClass alloc] init];
if ( myObject ) {
    [myObject doSomething];
} else {
    // error recovery...
}
```

Once an object is initialized, you should not initialize it again. If you attempt a re-initialization, the framework class of the instantiated object often raises an exception. For example, the second initialization in this example would result in an `NSInvalidArgumentException` being raised.

```
NSString *aStr = [[NSString alloc] initWithString:@"Foo"];
aStr = [aStr initWithString:@"Bar"];
```

Implementing an Initializer

There are several critical steps to follow when implementing an `init...` method that serves as a class's sole initializer or, if there are multiple initializers, its *designated initializer* (described in “[Multiple Initializers and the Designated Initializer](#)” (page 85)):

- Always invoke the superclass (`super`) initializer *first*.
- Check the object returned by the superclass. If it is `nil`, then initialization cannot proceed; return `nil` to the receiver.
- When initializing instance variables that are references to objects, retain or copy the object as necessary (in memory-managed code).
- After setting instance variables to valid initial values, return `self` unless:
 - It was necessary to return a substituted object, in which case release the freshly allocated object first (in memory-manged code).
 - A problem prevented initialization from succeeding, in which case return `nil`.

The `init...` method in Listing 2-3 illustrates these steps:

Listing 2-3 An example of an initializer

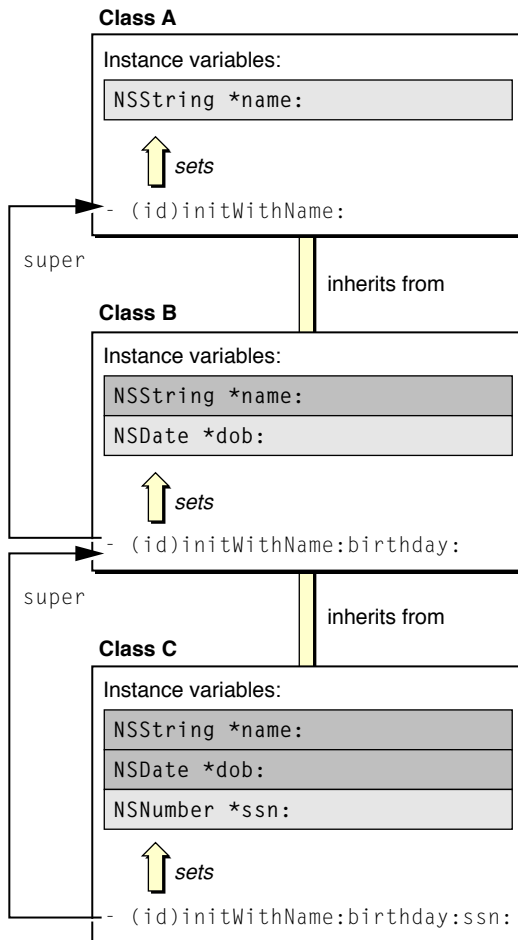
```
- (id)initWithAccountID:(NSString *)identifier {
    if ( self = [super init] ) {
        Account *ac = [accountDictionary objectForKey:identifier];
        if (ac) { // object with that ID already exists
            [self release];
            return [ac retain];
        }
    }
}
```

```
        if (identifier) {
            accountID = [identifier copy]; // accountID is instance variable
            [accountDictionary setObject:self forKey:identifier];
            return self;
        } else {
            [self release];
            return nil;
        }
    } else
        return nil;
}
```

Note: Although, for the sake of simplicity, this example returns `nil` if the argument is `nil`, the better Cocoa practice is to raise an exception.

It isn't necessary to initialize all instance variables of an object explicitly, just those that are necessary to make the object functional. The default set-to-zero initialization performed on an instance variable during allocation is often sufficient. Make sure that you retain or copy instance variables, as required for memory management.

The requirement to invoke the superclass' initializer as the first thing is important. Recall that an object encapsulates not only the instance variables defined by its class but the instance variables defined by all of its ancestor classes. By invoking `super`'s initializer first you help to ensure that the instance variables defined by classes up the inheritance chain are initialized first. The immediate superclass, in its initializer, invokes the initializer of its superclass, which invokes its superclass's main `init...` method, and so on (see Figure 2-8). The proper order of initialization is critical because the later initializations of subclasses may depend on superclass-defined instance variables being initialized to reasonable values.

Figure 2-8 Initialization up the inheritance chain

Inherited initializers are a concern when you create a subclass. Sometimes a superclass `init...` method sufficiently initializes instances of your class. But it is more likely it won't, and so you should override it. If you don't, the superclass implementation will be invoked, and since the superclass knows nothing about your class, your instances may not be correctly initialized.

Multiple Initializers and the Designated Initializer

A class can define more than one initializer. Sometimes multiple initializers let clients of the class provide the input for the same initialization in different forms. The `NSSet` class, for example, offers clients several initializers that accept the same data in different forms; one takes an `NSArray` object, another a counted list of elements, and another a `nil`-terminated list of elements:

```
- (id)initWithArray:(NSArray *)array;
- (id)initWithObjects:(id *)objects count:(unsigned)count;
- (id)initWithObjects:(id)firstObj, ...;
```

Some subclasses provide convenience initializers that supply default values to an initializer that takes the full complement of initialization parameters. This initializer is usually the designated initializer, the most important initializer of a class. For example, assume there is a `Task` class and it declares a designated initializer with this signature:

```
- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate;
```

The `Task` class might include secondary, or convenience, initializers that simply invoke the designated initializer, passing it default values for those parameters the secondary initializer doesn't explicitly request (Listing 2-4).

Listing 2-4 Secondary initializers

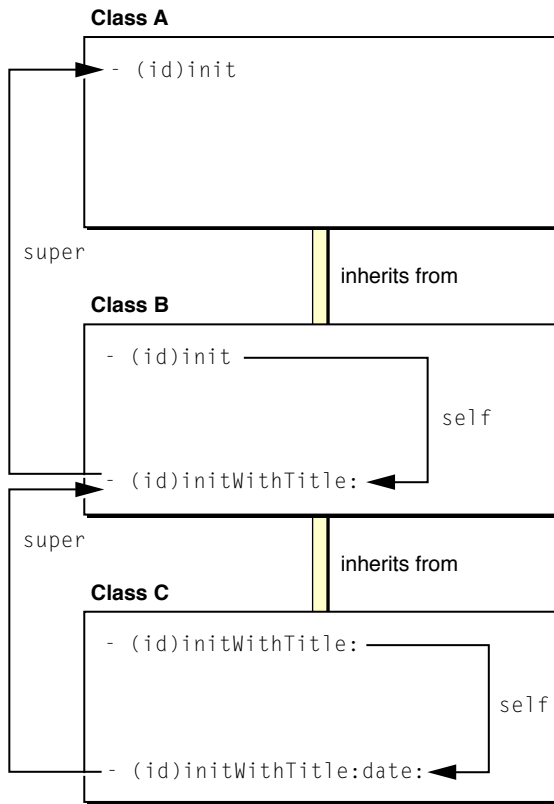
```
- (id)initWithTitle:(NSString *)aTitle {
    return [self initWithTitle:aTitle date:[NSDate date]];
}

- (id)init {
    return [self initWithTitle:@"Task"];
}
```

The designated initializer plays an important role for a class. It ensures that inherited instance variables are initialized by invoking `super`'s designated initializer. It is typically the `init...` method that has the most arguments and that does most of the initialization work, and it is the initializer that secondary initializers of the class invoke with messages to `self`.

When you define a subclass you must be able to identify the designated initializer of the superclass and invoke it in your subclass's designated initializer through a message to `super`. You must also make sure that inherited initializers are covered in some way. And you may provide as many convenience initializers as you deem necessary. When designing the initializers of your class, keep in mind that designated initializers are chained to each other through messages to `super` while other initializers are chained to the designated initializer of their class through messages to `self`.

An example will make this clearer. Let's say there are three classes, A, B, and C; class B inherits from class A, and class C inherits from class B. Each subclass adds an attribute as an instance variable and implements an `init...` method—the designated initializer—to initialize this instance variable. They also define secondary initializers and ensure that inherited initializers are overridden, if necessary. Figure 2-9 illustrates the initializers of all three class and their relationships.

Figure 2-9 Interactions of secondary and designated initializers

The designated initializer for each class is the initializer with the most coverage; it is the method that initializes the attribute added by the subclass. The designated initializer is also the `init...` method that invokes the designated initializer of the superclass in a message to `super`. In this example, the designated initializer of class C, `initWithTitle:date:`, invokes the designated initializer of its superclass, `initWithTitle:`, which in turn invokes the `init` method of class A. When creating a subclass, it's always important to know the designated initializer of the superclass.

While designated initializers are thus connected up the inheritance chain through messages to `super`, secondary initializers are connected to their class's designated initializer through messages to `self`. Secondary initializers (as in this example) are frequently overridden versions of inherited initializers. Class C overrides `initWithTitle:` to invoke its designated initializer, passing it a default date. This designated initializer, in turn, invokes the designated initializer of class B, which is the overridden method, `initWithTitle:`. If you sent an `initWithTitle:` message to objects of class B and class C, you'd be invoking different method implementations. On the other hand, if class C did *not* override `initWithTitle:` and you sent the message to an instance of class C, the class B implementation would be invoked. Consequently, the C instance would be incompletely initialized (since it would lack a date). When creating a subclass, it's important to make sure that all inherited initializers are adequately covered.

Sometimes the designated initializer of a superclass may be sufficient for the subclass, and so there is no need for the subclass to implement its own designated initializer. Other times, a class's designated initializer may be an overridden version of its superclass's designated initializer. This is frequently the case when the subclass needs to supplement the work performed by the superclass's designated initializer, even though the subclass does not add any instance variables of its own (or the instance variables it does add don't require explicit initialization).

The dealloc and finalize Methods

In Cocoa classes that use garbage collection, the `finalize` method is the place where the class disposes of any remaining resources and attachments of its instances before those instances are freed. In Cocoa classes that use traditional memory management, the comparable method for resource cleanup is the `dealloc` method. Although similar in purpose, there are significant differences in how these methods should be implemented.

In many respects, the `dealloc` method is the counterpart to a class's `init...` method, especially its designated initializer. Instead of being invoked just after the allocation of an object, `dealloc` is invoked just prior to the object's destruction. Instead of ensuring that the instance variables of an object are properly initialized, the `dealloc` method makes sure that object instance variables are released and that any dynamically allocated memory has been freed.

The final point of parallelism has to do with the invocation of the superclass implementation of the same method. In an initializer, you invoke the superclass's designated initializer as the first step. In `dealloc`, you invoke the superclass's `dealloc` implementation as the *last* step. The reason for this is mirror-opposite to that for initializers; subclasses should release or free the instance variables they own first before the instance variables of ancestor classes are released or freed.

Listing 2-5 shows how you might implement this method.

Listing 2-5 An example `dealloc` method

```
- (void)dealloc {
    [accountDictionary release];
    if ( mallocdChunk != NULL )
        free(mallocdChunk);
    [super dealloc];
}
```

Note that, as shown in this example, it's prudent to check whether a `malloc'd` instance variable is non-NULL before attempting to free it. Because you can safely send a message to a `nil` object, such a precaution is not necessary with object instance variables.

Similar to the `dealloc` method, the `finalize` method is the place to close resources used by an object in a garbage-collected environment prior to that object being freed and its memory reclaimed. As in `dealloc`, the final line of a `finalize` implementation should invoke the superclass implementation of the method. However, unlike `dealloc`, a `finalize` implementation does not have to release instance variables because the garbage collector destroys these objects at the proper time.

But there is a more significant difference between these “cleanup” methods. It is actually recommended that you do *not* implement a `finalize` method if possible. And, if you must implement `finalize`, you should reference as few other objects as possible. The primary reason for this admonition is that the order in which garbage-collected objects are sent `finalize` is indeterminate, even if there are references between them. Thus the consequences are indeterminate, and potentially negative, if messages pass between objects being finalized. Your code cannot depend on the side effects arising from the order of deallocation, as it can in `dealloc`. Generally, you should try to architect your code so that such actions as freeing `malloc'd` blocks, closing file descriptors, and unregistering observers happen before `finalize` is invoked.

Further Reading: To learn about approaches to implementing the `finalize` method, read “Implementing a `finalize` Method” in *Garbage Collection Programming Guide*.

Class Factory Methods

Class factory methods are implemented by a class as a convenience for clients. They combine allocation and initialization in one step and return the created object autoreleased (in memory-managed code). These methods are of the form `+(type)className...` (where *className* excludes any prefix).

Cocoa provides plenty of examples, especially among the “value” classes. `NSDate` includes the following class factory methods:

```
+ (id)dateWithTimeIntervalSinceNow:(NSTimeInterval)secs;
+ (id)dateWithTimeIntervalSinceReferenceDate:(NSTimeInterval)secs;
+ (id)dateWithTimeIntervalSince1970:(NSTimeInterval)secs;
```

And `NSData` offers the following factory methods:

```
+ (id)dataWithBytes:(const void *)bytes length:(unsigned)length;
+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length;
+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length
    freeWhenDone:(BOOL)b;
+ (id)dataWithContentsOfFile:(NSString *)path;
+ (id)dataWithContentsOfURL:(NSURL *)url;
+ (id)dataWithContentsOfMappedFile:(NSString *)path;
```

Factory methods can be more than a simple convenience. They can not only combine allocation and initialization, but the allocation can inform the initialization. As an example, let’s say you must initialize a collection object from a property-list file that encodes any number of elements for the collection (`NSString` objects, `NSData` objects, `NSNumber` objects, and so on). Before the factory method can know how much memory to allocate for the collection, it must read the file and parse the property list to determine how many elements there are and what object type these elements are.

Another purpose for a class factory method is to ensure that a certain class (`NSWorkspace`, for example) vends a singleton instance. Although an `init...` method could verify that only one instance exists at any one time in a program, it would require the prior allocation of a “raw” instance and then, in memory-managed code, would have to release that instance. A factory method, on the other hand, gives you a way to avoid blindly allocating memory for an object that you might not use (see Listing 2-6).

Listing 2-6 A factory method for a singleton instance

```
static AccountManager *DefaultManager = nil;

+ (AccountManager *)defaultManager {
    if (!DefaultManager) DefaultManager = [[self allocWithZone:NULL] init];
    return DefaultManager;
}
```

Further Reading: For a more detailed discussion of issues relating to the allocation and initialization of Cocoa objects, see “The Runtime System” in *The Objective-C 2.0 Programming Language*.

Introspection

Introspection is a powerful feature of object-oriented languages and environments, and Objective-C and Cocoa are particularly generous with this feature. Introspection refers to the capability of objects to divulge details about themselves as objects at runtime. Such details include an object’s place in the inheritance tree, whether it conforms to a specific protocol, and whether it responds to a certain message. The `NSObject` protocol and class define many introspection methods that you can use to query the runtime in order to characterize objects.

Used judiciously, introspection makes an object-oriented program more efficient and robust. It can help you to avoid message-dispatch errors, erroneous assumptions of object equality, and similar problems. The following sections show how you might effectively use the `NSObject` introspection methods in your code.

Evaluating Inheritance Relationships

Once you know the class an object belongs to, you probably know quite a bit about the object. You might know what its capabilities are, what attributes it represents, and what kinds of messages it can respond to. Even if after introspection you are unfamiliar with the class to which an object belongs, you now know enough to not send it certain messages.

The `NSObject` protocol declares several methods for determining an object’s position in the class hierarchy. These methods operate at different granularities. The `class` and `superclass` instance methods, for example, return the `Class` objects representing the class and superclass, respectively, of the receiver. These methods require you to compare one `Class` object with another. Listing 2-7 gives a simple (one might say trivial) example of their use.

Listing 2-7 Using the `class` and `superclass` methods

```
// ...
while ( id anObject = [objectEnumerator nextObject] ) {
    if ( [self class] == [anObject superclass] ) {
        // do something appropriate...
    }
}
```

Note: Sometimes you use the `class` or `superclass` methods to obtain an appropriate receiver for a class message.

More commonly to check an object’s class affiliation, you would send it a `isKindOfClass:` or `isMemberOfClass:` message. The former method returns whether the receiver is an instance of a given class or an instance of any class that inherits from that class. A `isMemberOfClass:` message, on the other hand, tells you if the receiver is an instance of the specified class. The `isKindOfClass:` method is generally more useful because from it you can know at once the complete range of messages you can send to an object. Consider the code snippet in Listing 2-8.

Listing 2-8 Using `isKindOfClass:`

```
if ([item isKindOfClass:[NSData class]]) {
    const unsigned char *bytes = [item bytes];
    unsigned int length = [item length];
    // ...
}
```

By learning that the object *item* inherits from the `NSData` class, this code knows it can send it the `NSData` `bytes` and `length` messages. The difference between `isKindOfClass:` and `isMemberOfClass:` becomes apparent if you assume that *item* is an instance of `NSMutableData`. If you use `isMemberOfClass:` instead of `isKindOfClass:`, the code in the conditionalized block is never executed because *item* is not an instance of `NSData` but rather of `NSMutableData`, a subclass of `NSData`.

Method Implementation and Protocol Conformance

Two of the more powerful introspection methods of `NSObject` are `respondToSelector:` and `conformsToProtocol:`. These methods tell you, respectively, whether an object implements a certain method and whether an object conforms to a specified formal protocol (that is, adopts the protocol, if necessary, and implements all the methods of the protocol).

You use these methods in a similar situation in your code. They enable you to discover whether some potentially anonymous object will respond appropriately to a particular message or set of messages *before* you send it any of those messages. By making this check before sending a message, you can avoid the risk of runtime exceptions resulting from unrecognized selectors. The Application Kit implements informal protocols—the basis of delegation—by checking whether delegates implement a delegation method (using `respondToSelector:`) prior to invoking that method.

Listing 2-9 illustrates how you might use the `respondToSelector:` method in your code:

Listing 2-9 Using `respondToSelector:`

```
- (void)doCommandBySelector:(SEL)aSelector {
    if ([self respondsToSelector:aSelector]) {
        [self performSelector:aSelector withObject:nil];
    } else {
        [_client doCommandBySelector:aSelector];
    }
}
```

Listing 2-10 illustrates how you might use the `conformsToProtocol:` method in your code:

Listing 2-10 Using `conformsToProtocol:`

```
// ...
if (!([((id)testObject) conformsToProtocol:@protocol(NSMenuItem)])) {
    NSLog(@"Custom MenuItem, '%@'", not loaded; it must conform to the
        'NSMenuItem' protocol.\n", [testObject class]);
    [testObject release];
    testObject = nil;
}
```

Object Comparison

Although they are not strictly introspection methods, the `hash` and `isEqual:` methods fulfill a similar role. They are indispensable runtime tools for identifying and comparing objects. But instead of querying the runtime for information about an object, they rely on class-specific comparison logic.

The `hash` and `isEqual:` methods, both declared by the `NSObject` protocol, are closely related. The `hash` method must be implemented to return an integer that can be used as a table address in a hash table structure. If two objects are equal (as determined by the `isEqual:` method), they must have the same hash value. If your object could be included in collections such as `NSSet`, you need to define `hash` and verify the invariant that if two objects are equal, they return the same hash value. The default `NSObject` implementation of `isEqual:` simply checks for pointer equality.

Using the `isEqual:` method is straightforward; it compares the receiver against the object supplied as argument. Object comparison frequently informs runtime decisions about what should be done with an object. As Listing 2-11 illustrates, you can use `isEqual:` to decide whether to perform an action, in this case to save user preferences that have been modified.

Listing 2-11 Using `isEqual:`

```
- (void)saveDefaults {
    NSDictionary *prefs = [self preferences];
    if (![origValues isEqual:prefs])
        [Preferences savePreferencesToDefaults:prefs];
}
```

If you are creating a subclass, you might need to override `isEqual:` to add further checks for points of equality. The subclass might define an extra attribute that might have to be the same value in two instances for them to be considered equal. For example, say you create a subclass of `NSObject` called `MyWidget` that contains two instance variables, `name` and `data`. Both of these must be the same value for two instances of `MyWidget` to be considered equal. Listing 2-12 illustrates how you might implement `isEqual:` for the `MyWidget` class.

Listing 2-12 Overriding `isEqual:`

```
- (BOOL)isEqual:(id)other {
    if (other == self)
        return YES;
    if (!other || ![other isKindOfClass:[self class]])
        return NO;
    return [self isEqualToWidget:other];
}

- (BOOL)isEqualToWidget:(MyWidget *)aWidget {
    if (self == aWidget)
        return YES;
    if (![self name] isEqual:[aWidget name])
        return NO;
    if (![self data] isEqualToData:[aWidget data])
        return NO;
    return YES;
}
```


This `isEqual:` method first checks for pointer equality, then class equality, and finally invokes an object comparator whose name indicates the class of object involved in the comparison. This type of comparator, which forces type-checking of the object passed in, is a common convention in Cocoa; the `NSString isEqualToString:` and the `NSTimeZone isEqualToTimeZone:` are but two examples. The class-specific comparator—`isEqualToWidget:` in this case—performs the checks for name and data equality.

In all `isEqualType:` methods of the Cocoa frameworks `nil` is not a valid argument and implementations of these methods may raise an exception upon receiving a `nil`. However, for backwards compatibility, `isEqual:` methods of the Cocoa frameworks do accept `nil`, returning `NO`.

Object Mutability

Cocoa objects are either mutable or immutable. You cannot change the encapsulated values of immutable objects; once such an object is created, the value it represents remains the same throughout the object's life. But you can change the encapsulated value of a mutable object at any time. The following sections explain the reasons for having mutable and immutable variants of an object type, describes the characteristics and side-effects of object mutability, and recommends how best to handle objects when their mutability is an issue.

Why Mutable and Immutable Object Variants?

Objects by default are mutable. Most objects allow you to change their encapsulated data through “setter” accessor methods. For example, you can change the size, positioning, title, buffering behavior, and other characteristics of an `NSWindow` object. A well-designed model object—say, an object representing a customer record—*requires* setter methods to change its instance data.

The Foundation framework adds some nuance to this picture by introducing classes that have mutable and immutable variants. The mutable subclasses are typically subclasses of their immutable superclass and have “Mutable” embedded in the class name. These classes include the following:

```
NSMutableArray  
NSMutableDictionary  
NSMutableSet  
NSMutableIndexSet  
NSMutableCharacterSet  
NSMutableData  
NSMutableString  
NSMutableAttributedString  
NSMutableURLRequest
```

Note: Except for `NSMutableParagraphStyle` in the Application Kit, the Foundation framework currently defines all explicitly named mutable classes. However, any Cocoa framework can potentially have its own mutable and immutable class variants.

Although these classes have atypical names, they are closer to the mutable norm than their immutable counterparts. Why this complexity? What purpose does having an immutable variant of a mutable object serve?

Consider a scenario where all objects are capable of being mutated. In your application you invoke a method and are handed back a reference to an object representing a string. You use this string in your user interface to identify a particular piece of data. Now another subsystem in your application gets its own reference to that same string and decides to mutate it. Suddenly your label has changed out from under you. Things can become even more dire if, for instance, you get a reference to an array that you use to populate a table view. The user selects a row corresponding to an object in the array that has been removed by some code elsewhere in the program—problems ensue. Immutability is a guarantee that an object won't unexpectedly change in value while you're using it.

Objects that are good candidates for immutability are ones that encapsulate collections of discrete values or contain values that are stored in buffers (which are themselves kinds of collections, either of characters or bytes). But not all “value” objects necessarily benefit from having mutable versions. Objects that contain a single simple value, such as instances of `NSNumber` or `NSDate`, are not good candidates for mutability. When the represented value changes in these cases, it makes more sense to replace the old instance with a new instance.

Performance is also a reason for immutable versions of objects representing things such as strings and dictionaries. Mutable objects such as these bring some overhead with them. Because they must dynamically manage a changeable backing store—allocating and deallocating chunks of memory as needed—mutable objects can be less efficient than their immutable counterparts.

Although in theory immutability guarantees that an object's value is stable, in practice this guarantee isn't always assured. A method may choose to hand out a mutable object under the return type of its immutable variant; later, it may decide to mutate the object, possibly violating assumptions and choices the recipient has made based on the earlier value. The mutability of an object itself may change as it undergoes various transformations. For example, serializing a property list (using the `NSPropertyListSerialization` class) does not preserve the mutability aspect of objects, only their general kind—a dictionary, an array, and so on. Thus, when you deserialize this property list, the resulting objects might not be of the same class as the original objects. For instance, what was once an `NSMutableDictionary` object might now be a `NSDictionary` object.

Programming With Mutable Objects

When the mutability of objects is an issue, it's best to adopt some defensive programming practices. Here are a few general rules of thumb:

- Use a mutable variant of an object when you need to modify its contents frequently and incrementally after it has been created.
- Sometimes it's preferable to replace one immutable object with another; for example, most instance variables that hold string values should be assigned immutable `NSString` objects that are replaced with “setter” methods.
- Rely on the return type for indications of mutability.

- if you have any doubts about whether an object is, or should be, mutable, go with immutable.

This section explores the gray areas in these rules of thumb, discussing typical choices you have to make when programming with mutable objects. It also gives an overview of methods in the Foundation framework for creating mutable objects and for converting between mutable and immutable object variants.

Creating and Converting Mutable Objects

You can create a mutable object through the standard nested `alloc-init` message, for example:

```
NSMutableDictionary *mutDict = [[NSMutableDictionary alloc] init];
```

However, many mutable classes offer initializers and factory methods that let you specify the initial or probable capacity of the object, such as the `arrayWithCapacity:` class method of `NSMutableArray`:

```
NSMutableArray *mutArray = [NSMutableArray arrayWithCapacity:[timeZones count]];
```

This hint enables more efficient storage of the mutable object's data. (Because the convention for class factory methods is to return autoreleased instances, be sure to retain the object if you wish to keep it viable in your code.)

You can also create a mutable object by making a mutable copy of an existing object of that general type. To do so, invoke the `mutableCopy` method that each immutable super class of a Foundation mutable class implements:

```
NSMutableSet *mutSet = [aSet mutableCopy];
```

In the other direction, you can send `copy` to a mutable object to make an immutable copy of it.

Many Foundation classes with immutable and mutable variants include methods for converting between the variants, including:

- *typeWithType:*—for example, `arrayWithArray:`
- *setType:*—for example, `setString:` (mutable classes only)
- *initWithType:copyItems:*—for example, `initWithDictionary:copyItems:`

Storing and Returning Mutable Instance Variables

A common decision in Cocoa development is whether to make an instance variable mutable or immutable. For an instance variable whose value can change, such as a dictionary or string, when is it appropriate to make the object mutable? And when is it better to make the object immutable and replace it with another object when its represented value changes?

Generally, when you have an object whose contents change wholesale, it's better to use an immutable object. Strings (`NSString`) and data objects (`NSData`) usually fall into this category. If an object is likely to change incrementally, it is a reasonable approach to make it mutable. Collections such as arrays and dictionaries fall into this category. However, the frequency of changes and the size of the collection should be factors in this decision. For example, if you have a small array that seldom changes, it's better to make it immutable.

There are a couple of other considerations when deciding on the mutability of an collection held as an instance variable:

- If you have a mutable collection that is frequently changed and that you frequently hand out to clients (that is, you return it directly in a “getter” accessor method) you run the risk of mutating something that your clients might have a reference to. If this risk is probable, the instance variable should be immutable.
- If the value of the instance variable frequently changes but you rarely return it to clients in getter methods, you can make the instance variable mutable but return an immutable copy of it in your accessor method; in memory-managed programs, this object would be autoreleased (Listing 2-13).

Listing 2-13 Returning an immutable copy of a mutable instance variable

```
@interface MyClass : NSObject {
    // ...
    NSMutableSet *widgets;
}
// ...
@end

@implementation MyClass
- (NSSet *)widgets {
    return (NSSet *)[[widgets copy] autorelease];
}
```

One sophisticated approach for handling mutable collections that are returned to clients is to maintain a flag that records whether the object is currently mutable or immutable. If there is a change, make the object mutable and apply the change. When handing out the collection, make the object immutable (if necessary) before returning it.

Receiving Mutable Objects

The invoker of a method is interested in the mutability of a returned object for two reasons:

- It wants to know if it can change the object’s value.
- It wants to know if the object’s value will change unexpectedly while it has a reference to it.

Use Return Type, Not Introspection

To determine whether it can change a received object, the receiver must rely on the formal type of the return value. If it receives, for instance, an array object typed as immutable, it should not attempt to mutate it. It is not an acceptable programming practice to determine if an object is mutable based on its class membership, for example:

```
if ( [anArray isKindOfClass:[NSMutableArray class]] ) {
    // add, remove objects from anArray
}
```

Because of implementation reasons, what `isKindOfClass:` returns in this case may not be accurate. But for reasons other than this, you should not make decisions on whether an object is mutable based on class membership. That decision should be guided solely by what the signature of the method vending the object says about its mutability. If you are not sure whether an object is mutable or immutable, assume it's immutable.

A couple of examples might help clarify why this guideline is important:

- You read a property list from a file. When the Foundation framework processes the list it notices that various subsets of the property list are identical, so it creates a set of objects that it shares among all those subsets. Afterwards you look at the created property list objects and decide to mutate one subset. Suddenly, and without being aware of it, you've changed the tree in multiple places.
- You ask `NSView` for its subviews (`subviews` method) and it returns an object that is declared to be an `NSArray` but which could be an `NSMutableArray` internally. Then you pass that array to some other code that, through introspection, determines it to be mutable and changes it. By changing this array, the code is mutating `NSView`'s internal data structures.

So don't make a decision on object mutability based on what introspection tells you about an object. Treat objects as mutable or not based on what you are handed at the API boundaries (that is, based on the return type). If you need to unambiguously mark an object as mutable or immutable when you pass it to clients, pass that information as a flag along with the object.

Make Snapshots of Received Objects

If you want to ensure that a supposedly immutable object received from a method does not mutate without your knowing about it, you can make "snapshots" of the object by copying it locally. Then occasionally compare the stored version of the object with the most recent version. If the object has mutated, you can adjust anything in your program that is dependent on the previous version of the object. Listing 2-14 shows a possible implementation of this technique.

Listing 2-14 Making a snapshot of a potentially mutable object

```
static NSArray *snapshot = nil;
- (void)myFunction {
    NSArray *thingArray = [otherObj things];
    if (snapshot) {
        if ( ![thingArray isEqualToArray:snapshot] ) {
            [self updateStateWith:thingArray];
        }
    }
    snapshot = [thingArray copy];
}
```

A problem with making snapshots of objects for later comparison is that it is expensive. You're required to make multiple copies of the same object. A more efficient alternative is to use key-value observing. See "[Key-Value Observing](#)" (page 158) for an overview of this protocol.

Mutable Objects in Collections

Storing mutable objects in collection objects can cause problems. Certain collections can become invalid or even corrupt if objects they contain mutate because, by mutating, these objects can affect the way they are placed in the collection. In the first case, the properties of objects that are keys in

hashing collections such as `NSDictionary` objects or `NSSet` objects will, if changed, corrupt the collection if the changed properties affect the results of the object's `hash` or `isEqual:` methods. (If the `hash` method of the objects in the collection does not depend on their internal state, corruption is less of a likelihood.) Second, if an object in an ordered collection such as a sorted array has its properties changed, this might affect how the object compares to other objects in the array, thus rendering the ordering invalid.

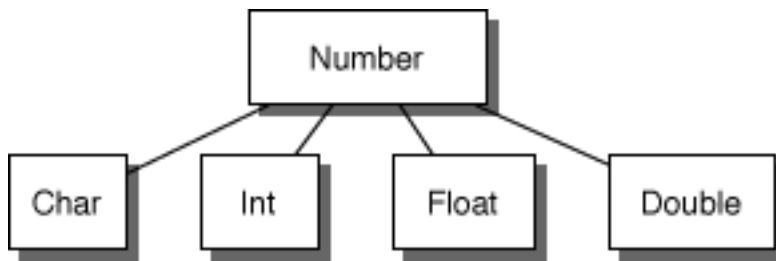
Class Clusters

Class clusters are a design pattern that the Foundation framework makes extensive use of. Class clusters group a number of private, concrete subclasses under a public, abstract superclass. The grouping of classes in this way simplifies the publicly visible architecture of an object-oriented framework without reducing its functional richness. Class clusters are based on the Abstract Factory design pattern discussed in “Cocoa Design Patterns” (page 141).

Simple Concept, Complex Interface

To illustrate the class cluster architecture and its benefits, consider the problem of constructing a class hierarchy that defines objects to store numbers of different types (`char`, `int`, `float`, `double`). Since numbers of different types have many features in common (they can be converted from one type to another and can be represented as strings, for example), they could be represented by a single class. However, their storage requirements differ, so it's inefficient to represent them all by the same class. This suggests the following architecture:

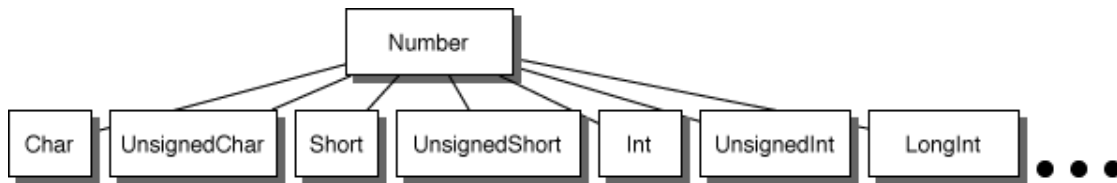
Figure 2-10 A simple hierarchy for number classes



`Number` is the abstract superclass that declares in its methods the operations common to its subclasses. However, it doesn't declare an instance variable to store a number. The subclasses declare such instance variables and share in the programmatic interface declared by `Number`.

So far, this design is relatively simple. However, if the commonly used modifications of these basic C types are taken into account, the diagram looks more like this:

Figure 2-11 A more complete number class hierarchy

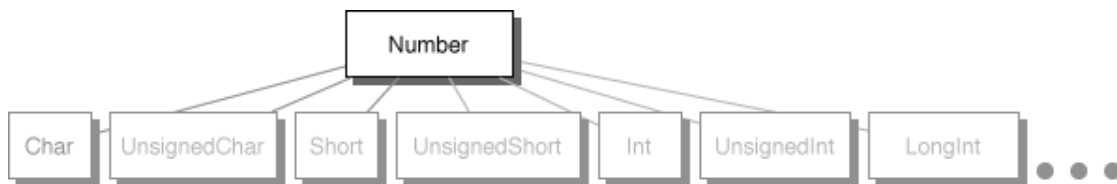


The simple concept—creating a class to hold number values—can easily burgeon to over a dozen classes. The class cluster architecture presents a design that reflects the simplicity of the concept.

Simple Concept, Simple Interface

Applying the class cluster design pattern to this problem yields the following hierarchy (private classes are in gray):

Figure 2-12 Class cluster architecture applied to number classes



Users of this hierarchy see only one public class, `Number`, so how is it possible to allocate instances of the proper subclass? The answer is in the way the abstract superclass handles instantiation.

Creating Instances

The abstract superclass in a class cluster must declare methods for creating instances of its private subclasses. It's the superclass's responsibility to dispense an object of the proper subclass based on the creation method that you invoke—you don't, and can't, choose the class of the instance.

In the Foundation framework, you generally create an object by invoking a `+ className... method` or the `alloc...` and `init...` methods. Taking the Foundation framework's `NSNumber` class as an example, you could send these messages to create number objects:

```
NSNumber *aChar = [NSNumber numberWithInt:'a'];  
NSNumber *anInt = [NSNumber numberWithInt:1];  
NSNumber *aFloat = [NSNumber numberWithFloat:1.0];  
NSNumber *aDouble = [NSNumber numberWithDouble:1.0];
```

(This style of instantiation creates objects that will be deallocated automatically—See [“Class Factory Methods”](#) (page 89) for more information. Many classes also provide the standard `alloc...` and `init...` methods to create objects that require you to manage their deallocation.)

Each object returned—`aChar`, `anInt`, `aFloat`, and `aDouble`—may belong to a different private subclass (and in fact does). Although each object’s class membership is hidden, its interface is public, being the interface declared by the abstract superclass, `NSNumber`. Although it is not precisely correct, it’s convenient to consider the `aChar`, `anInt`, `aFloat`, and `aDouble` objects to be instances of the `NSNumber` class, since they’re created by `NSNumber` class methods and accessed through instance method declared by `NSNumber`.

Class Clusters With Multiple Public Superclasses

In the example above, one abstract public class declares the interface for multiple private subclasses. This is a class cluster in the purest sense. It’s also possible, and often desirable, to have two (or possibly more) abstract public classes that declare the interface for the cluster. This is evident in the Foundation Framework, which includes these clusters:

Class Cluster	Public Superclasses
NSData	NSData
	NSMutableData
NSArray	NSArray
	NSMutableArray
NSDictionary	NSDictionary
	NSMutableDictionary
NSString	NSString
	NSMutableString

Other clusters of this type also exist, but these clearly illustrate how two abstract nodes cooperate in declaring the programmatic interface to a class cluster. In each of these clusters, one public node declares methods that all cluster objects can respond to, and the other node declares methods that are only appropriate for cluster objects that allow their contents to be modified.

This factoring of the cluster’s interface helps make an object-oriented framework’s programmatic interface more expressive. For example, imagine a `Book` object that declares this method:

```
- (NSString *)title;
```

The book object could return its own instance variable or create a new string object and return that—it doesn’t matter. It’s clear from this declaration that the returned string can’t be modified. Any attempt to modify the returned object will elicit a compiler warning.

Creating Subclasses Within a Class Cluster

The class cluster architecture involves a trade-off between simplicity and extensibility: Having a few public classes stand in for a multitude of private ones makes it easier to learn and use the classes in a framework but somewhat harder to create subclasses within any of the clusters. However, if it's rarely necessary to create a subclass, then the cluster architecture is clearly beneficial. Clusters are used in the Foundation Framework in just these situations.

If you find that a cluster doesn't provide the functionality your program needs, then a subclass may be in order. For example, imagine that you want to create an array object whose storage is file-based rather than memory-based, as in the `NSArray` class cluster. Since you are changing the underlying storage mechanism of the class, you'd have to create a subclass.

On the other hand, in some cases it might be sufficient (and easier) to define a class that embeds within it an object from the cluster. Let's say that your program needs to be alerted whenever some data is modified. In this case, creating a simple cover for a data object that the Foundation framework defines may be the best approach. An object of this class could intervene in messages that modify the data, intercepting the messages, acting on them, and then forwarding them to the embedded data object.

In summary, if you need to manage your object's storage, create a true subclass. Otherwise, create a composite object, one that embeds a standard Foundation framework object in an object of your own design. The sections below give more detail on these two approaches.

A True Subclass

A new class that you create within a class cluster must:

- Be a subclass of the cluster's abstract superclass
- Declare its own storage
- Override the superclass's primitive methods (described below)

Since the cluster's abstract superclass is the only publicly visible node in the cluster's hierarchy, the first point is obvious. This implies that the new subclass will inherit the cluster's interface but no instance variables, since the abstract superclass declares none. Thus the second point: The subclass must declare any instance variables it needs. Finally, the subclass must override any method it inherits that directly accesses an object's instance variables. Such methods are called *primitive methods*.

A class's primitive methods form the basis for its interface. For example, take the `NSArray` class, which declares the interface to objects that manage arrays of objects. In concept, an array stores a number of data items, each of which is accessible by index. `NSArray` expresses this abstract notion through its two primitive methods, `count` and `objectAtIndex:`. With these methods as a base, other methods—*derived methods*—can be implemented, for example:

Derived Method	Possible Implementation
<code>lastObject</code>	Find the last object by sending the array object this message: <code>[self objectAtIndex: ([self count] -1)]</code> .

Derived Method	Possible Implementation
<code>containsObject:</code>	Find an object by repeatedly sending the array object an <code>objectAtIndex:</code> message, each time incrementing the index until all objects in the array have been tested.

The division of an interface between primitive and derived methods makes creating subclasses easier. Your subclass must override inherited primitives, but having done so can be sure that all derived methods that it inherits will operate properly.

The primitive-derived distinction applies to the interface of a fully initialized object. The question of how `init...` methods should be handled in a subclass also needs to be addressed.

In general, a cluster's abstract superclass declares a number of `init...` and `+ className` methods. As described in [“Creating Instances”](#) (page 99), the abstract class decides which concrete subclass to instantiate based your choice of `init...` or `+ className` method. You can consider that the abstract class declares these methods for the convenience of the subclass. Since the abstract class has no instance variables, it has no need of initialization methods.

Your subclass should declare its own `init...` (if it needs to initialize its instance variables) and possibly `+ className` methods. It should not rely on any of those that it inherits. To maintain its link in the initialization chain, it should invoke its superclass's designated initializer within its own designated initializer method. (See “The Runtime System” in *The Objective-C 2.0 Programming Language* for a discussion of designated initializers.) Within a class cluster, the designated initializer of the abstract superclass is always `init`.

True Subclasses: An Example

An example will help clarify the foregoing discussion. Let's say that you want to create a subclass of `NSArray`, named `MonthArray`, that returns the name of a month given its index position. However, a `MonthArray` object won't actually store the array of month names as an instance variable. Instead, the method that returns a name given an index position (`objectAtIndex:`) will return constant strings. Thus, only twelve string objects will be allocated, no matter how many `MonthArray` objects exist in an application.

The `MonthArray` class is declared as:

```
#import <foundation/foundation.h>
@interface MonthArray : NSArray
{
}

+ monthArray;
- (unsigned)count;
- (id)objectAtIndex:(unsigned)index;

@end
```

Note that the `MonthArray` class doesn't declare an `init...` method since it has no instance variables to initialize. The `count` and `objectAtIndex:` methods simply cover the inherited primitive methods, as described above.

The implementation of the `MonthArray` class looks like this:

```

#import "MonthArray.h"

@implementation MonthArray

static MonthArray *sharedMonthArray = nil;
static NSString *months[] = { @"January", @"February", @"March",
    @"April", @"May", @"June", @"July", @"August", @"September",
    @"October", @"November", @"December" };

+ monthArray
{
    if (!sharedMonthArray) {
        sharedMonthArray = [[MonthArray alloc] init];
    }
    return sharedMonthArray;
}

- (unsigned)count
{
    return 12;
}

- objectAtIndex:(unsigned)index
{
    if (index >= [self count])
        [NSException raise:NSRangeException format:@"%s: index (%d) beyond bounds (%d)", sel_getName(_cmd), index, [self count] - 1];
    else
        return months[index];
}

@end

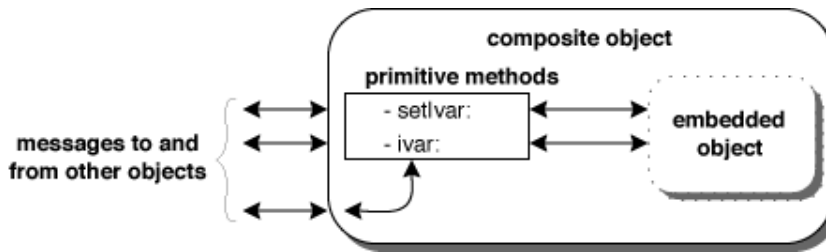
```

Since `MonthArray` overrides the inherited primitive methods, the derived methods that it inherits will work properly without being overridden. `NSArray`'s `lastObject`, `containsObject:`, `sortedArrayUsingSelector:`, `objectEnumerator`, and other methods work without problems for `MonthArray` objects.

A Composite Object

By embedding a private cluster object in an object of your own design, you create a composite object. This composite object can rely on the cluster object for its basic functionality, only intercepting messages that it wants to handle in some particular way. Using this approach reduces the amount of code you must write and lets you take advantage of the tested code provided by the Foundation Framework.

A composite object can be viewed in this way:

Figure 2-13 Embedding a Cluster Object

The composite object must declare itself to be a subclass of the cluster’s abstract node. As a subclass, it must override the superclass’s primitive methods. It can also override derived methods, but this isn’t necessary since the derived methods work through the primitive ones.

Using `NSArray`’s `count` method as an example, the intervening object’s implementation of a method it overrides can be as simple as:

```
- (unsigned)count
{
    return [embeddedObject count];
}
```

However, your object could put code for its own purposes in the implementation of any method it overrides.

A Composite Object: An Example

To illustrate the use of a composite object, imagine you want a mutable array object that tests changes against some validation criteria before allowing any modification to the array’s contents. The example that follows describes a class called `ValidatingArray`, which contains a standard mutable array object. `ValidatingArray` overrides all of the primitive methods declared in its superclasses, `NSArray` and `NSMutableArray`. It also declares the `array`, `validatingArray`, and `init` methods, which can be used to create and initialize an instance:

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

@interface ValidatingArray : NSMutableArray
{
    NSMutableArray *embeddedArray;
}

+ validatingArray;
- init;
- (unsigned)count;
- objectAtIndex:(unsigned)index;
- (void)addObject:object;
- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
- (void)removeLastObject;
- (void)insertObject:object atIndex:(unsigned)index;
- (void)removeObjectAtIndex:(unsigned)index;

@end
```

The implementation file shows how, in a `ValidatingArray`'s `init` method, the embedded object is created and assigned to the *embeddedArray* variable. Messages that simply access the array but don't modify its contents are relayed to the embedded object. Messages that could change the contents are scrutinized (here in pseudo-code) and relayed only if they pass the hypothetical validation test.

```
#import "ValidatingArray.h"

@implementation ValidatingArray

- init
{
    self = [super init];
    if (self) {
        embeddedArray = [[NSMutableArray allocWithZone:[self zone]] init];
    }
    return self;
}

+ validatingArray
{
    return [[[self alloc] init] autorelease];
}

- (unsigned)count
{
    return [embeddedArray count];
}

- objectAtIndex:(unsigned)index
{
    return [embeddedArray objectAtIndex:index];
}

- (void)addObject:object
{
    if (/* modification is valid */) {
        [embeddedArray addObject:object];
    }
}

- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
{
    if (/* modification is valid */) {
        [embeddedArray replaceObjectAtIndex:index withObject:object];
    }
}

- (void)removeLastObject;
{
    if (/* modification is valid */) {
        [embeddedArray removeLastObject];
    }
}

- (void)insertObject:object atIndex:(unsigned)index;
{
    if (/* modification is valid */) {
        [embeddedArray insertObject:object atIndex:index];
    }
}
```

```

}
- (void)removeObjectAtIndex:(unsigned)index;
{
    if (/* modification is valid */) {
        [embeddedArray removeObjectAtIndex:index];
    }
}

```

Creating a Singleton Instance

Some classes of Foundation and the Application Kit create singleton objects: the sole allowable instance of a class in the current process. For example, the `NSFileManager` and the `NSWorkspace` classes instantiate singleton objects for use by a process. When you ask for an instance from these classes, they pass you a reference to the sole instance, allocating and initializing it if it doesn't yet exist.

A singleton object acts as a kind of control center, directing or coordinating the services of the class. Your class should generate a singleton instance rather than multiple instances when there is conceptually only one instance (as with, for example, `NSWorkspace`). You use singleton instances rather than factory methods or functions when it is conceivable that there might be multiple instances one day.

To create a singleton, you need to do a few things in your code:

- Declare a static instance of your singleton object and initialize it to `nil`.
- In your class factory method for the class (named something like “sharedInstance” or “sharedManager”), generate an instance of the class but only if the static instance is `nil`.
- Override the `allocWithZone:` method to ensure that another instance is not allocated if someone tries to allocate and initialize an instance of your class directly instead of using the class factory method.
- Implement the base protocol methods `copyWithZone:`, `release`, `retain`, `retainCount`, and `autorelease` to do the appropriate things to ensure singleton status. (The last four of these methods apply to memory-managed code, not to garbage-collected code.)

Listing 2-15 illustrates how you might implement a singleton:

Listing 2-15 Implementation of a singleton

```

static MyGizmoClass *sharedGizmoManager = nil;

+ (MyGizmoClass*)sharedManager
{
    @synchronized(self) {
        if (sharedGizmoManager == nil) {
            [[self alloc] init]; // assignment not done here
        }
    }
    return sharedGizmoManager;
}

+ (id)allocWithZone:(NSZone *)zone
{

```

```

        @synchronized(self) {
            if (sharedGizmoManager == nil) {
                sharedGizmoManager = [super allocWithZone:zone];
                return sharedGizmoManager; // assignment and return on first
allocation
            }
        }
        return nil; //on subsequent allocation attempts return nil
    }

- (id)copyWithZone:(NSZone *)zone
{
    return self;
}

- (id)retain
{
    return self;
}

- (unsigned)retainCount
{
    return UINT_MAX; //denotes an object that cannot be released
}

- (void)release
{
    //do nothing
}

- (id)autorelease
{
    return self;
}

```

Situations could arise where you want a singleton instance (created and controlled by the class factory method) but also have the ability to create other instances as needed through allocation and initialization. In these cases, you would not override `allocWithZone:` and the other methods following it as shown in [Listing 2-15](#) (page 106).

Adding Behavior to a Cocoa Program

When you develop a program in Objective-C, you won't be on your own. You'll be drawing on the work done by Apple and others, on the classes they've developed and packaged in Objective-C frameworks. These frameworks give you a set of interdependent classes that work together to structure a part—often a substantial part—of your program.

This chapter describes what it's like to write an Objective-C program using a Cocoa framework. It also gives you the basic information you need to know to make a subclass of a framework class.

Starting Up

Using a framework of Objective-C classes and their methods differs from using a library of C functions. In the latter case, you can pretty much pick and choose which functions to use and when to use them, depending on the program you're trying to write. But a framework, on the other hand, imposes a design on your program, or at least on a certain problem space your program is trying to address. With a procedural program, you call library functions as necessary to get the work of the program done. Using an object-oriented framework is similar in that you must invoke methods of the framework to do much of the work of the program. However, you'll also need to customize the framework and adapt it to your needs by implementing methods that the framework will invoke at the appropriate time. These methods are “hooks” that introduce your code into the structure imposed by the framework, augmenting it with the behavior that characterizes your program. In a sense, the usual roles of program and library are reversed. Instead of incorporating library code into your program, you incorporate your program code into the framework.

You can gain some insight on the relationship between custom code and framework by considering what takes place when a Cocoa program begins executing.

What Happens in the main Function

Objective-C programs begin executing where C programs do, in the `main` function. In a complex Objective-C program, the job of `main` is fairly simple. It consists of two steps:

- Set up a core group of objects.
- Turn program control over to those objects.

Objects in the core group might create other objects as the program runs, and those objects might create still other objects. From time to time, the program might also load classes, unarchive instances, connect to remote objects, and find other resources as they're needed. However, all that's required at the outset is enough structure—enough of the object network—to handle the program's initial tasks. The `main` function puts this initial structure in place and gets it ready for the task ahead.

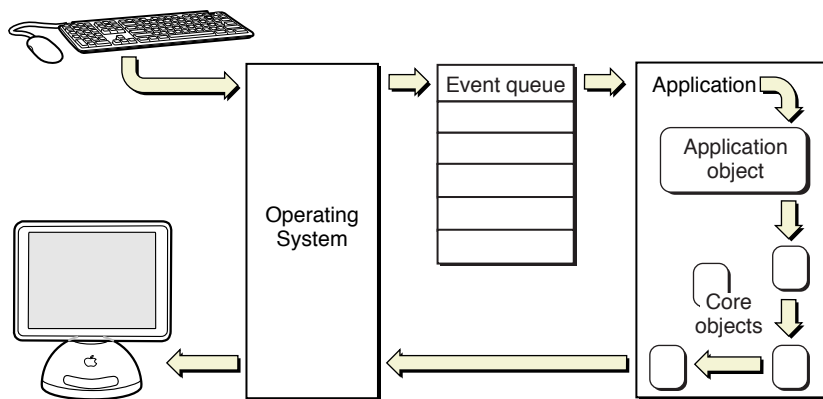
Typically, one of the core objects has the responsibility for overseeing the program or controlling its input. When the core structure is ready, `main` sets this overseer object to work. If the program is a command-line tool or a background server, what this entails might be as simple as passing command-line arguments or opening a remote connection. But for the most common type of Cocoa program, an application, what happens is a bit more involved.

For an application, the core group of objects that `main` sets up must include some objects that draw the user interface. This interface, or at least part of it (such as an application's menu), must appear on the screen when the user launches the application. Once the initial user interface is on the screen, the application is thereafter driven by external events, the most important of which are those originated by users: clicking a button, choosing a menu item, dragging an icon, typing something in a field, and so on. Each such event is reported to the application along with a good deal of information about the circumstances of the user action—for example, which key was pressed, whether the mouse button was pressed or released, where the cursor was located, and which window was affected.

An application gets an event, looks at it, responds to it—often by drawing a part of the user interface—then waits for the next event. It keeps getting events, one after another, as long as the user or some other source (such as a timer) initiates them. From the time it's launched to the time it terminates, almost everything the application does is driven by user actions in the form of events.

The mechanism for getting and responding to events is the main event loop (called “main” because an application can set up subordinate events loops for brief periods.) An event loop is essentially a run loop with one or more input sources attached to it. One object in the core group is responsible for running the main event loop—getting an event, dispatching the event to the object or objects that can best handle it, and then getting the next event. In a Cocoa application, this coordinating object is the global application object. On Mac OS X, this object is an instance of `NSApplication`, and on iPhone OS, it's an instance of `UIApplication`. Figure 3-1 illustrates the main event loop for Cocoa applications on Mac OS X.

Figure 3-1 The main event loop (Mac OS X)



The `main` function in almost all Cocoa applications is extremely simple. In Mac OS X, it consists of only one function call (see Listing 3-1). The `NSApplicationMain` function creates the application object, sets up an autorelease pool, loads the initial user interface from the main nib file, and runs the application, thereby requesting it to begin handling events received on the main event loop.

Listing 3-1 The main function in a Cocoa for Mac OS X application

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

int main(int argc, const char *argv[]) {
    return NSApplicationMain(argc, argv);
}
```

The main function for iPhone OS-based applications calls a similar function: `UIApplicationMain`.

The appendix “[The Core Application Architecture on Mac OS X](#)” (page 201) describes in greater detail the main event loop, the global `NSApplication` instance, and other core application objects in an application running on Mac OS X.

Using a Cocoa Framework

Library functions impose few restrictions on the programs that use them; you can call them whenever you need to. The methods in an object-oriented library or framework, on the other hand, are tied to class definitions and can’t be invoked unless you create or obtain an object that has access to those definitions. Moreover, in most programs the object must be connected to at least one other object so that it can operate in the program network. A class defines a program component; to access its services, you need to craft it into the structure of your application.

That said, some framework classes generate instances that behave much as a set of library functions does. You simply create an instance, initialize it, and either send it a message to accomplish a task, or you insert it into a waiting slot in your application. For example, you can use the `NSFileManager` class to perform various file-system operations, such as moving, copying, and deleting files. If you need to display an alert dialog, you would create an instance of the `NSAlert` class, or with `UIKit`, an instance of the `UIAlertView` class and send it the appropriate message.

In general, however, environments such as Cocoa are more than a grab bag of individual classes that offer their services. They consist of object-oriented frameworks, collections of classes that structure a problem space and present an integrated solution to it. Instead of providing discrete services that you can use as needed (as with function libraries), a framework maps out and implements an entire program structure—or model—that your own code must adapt to. Because this program model is generic, you can specialize it to meet the requirements of your particular program. Rather than design a program that you plug library functions into, you plug your own code into the design provided by the framework.

To use a framework, you must accept the program model it defines and employ and customize as many of its classes as necessary to mold your particular program to that model. The classes are mutually dependent and come as a group, not individually. At first glance, the need to adapt your code to a framework’s program model might seem restrictive. But the reality is quite the opposite. A framework offers you many ways in which you can alter and extend its generic behavior. It simply requires you to accept that all Cocoa programs behave in the same fundamental ways because they are all based on the same program model.

Kinds of Framework Classes

The classes in a Cocoa framework deliver their services in four ways:

- **Off the shelf.** Some classes define off-the-shelf objects, ready to be used. You simply create instances of the class and initialize them as needed. For the Application Kit, subclasses of `NSControl`, such as `NSTextField`, `NSButton`, and `NSTableView`, fall into this category. The corresponding UIKit classes are `UIControl`, `UITextField`, `UIButton`, and `UITableView`. You typically create and initialize off-the-shelf objects using Interface Builder, although you can create and initialize them programmatically.
- **Behind the scenes.** As a program runs, Cocoa creates some framework objects for it “behind the scenes.” You don’t need to explicitly allocate and initialize these objects; it’s done for you. Often the classes are private, but they are necessary to implement the desired behavior.
- **Generically.** Some framework classes are generic. A framework might provide some concrete subclasses of the generic class that you can use unchanged. Yet you can—and *must* in some circumstances—define your own subclasses and override the implementations of certain methods. `NSView` and `NSDocument` are examples of this kind of class in the Application Kit, and `UIView` and `UIScrollView` are UIKit examples.
- **Through delegation and notification.** Many framework objects keep other objects informed of their actions and even delegate certain responsibilities to those other objects. The mechanisms for delivering this information are delegation and notification. A delegating object publishes an interface known as a protocol (see “[Protocols](#)” (page 61) for details). Client objects must first register as delegates and then implement one or more methods of this interface. A notifying object publishes the list of notifications it broadcasts, and any client is free to observe one or more of them. Many framework classes broadcast notifications.

Some classes provide more than one of these general kinds of services. For example, you can drag a ready-made `NSWindow` object from an Interface Builder palette and use it with only minor initializations. Thus the `NSWindow` class provides off-the-shelf instances. But an `NSWindow` object also sends messages to its delegate and posts a variety of notifications. You can even subclass `NSWindow` if, for example, you want to have round windows.

It is the Cocoa classes in the last two categories—generic and delegation/notification—that offer the most possibilities for integrating your program-specific code into the structure provided by the frameworks. “[Inheriting From a Cocoa Class](#)” (page 115) discusses in general terms how you create subclasses of framework classes, particularly generic classes. See “[Communicating With Objects](#)” (page 177) for information on delegation, notification, and other mechanisms for communication between objects in a program network.

Cocoa API Conventions

When you start using the classes, methods, and other API of the Cocoa frameworks, you should be aware of a few conventions that are intended to ensure efficiency and consistency in usage.

- Methods that return objects typically return `nil` to indicate “failure to create” or “no object to return.” They do not return a status code.

The convention of returning `nil` is often used to indicate a runtime error or other non-exceptional condition. The Cocoa frameworks deal with errors such as “array index out of bounds” or “method selector not recognized” by raising an exception (which is handled by a top-level handler) and, if the method signature so requires, returning `nil`.

- Some of these same methods that might return `nil` include a final parameter for returning error information by reference.

This final parameter takes a pointer to an `NSError` object; upon return from a method call that fails (that is, returns `nil`), you can inspect the returned error object to determine the cause of the error or you can display the error to the user in a dialog.

As an example, here’s a method from the `NSDocument` class:

```
- (id)initWithType:(NSString *)typeName error:(NSError **)outError;
```

- In a similar fashion, methods that perform some system operation (such as reading or writing a file) often return a Boolean value to indicate success or failure.

These methods might also include a pointer to an `NSError` object as a final by-reference parameter. For example, there’s this method from the `NSData` class:

```
- (BOOL)writeToFile:(NSString *)path options:(unsigned)writeOptionsMask
error:(NSError **)errorPtr;
```

- Empty container objects are used to indicate a default value or no value—`nil` is usually not a valid object argument.

Many objects encapsulate values or collections of objects, for example, instances of `NSString`, `NSDate`, `NSArray`, and `NSDictionary`. Methods that take these objects as parameters may accept an “empty” object (for example, `@""`) to indicate “no value” or “default value”. For example, the following message sets the represented filename for a window to “no value” by specifying an empty string:

```
[aWindow setRepresentedFilename:@""];
```

Note: The Objective-C construct `@"characters"` creates an `NSString` object containing the literal *characters*. Thus, `@""` would create a string object with no characters—or, an empty string. For more information, see *String Programming Guide for Cocoa*.

- The Cocoa frameworks expect that global string constants rather than string literals are used for dictionary keys, notification and exception names, and some method parameters that take strings.

You should always prefer string constants over string literals when you have a choice. By using string constants, you enlist the help of the compiler to check your spelling and thus avoid runtime errors.

- The Cocoa frameworks use types consistently, giving higher impedance matching across their API sets.

For example, the frameworks use `float` for coordinate values, `CGFloat` for both graphical and coordinate values, `NSPoint` (Application Kit) or `CGPoint` (UIKit) for a location in a coordinate system, `NSString` objects for string values, `NSRange` (Application Kit) for ranges, and `NSInteger` and `NSUInteger` for, respectively, signed and unsigned integral values. When you design your own APIs, you should strive for similar type consistency.

- Unless stated otherwise in the documentation or header files, sizes that are returned in methods such as `frame` and `bounds` (`NSView`) are in points.

A substantial subset of Cocoa API conventions concerns the naming of classes, methods, functions, constants, and other symbols. You should be aware of these conventions when you begin designing your own programmatic interfaces. Some of the more important of these naming conventions are the following:

- Use prefixes for class names and for symbols associated with the class, such as functions and typedef's.

A prefix protects against collisions and helps to differentiate functional areas. The prefix convention is two or three unique uppercase letters, for example, the “AC” in `ACCircle`.

- With API names, it's better to be clear than brief.

For example, it's easy to understand what `removeObjectAtIndex:` does but `remove:` is ambiguous.

- Avoid ambiguous names.

For example, `displayName` is ambiguous because it's unclear whether it displays the name or returns the display name.

- Use verbs in the names of methods or functions that represent actions.

- If a method returns an attribute or computed value, the name of the method is the name of the attribute.

These methods are known as “getter” accessor methods. For example, if the attribute is background color, the getter method should be named `backgroundColor`. Getter methods that return a Boolean value are of a slight variation, using an “is” or “has” prefix—for example, `hasColor`.

- If a method sets the value of an attribute—that is, a “setter” accessor method—it begins with “set” followed by the attribute name.

The first letter of the attribute name is in uppercase—for example, `setBackgroundColor:`.

Note: The implementation of setter and getter methods are discussed in detail in [“Storing and Accessing Properties”](#) (page 126) and in *Model Object Implementation Guide*.

- Do not abbreviate parts of API names unless the abbreviation is well known (for example, HTML or TIFF).

For the complete set of naming guidelines for Objective-C programmatic interfaces, see *Coding Guidelines for Cocoa*.

A general, overarching API convention regards object ownership with memory-managed applications (versus garbage-collected applications). Briefly stated, the convention is that a client owns an object if it creates the object (by allocation then initialization), copies it, or retains it (by sending it `retain`). An owner of an object is responsible for its disposal by sending `release` or `autorelease` to the object when it no longer needs it. For more on this subject, see [“How Memory Management Works”](#) (page 76) and *Memory Management Programming Guide for Cocoa*.

Inheriting From a Cocoa Class

A framework such as Application Kit or UIKit defines a program model that, because it is generic, many different types of applications can share. Since the model is generic, it is not surprising that some framework classes are abstract or intentionally incomplete. A class often does much of its work in low-level and common code, but leaves significant portions of the work either undone or completed in a safe but generic “default” fashion.

An application often needs to create a subclass that fills in these gaps in its superclass, supplying the pieces the framework class is missing. A subclass is the primary way to add application-specific behavior to a framework. An instance of your custom subclass takes its place in the network of objects the framework defines. It inherits the ability to work with other objects from the framework. For example, if you create a subclass of `NSCell` (Application Kit), instances of this new class are able to appear in an `NSMatrix` object, just as `NSButtonCell`, `NSTextFieldCell`, and other framework-defined cell objects can.

When you make a subclass, one of your primary tasks is to implement a specific set of methods declared by a superclass (or in a protocol adopted by a superclass). Re-implementing an inherited method is known as *overriding* that method.

When to Override a Method

Most methods defined in a framework class are fully implemented; they exist so you can invoke them to obtain the services the class provides. You rarely need to override such methods and shouldn’t attempt to. The framework depends on them doing just what they do—nothing more and nothing less. In other cases, you can override a method, but there’s no real reason to do so. The framework’s version of the method does an adequate job. But just as you might implement your own version of a string-comparison function rather than use `strcmp`, you can choose to override the framework method if you wish.

Some framework methods, however, are intended to be overridden; they exist to let you add program-specific behavior to the framework. Often the method, as implemented by the framework, does little or nothing that’s of value to your application, but is invoked in messages initiated by other framework methods. To give content to these kinds of methods, an application must implement its own version.

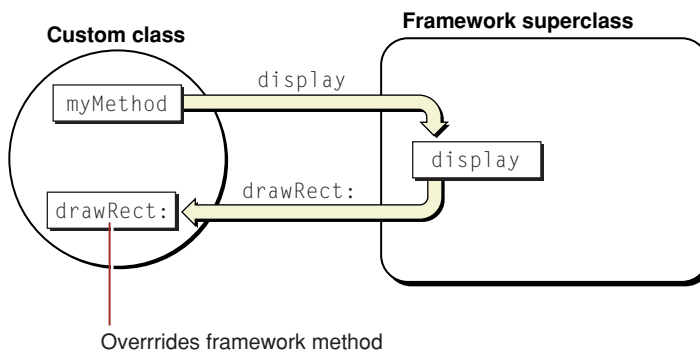
Invoke or Override?

The framework methods you override in a subclass generally won’t be ones that you’ll invoke yourself, at least directly. You simply re-implement the method and leave the rest up to the framework. In fact, the more likely you are to write an application-specific version of a method, the less likely you are to invoke it in your own code. There’s a good reason for this. In a general sense, a framework class declares public methods so that you, the developer, can do one of two things:

- Invoke them to avail yourself of the services the class provide
- Override them to introduce your own code into the program model defined by the framework

Sometimes a method falls into both these categories; it renders a valuable service upon invocation, and it can be strategically overridden. But generally, if a method is one that you can invoke, it's fully defined by the framework and doesn't need to be redefined in your code. If the method is one that you need to re-implement in a subclass, the framework has a particular job for it to do and so will invoke the method itself at the appropriate times. Figure 3-2 illustrates the two general types of framework methods.

Figure 3-2 Invoking a framework method that messages an overridden method



Much of the work of object-oriented programming with a Cocoa framework is implementing methods that your program uses only indirectly, through messages arranged by the framework.

Types of Overridden Methods

You may choose to define several different types of methods in a subclass:

- Some framework methods are fully implemented and are meant to be invoked by other framework methods. In other words, even though you may re-implement these methods, you often don't invoke them elsewhere in your code. They provide some service—data or behavior—required by some other code at some point during program execution. These methods exist in the public interface for just one reason—so that you can override them if you want to. They give you an opportunity either to substitute your own algorithm for the one used by the framework or to modify or extend the framework algorithm.

An example of this type of method is `trackWithEvent:`, defined in the `NSMenuView` class of the Application Kit. `NSMenuView` implements this method to satisfy the immediate requirement—handling menu tracking and item selection—but you may override it if you want different behavior.

- Another type of method is one that makes an object-specific decision, such as whether an attribute is turned on or whether a certain policy is in effect. The framework implements a default version of this method that makes the decision one way, and you must implement your own version if you want a different decision. In most cases, implementation is simply a matter of returning YES or NO, or of calculating a value other than the default.

The `acceptsFirstResponder` method of `NSResponder` is typical of this kind. Views are sent `acceptsFirstResponder` messages asking, among other things, if they respond to key strokes or mouse clicks. By default, `NSView` objects return NO for this method—most views don't accept typed input. But some do, and they should override `acceptsFirstResponder` to return YES. A UIKit example is the `layerClass` class method of `UIView`; if you do not want the default layer class for the view, you can override the method and substitute your own.

- Some methods must be overridden, but only to add something, not to replace what the framework implementation of the method does. The subclass version of the method augments the behavior of the superclass version. When your program implements one of these methods, it's important that it incorporate the very method it overrides. It does this by sending a message to `super` (the superclass), invoking the framework-defined version of the method.

Often this kind of method is one that every class in a chain of inheritance is expected to contribute to. For example, objects that archive themselves must conform to the `NSCoding` protocol and implement the `initWithCoder:` and `encodeWithCoder:` methods. But before a class performs the encoding or decoding that is specific to its instance variables, it must invoke the superclass version of the method.

Sometimes a subclass version of a method wants to “reuse” the superclass behavior and then add something, however small, to the final result. In the `NSView drawRect:` and `UIView drawRect:` methods, for example, a subclass of a view class that performs some complicated drawing might want to draw a border around the drawing, so it would invoke `super` first.

- Some framework methods do nothing at all or merely return some trivial default value (such as `self`) to prevent runtime or compile-time errors. These methods are meant to be overridden. The framework cannot define these methods even in rudimentary form because they carry out tasks that are entirely program-specific. There's no need to incorporate the framework implementation of the method with a message to `super`.

Most methods that a subclass overrides belong in this group. For example, the `NSDocument` methods `dataOfType:error:` and `readFromData:ofType:error:` (among others) must be overridden when you create a document-based application.

Overriding a method does not have to be a formidable task. You can often make significant change in superclass behavior by a careful re-implementation of the method that entails no more than one or two lines of code. And you are not entirely on your own when you implement your own version of a method. You can draw on the classes, methods, functions, and types already provided by the Cocoa frameworks.

When to Make a Subclass

Just as important as knowing which methods of a class to override—and indeed preceding that decision—is identifying those classes to inherit from. Sometimes these decisions can be obvious, and sometimes they can be far from simple. A few design considerations can guide your choices.

First, know the framework. You should become familiar with the purpose and capabilities of each framework class. Maybe there is a class that already does what you want to do. And if you find a class that does *almost* what you want done, you're in luck. That class is a promising superclass for your custom class. Subclassing is a process of reusing an existing class and specializing it for your needs. Sometimes all a subclass needs to do is override a single inherited method and have the method do something slightly different from the original behavior. Other subclasses might add one or two attributes to their superclass (as instance variables), and then define the methods that access and operate on these attributes, integrating them into the superclass behavior.

There are other considerations that can help you decide where your subclass best fits into the class hierarchy. What is the nature of the application, or of the part of the application you're trying to craft? Some Cocoa architectures impose their own subclassing requirements. For example, if yours is a multiple-document application on Mac OS X, the document-based architecture defined by the

Application Kit requires you to subclass `NSDocument` and perhaps other classes as well. To make your Mac OS X application scriptable (that is, responsive to AppleScript commands), you might have to subclass one of the scripting classes of the Application Kit, such as `NSScriptCommand`.

Another factor is the role that instances of the subclass will play in the application. The Model-View-Control design pattern, a major one in Cocoa, assigns roles to objects: they're view objects that appear on the user interface; model objects holding application data (and the algorithms that act on that data); or controller objects, which mediate between view and model objects. (For details, see [“The Model-View-Controller Design Pattern”](#) (page 161).) Knowing what role an object plays can narrow the decision for which superclass to use. If instances of your class are view objects that do their own custom drawing and event-handling, your class should probably inherit from `NSView` (if you are using the Application Kit) or from `UIView` (if you are using UIKit). If your application needs a controller object, you can either use one of the off-the-shelf controller classes of the Application Kit (such as `NSObjectController`) or, if you want a different behavior, subclass `NSController` or `NSObject`. If your class is a typical model class—say, one whose objects represent rows of corporate data in a spreadsheet—you probably should subclass `NSObject` or use the Core Data framework.

iPhone OS Note: The controller classes of the Application Kit, including `NSController`, have no counterparts in UIKit. There are controller classes in UIKit—`UIViewController`, `UITableViewController`, `UINavigationController`, and `UITabBarController`—but these classes are based on a different design and have a different purpose—application navigation and mode selection (see [“View Controllers in UIKit”](#) (page 154) for a summary). In addition, iPhone OS does not include the Core Data framework.

Yet subclassing is sometimes not the best way to solve a problem. There may be a better approach you could take. If you just want to add a few convenience methods to a class, you might create a category instead of a subclass. Or you could employ one of the many other design-pattern based resources of the Cocoa development “tool box,” such as delegation, notification, and target-action (described in [“Communicating With Objects”](#) (page 177)). When deciding on a candidate superclass, scan the header file of the class (or the reference documentation) to see if there is a delegation method, a notification, or some other mechanism that will enable you to do what you want without subclassing.

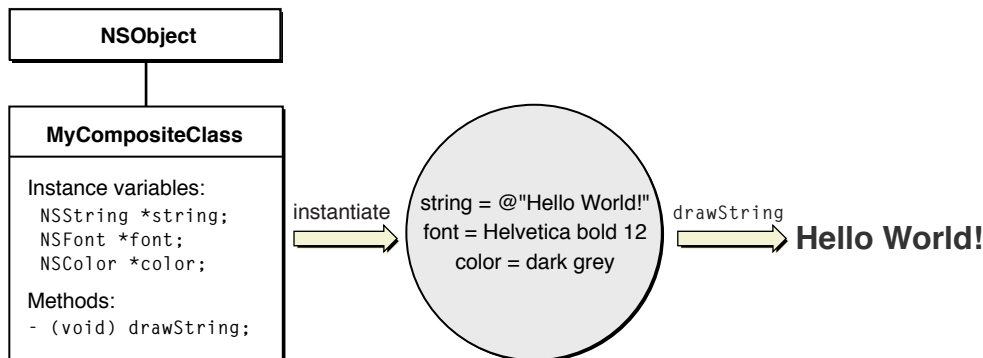
In a similar vein, you can also examine the header files or documentation for the framework's protocols. By adopting a protocol, you might be able to accomplish your goal while avoiding the difficulties of a complex subclass. For example, if you want to manage the enabled states of menu items, you can adopt the `NSMenuValidation` protocol in a custom controller class; you don't have to subclass `NSMenuItem` or `NSMenu` to get this behavior.

Just as some framework methods are not intended to be overridden, some framework classes (such as `NSFileManager` in the Application Kit and, in UIKit, `UIWebView`) are not intended to be subclassed. If you do attempt such a subclass, you should proceed with caution. The implementations of certain framework classes are complicated and tightly integrated into the implementations of other classes and even different parts of the operating system. Often it is difficult to duplicate correctly what a framework method does or to anticipate interdependencies or effects the method might have. Changes that you make in some method implementations could have far-reaching, unforeseen, and unpleasant consequences.

In some cases, you can get around these difficulties by using object composition, a general technique of assembling objects in a “host” object, which manages them to get complex and highly customized behavior (see Figure 3-3). Instead of inheriting directly from a complex framework superclass, you might create a custom class that holds an instance of that superclass as an instance variable. The custom class itself could be fairly simple, perhaps inheriting directly from the root class, `NSObject`; although simple in terms of inheritance, the class manipulates, extends, and augments the embedded

instance. To client objects it can appear in some respects to be a subclass of the complex superclass, although it probably won't share the interface of the superclass. The Foundation framework class `NSAttributedString` gives an example of object composition. `NSAttributedString` holds an `NSString` object as an instance variable, and exposes it through the `string` method. `NSString` is a class with complex behaviors, including string encoding, string searching, and path manipulation. `NSAttributedString` augments these behaviors with the capability for attaching attributes such as font, color, alignment, and paragraph style to a range of characters. And it does so without subclassing `NSString`.

Figure 3-3 Object composition



Sometimes what seems to be the most obvious candidate for your superclass is not the best choice. As you might know, `NSView` and `UIView` objects are what Cocoa uses for drawing in most cases. But if you are designing a drawing or CAD program with potentially hundreds or thousands of graphic elements, you should think about designing and using your own custom graphic-element classes that do not inherit from the base view classes. In terms of object size, both `NSView` and `UIView` objects carry around a lot of instance data. The graphic-element instances of your custom classes can be "lightweight" and yet contain all the information a single view object needs to draw them.

Basic Subclass Design

All subclasses, regardless of ancestor or role, share certain characteristics when they are well designed. A poorly designed subclass is susceptible to error, hard to use, difficult to extend, and a drag on performance. A well-designed subclass is quite the opposite. This article offers suggestions for designing subclasses that are efficient, robust, and both usable and reusable.

Further Reading: Although this document describes some of the mechanics of crafting a subclass, it focuses primarily on basic design. It does not describe how you can use the development applications Xcode and Interface Builder to automate part of class definition. To start learning about these applications, read *Xcode Quick Tour Guide for Mac OS X*. The design information presented in this section is particularly applicable to model objects. *Model Object Implementation Guide* discusses the proper design and implementation of model objects at length.

The Form of a Subclass Definition

An Objective-C class consists of an interface and an implementation. By convention, these two parts of a class definition go in separate files. The interface file (as with ANSI C header files) has an extension of `.h`; the implementation file has an extension of `.m`. The name of the file up to the extension is typically the name of the class. Thus for a class named `Controller`, the files would be:

```
Controller.h
Controller.m
```

The interface file contains a list of property, method, and function declarations that establish the public interface of the class. It also holds declarations of instance variables, constants, string globals, and other data types. The directive `@interface` introduces the essential declarations of the interface, and the `@end` directive terminates the declarations. The `@interface` directive is particularly important because it identifies the name of the class and the class directly inherited from, using the form:

```
@interface ClassName : Superclass
```

Listing 3-2 gives an example of the interface file for the hypothetical `Controller` class, before any declarations are made.

Listing 3-2 The basic structure of an interface file

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

```
@interface Controller : NSObject {
}
```

```
@end
```

To complete the class interface, you must make the necessary declarations in the appropriate places in this interface structure, as shown in Figure 3-4.

Figure 3-4 Where to put declarations in the interface file

Additional imports	—	<code>#import <Foundation/Foundation.h></code>
Function and data type declarations	—	<code>@interface Controller : NSObject {</code>
Instance variables	—	<code>}</code>
Method and property declarations	—	<code>@end</code>

See “[Instance Variables](#)” (page 122) and “[Functions, Constants, and Other C Types](#)” (page 135) for more information about these types of declarations.

You begin the class interface by importing the appropriate Cocoa frameworks as well as the header files for any types that appear in the interface. The `#import` preprocessor command is similar to `#include` in that it incorporates the specified header file. However, `#import` improves on efficiency by including the file only if it was not previously included, directly or indirectly, for compilation. The angle brackets (`<...>`) following the command identify the framework the header file is from and, after a slash character, the header file itself. Thus the syntax for an `#import` line takes the form:

```
#import <Framework/File.h>
```

The framework must be in one of the standard system locations for frameworks. In the example in [Listing 3-2](#) (page 120), the class interface file imports the file `Foundation.h` from the Foundation framework. As in this case, the Cocoa convention is that the header file having the same name as the framework contains a list of `#import` commands that include all the public interfaces (and other public headers) of the framework. Incidentally, if the class you’re defining is part of an application, all you need to do is import the `Cocoa.h` file of the Cocoa umbrella framework.

If you must import a header file that is part of your project, use quotation marks rather than angle brackets as delimiters; for example:

```
#import "MyDataTypes.h"
```

The class implementation file has a simpler structure, as shown by [Listing 3-3](#). It is important that the class implementation file begin by importing the class interface file.

Listing 3-3 The basic structure of an implementation file

```
#import "Controller.h"
```

```
@implementation Controller
```

```
@end
```

You must write all method and property implementations between the `@implementation` and `@end` directives. The implementations of functions related to the class can go anywhere in the file, although by convention they are also put between the two directives. Declarations of private types (functions, structures, and so on) typically go between the `#import` commands and the `@implementation` directive.

Overriding Superclass Methods

Custom classes, by definition, modify the behavior of their superclass in some program-specific way. Overriding superclass methods is usually the way to effect this change. When you design a subclass, an essential step is identifying the methods you are going to override and thinking about how you are going to re-implement them.

Although [“When to Override a Method”](#) (page 115) offers some general guidelines, identifying which superclass methods to override requires you to investigate the class. Peruse the header files and documentation of the class. Also locate the designated initializer of the superclass because, for initialization to succeed, you must invoke the superclass version of this method in your class’s designated initializer. (See [“Object Creation”](#) (page 80) for details.)

Once you have identified the methods, you can start—from a purely practical standpoint—by copying the declarations of the methods to be overridden into your interface file. Then copy those same declarations into your `.m` file as skeletal method implementations by replacing the terminating semicolon with enclosing braces.

Remember, as discussed in [“When to Override a Method”](#) (page 115), a Cocoa framework (and not your own code) usually invokes the framework methods that you override. In some situations you need to let the framework know that it should invoke your overridden version of a method and not the original method. Cocoa gives you various ways to accomplish this. The Interface Builder application, for example, allows you to substitute your class for a (compatible) framework class in its Info (inspector) window. If you create a custom `NSCell` class, you can associate it with a particular control using the `setCell:` method of the `NSControl` class.

Instance Variables

The reason for creating a custom class, aside from modifying the behavior of the superclass, is to add properties to it. (The word “properties” here is intended in a general sense, indicating both the attributes of an instance of the subclass and the relationships that instance has to other objects.) Given a hypothetical `Circle` class, if you are going to make a subclass that adds color to the shape, the subclass must somehow carry the attribute of color. To accomplish this, you might add a `color` instance variable (most likely typed as an `NSColor` object for a Mac OS X application, or a `UIColor` object for an iPhone application) to the class interface. Instances of your custom class encapsulate the new attribute, holding it as a persistent, characterizing piece of data.

Instance variables in Objective-C are declarations of objects, structures, and other data types that are part of the class definition. If they are objects, the declaration can either dynamically type (using `id`) or statically type the object. The following example shows both styles:

```
id delegate;
NSColor *color;
```

Generally, you dynamically type an object instance variable when the class membership of the object is indeterminate or unimportant. Objects held as instance variables should be created, copied, or explicitly retained—unless they are already retained by a parent object (as is the case with delegates). If an instance is unarchived, it should retain its object instance variables as it decodes and assigns them in `initWithCoder:`. (For more on object archiving and unarchiving, see [“Entry and Exit Points”](#) (page 123).)

The convention for naming instance variables is to make them a lowercase string containing no punctuation marks or special characters. If the name contains multiple words, run them together, but make the first letter of the second and each succeeding word uppercase. For example:

```
NSString *title;
UIColor *backgroundColor;
NSRange currentSelectedRange;
```

When the tag `IBOutlet` precedes the declarations of instance variables, it identifies an outlet with (presumably) a connection archived in a nib file. The tag also enables Interface Builder and Xcode to synchronize their activities. When outlets are unarchived from an Interface Builder nib file, the connections are automatically reestablished.

Instance variables can hold more than an object's attributes, vendable to clients of the object. Sometimes instance variables can hold private data that is used as the basis for some task performed by the object; a backing store or a cache are examples. (If data is not per-instance, but is to be shared among instances of a class, use global variables instead of instance variables.)

When you add instance variables to a subclass, it's worthwhile to heed a few guidelines:

- Add only instance variables that are absolutely necessary. The more instance variables you add, the more the instance size swells. And the more instances of your class that are created, the more of a concern this becomes. If it's possible, compute a critical value from existing instance variables rather than add another instance variable to hold that value.
- For the same reason of economy, try to represent the instance data of the class efficiently. For example, if you want to specify a number of flags as instance variables, use a bit field instead of a series of Boolean declarations. (Be aware, however, that there are archiving complications with bit fields). You might use an `NSDictionary` object to consolidate a number of related attributes as key-value pairs; if you do this, make sure that the keys are adequately documented.
- Give the proper scope to your instance variables. Never scope a variable as `@public` as this violates the principle of encapsulation. Use `@protected` for instance variables if you know which are the likely subclasses of your class (such as the classes of your application) and you require efficient access of data. Otherwise, `@private` is a reasonable choice for the high degree of implementation hiding it provides. This implementation hiding is especially important for classes exposed by a framework and used by applications or other frameworks; it permits the modification of the class implementation without the need to recompile all clients.
- Make sure there are declared properties or accessor methods for those instance variables that are essential attributes or relationships of the class. Accessor methods and declared properties preserve encapsulation by enabling the values of instance variables to be set and retrieved. See [“Storing and Accessing Properties”](#) (page 126) for more information on this subject.
- If you intend your subclass to be public—that is, you anticipate that others might subclass it—pad the end of the instance variable list with a reserved field, usually typed as `id`. This reserved field helps to ensure binary compatibility if, at any time in the future, you need to add another instance variable to the class. See [“When the Class Is Public \(Mac OS X\)”](#) (page 136) for more information on preparing public classes.

Entry and Exit Points

The Cocoa frameworks send an object messages at various points in the object's life. Almost all objects (including classes, which are really objects themselves) receive certain messages at the beginning of their runtime life and just before their destruction. The methods invoked by these messages (if implemented) are “hooks” that allow the object to perform tasks relevant to the moment. These methods are (in the order of invocation):

1. `initialize`—This class method enables a class to initialize itself before it or its instances receive any other messages. Superclasses receive this message before subclasses. Classes that use the old archiving mechanism can implement `initialize` to set their class version. Other possible uses of

`initialize` are registering a class for some service and initializing some global state that is used by all instances. However, sometimes it's better to do these kinds of tasks lazily (that is, the first time they're needed) in an instance method rather than in `initialize`.

2. `init` (or other initializer)—You must implement `init` or some other primary initializer to initialize the state of the instance unless the work done by the superclass designated initializer is sufficient for your class. See “Object Creation” (page 80) for information on initializers, including designated initializers.
3. `initWithCoder:`—If you expect objects of your class to be archived—for example, yours is a model class—you should adopt (if necessary) the `NSCoding` protocol and implement its two methods, `initWithCoder:` and `encodeWithCoder:`. In these methods you must decode and encode, respectively, the instance variables of the object in a form suitable for an archive. You can use the decoding and encoding methods of `NSCoder` for this purpose or the key-archiving facilities of `NSKeyedArchiver` and `NSKeyedUnarchiver`. When the object is being unarchived instead of being explicitly created, the `initWithCoder:` method is invoked instead of the initializer. In this method, after decoding the instance-variable values, you assign them to the appropriate variables, retaining or copying them if necessary. For more on this subject, see *Archives and Serializations Programming Guide for Cocoa*.
4. `awakeFromNib`—When an application loads a nib file, an `awakeFromNib` message is sent to each object loaded from the archive, but only if it can respond to the message, and only after all the objects in the archive have been loaded and initialized. When an object receives an `awakeFromNib` message, it's guaranteed to have all its outlet instance variables set. Typically, the object that owns the nib file (File's Owner) implements `awakeFromNib` to perform programmatic initializations that require outlet and target-action connections to be set.
5. `encodeWithCoder:`—Implement this method if instances of your class should be archived. This method is invoked just prior to the destruction of the object. See the description of `initWithCoder:` above.
6. `dealloc` or `finalize`—In memory-managed programs, implement the `dealloc` method to release instance variables and deallocate any other memory claimed by an instance of your class. The instance is destroyed soon after this method returns. In garbage-collected code, you may need to implement the `finalize` method instead. See “The `dealloc` and `finalize` Methods” (page 88) for further information.

An application's singleton application object (stored by the Application Kit in the global variable `NSApp`) also sends messages to an object—if it's the application object's delegate and implements the appropriate methods—at the beginning and end of the application's life. Just after the application launches, the application object sends its delegate the following messages, depending on the framework and platform:

- In the Application Kit, the application object sends `applicationWillFinishLaunching:` and `applicationDidFinishLaunching:` to the delegate. The former is sent just before any double-clicked document is opened, the latter just afterwards.
- In UIKit, the application object sends `applicationDidFinishLaunching:` to the delegate.

In these methods the delegate can restore application state and specify global application logic that needs to happen just once early in the application's runtime existence. Just before it terminates, the application object for both frameworks sends `applicationWillTerminate:` to its delegate, which can implement the method to gracefully handle program termination by saving documents and (particularly on iPhone OS) application state.

The Cocoa frameworks give you numerous other hooks for events ranging from window closing to application activation and deactivation. These are usually implemented as delegation messages and require your object to be the delegate of the framework object and to implement the required method. Sometimes the hooks are notifications. (See [“Communicating With Objects”](#) (page 177) for more on delegation and notification.)

Initialize or Decode?

If you want objects of your class to be archived and unarchived, the class must conform to the `NSCoding` protocol; it must implement the methods that encode its objects (`encodeWithCoder:`) and decode them (`initWithCoder:`). Instead of an initializer method or methods, the `initWithCoder:` method is invoked to initialize an object being created from an archive.

Because the class's initializer method and `initWithCoder:` might be doing much of the same work, it makes sense to centralize that common work in a helper method called by both the initializer and `initWithCoder:`. For example, if as part of its set-up routine, an object specifies drag types and dragging source, it might implement something such as shown in Listing 3-4.

Listing 3-4 Initialization helper method

```
(id)initWithFrame:(NSRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        [self setTitleColor:[NSColor lightGrayColor]];
        [self registerForDragging];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)aCoder {
    self = [super initWithCoder:aCoder];
    titleColor = [[aCoder decodeObject] copy];
    [self registerForDragging];
    return self;
}

- (void)registerForDragging {
    [theView registerForDraggedTypes:
        [NSArray arrayWithObjects:DragDropSimplePboardType, NSStringPboardType,
        NSFileNamesPboardType, nil]];
    [theView setDraggingSourceOperationMask:NSDragOperationEvery forLocal:YES];
}
```

An exception to the parallel roles of a class initializer and `initWithCoder:` is when you create a custom subclass of a framework class whose instances appear on an Interface Builder palette. The general procedure is to define the class in your project, drag the object from the Interface Builder palette into your interface, and then associate that object with your custom subclass using the Custom Class pane of the Interface Builder Info window. However, in this case the `initWithCoder:` method

for the subclass is not invoked during unarchiving; an `init` message is sent instead. You should perform any special set-up tasks for your custom object in `awakeFromNib`, which is invoked after all objects in a nib file have been unarchived.

Storing and Accessing Properties

The word “property” has both a general and a language-specific meaning in Cocoa. The general notion comes from the design pattern of object modeling. The language-specific meaning relates to the language feature of declared properties.

As described in “[Object Modeling](#)” (page 169), a property is a defining characteristic of an object (called in the object modeling pattern an entity) and is part of its encapsulated data. A property can be of two sorts: an attribute, such as title and location, or a relationship. Relationships can be to-one or to-many, and can take many forms, such as an outlet, a delegate, or a collection of other objects. Properties are usually, but not necessarily, stored as instance variables.

Part of the overall strategy for a well-designed class is to enforce encapsulation. Client objects should not be able to directly access properties where they are stored, but instead access them through the interface of the class. The methods that grant access to the properties of an object are called accessor methods. Accessor methods (or, simply, “accessors”) get and set the values of an object’s properties. They function as the gates to those properties, regulating access to them and enforcing the encapsulation of the object’s instance data. The accessor method that gets a value is, by convention, called a “getter” method and the method that sets a property’s value is called a “setter” method.

The declared properties feature (introduced in Objective-C 2.0) has brought changes, some subtle, to how you control access to properties. A declared property is a syntactical shorthand for declaring the getter and setter methods of a class’s property. In the implementation block you can then request the compiler to synthesize those methods. As summarized in “[Declared Properties](#)” (page 63), you declare a property, using the `@property` directive, in the `@interface` block of a class definition. The declaration specifies the name and type of the property and may include qualifiers that tell the compiler *how* to implement the accessor methods.

When you design a class, several factors affect how you might store and access the properties of the class. You may choose to use the declared properties feature and have the compiler synthesize accessor methods for you. Or you may elect to implement the class’s accessor methods yourself; this is an option even if you do use declared properties. Another important factor is whether your code operates in a garbage-collected environment. If the garbage collector is enabled, the implementation of accessor methods is much simpler than it is in memory-managed code. The following sections describe the various alternatives for storing and accessing properties.

Taking Advantage of Declared Properties

Unless you have a compelling reason otherwise, it is recommended that you use the declared properties feature of Objective-C 2.0 in new projects rather than manually writing all of your own accessor methods. First, declared properties relieve you of the tedious burden of accessor-method implementation. Second, by having the compiler synthesize accessor methods for you, you reduce the possibility of programming errors and ensure more consistent behavior in your class implementation. And finally, the declarative style of declared properties makes your intent clear to other developers.

You might declare some properties with `@property` but still need to implement your own accessor methods for them. The typical reason for doing this is wanting the method to do something more than simply get or set the value of the property. For example, the resource for an attribute has to be loaded from the file system—say, a large image file—and for performance reasons you want your getter method to load the resource lazily (that is, the first time it is requested). If you have to implement an accessor method for such a reason, see [“Implementing Accessor Methods”](#) (page 128) for tips and guidelines. You tell the compiler not to synthesize accessor methods for a property either by designating that property with a `@dynamic` directive in the class’s `@implementation` block or simply by not specifying a `@synthesize` directive for the property (`@dynamic` is the default); the compiler refrains from synthesizing an accessor method if it sees that such a method already exists in the implementation and there is no `@synthesize` directive for it.

If garbage collection is disabled for your program, the attribute you use to qualify your property declarations is important. By default (the `assign` attribute), a synthesized setter method performs simple assignment, which is appropriate for garbage-collected code. But in memory-managed code simple assignment is not appropriate for properties with object values. You must either retain or copy the assigned object in your setter method; you tell the compiler to do this when you qualify a property declaration with a `retain` or `copy` attribute. Note that you should *always* specify the `copy` attribute for an object property whose class conforms to the `NSCopying` protocol, even if garbage collection is enabled.

The following examples illustrate a variety of behaviors you can achieve with property declarations (with different sets of qualifying attributes) in the `@interface` block and `@synthesize` and `@dynamic` directives in the `@implementation` block. The following code instructs the compiler to synthesize accessor methods for the declared `name` and `accountID` properties; because the `accountID` property is read-only, the compiler synthesizes only a getter method. In this case, garbage collection is assumed to be enabled.

```
@interface MyClass : NSObject {
    NSString *name;
    NSNumber *accountID;
}
@property (copy) NSString *name;
@property (readonly) NSNumber *accountID;
// ...
@end
@implementation MyClass
@synthesize name, accountID;
// ...
@end
```

Note: As shown in this and subsequent code examples, a property in a 32-bit process must be backed by an instance variable. In a program with a 64-bit address space, properties do not have to be backed by instance variables.

The following code listing illustrates another aspect of declared properties. The assumption is that garbage collection is not enabled, so the declaration for the `currentHost` property has a `retain` attribute, which instructs the compiler to retain the new value in its synthesized getter method. In addition, the declaration for the `hidden` property includes a qualifying attribute that instructs the compiler to synthesize a getter method named `isHidden`. (Because the value of this property is a non-object value, simple assignment is sufficient in the setter method.)

```
@interface MyClass : NSObject {
    NSHost *currentHost;
```

```

        Boolean *hidden;
    }
    @property (retain, nonatomic) NSHost *currentHost;
    @property (getter=isHidden, nonatomic) Boolean *hidden;
    // ...
@end
@implementation MyClass
@synthesize name, hidden;
// ...
@end

```

Note: Note that this example specifies the `nonatomic` attribute for the properties, a change from the default `atomic`. When a property is `atomic`, its synthesized getter method is guaranteed to return a valid value, even when other threads are executing simultaneously. See the discussion of properties in *The Objective-C 2.0 Programming Language* for a discussion of atomic versus nonatomic properties, especially with regard to performance. For more on thread-safety issues, see *Threading Programming Guide*.

The property declaration in the following example tells the compiler that the `previewImage` property is read-only so that it won't expect to find a setter method. By using the `@dynamic` directive in the `@implementation` block, it also instructs the compiler not to synthesize the getter method, and then provides an implementation of that method.

```

@interface MyClass : NSObject {
    NSImage *previewImage;
}
@property (readonly) NSImage *previewImage;
// ...
@end
@implementation MyClass
@dynamic previewImage;
// ...
- (NSImage *)previewImage {
    if (previewImage == nil) {
        // lazily load image and assign to ivar
    }
    return previewImage;
}
// ...
@end

```

Further Reading: Read “Properties” in *The Objective-C 2.0 Programming Language* for the definitive description of declared properties and for guidelines on using them in your code.

Implementing Accessor Methods

For reasons of convention—and because it enables classes to be compliant with key-value coding (see “[Key-Value Mechanisms](#)” (page 131))—the names of accessor methods must have a specific form. For a method that returns the value of an instance variable (sometimes called the *getter*), the name of the method is simply the name of the instance variable. For a method that sets the value of an instance variable (the *setter*), the name begins with “set” and is immediately followed by the name of the instance variable (with the first letter uppercase). For example, if you had an instance variable named “color,” the declarations of the getter and setter accessors would be:

```
- (NSColor *)color;
- (void)setColor:(NSColor *)aColor;
```

With Boolean attributes a variation of the form for getter methods is acceptable. The syntax in this case would take the form `isAttribute`. For example, a Boolean instance variable named `hidden` would have a `isHidden` getter method and a `setHidden:` setter method.

If an instance variable is a scalar C type, such as `int` or `float`, the implementations of accessor methods tend to be very simple. Assuming an instance variable named `currentRate`, of type `float`, [Listing 3-5](#) (page 129) shows how to implement the accessor methods.

Listing 3-5 Implementing accessors for a scalar instance variable

```
- (float)currentRate {
    return currentRate;
}

- (void)setCurrentRate:(float)newRate {
    currentRate = newRate;
}
```

If garbage collection is enabled, implementation of the setter accessor method for properties with object values is also a matter of simple assignment. Listing 3-6 shows how you would implement such methods.

Listing 3-6 Implementing accessors for an object instance variable (garbage collection enabled)

```
- (NSString *)jobTitle {
    return jobTitle;
}

- (void)setJobTitle:(NSString *)newTitle {
    jobTitle = newTitle;
}
```

When instance variables hold objects and garbage collection is not enabled, the situation becomes more nuanced. Because they are instance variables, these objects must be persistent and so must be created, copied, or retained when they are assigned. When a setter accessor changes the value of an instance variable, it is responsible not only for ensuring persistence but for properly disposing of the old value. The getter accessor vends the value of the instance variable to the object requesting it. The operations of both types of accessors have implications for memory management given two assumptions from the Cocoa object-ownership policy:

- Objects returned from methods (such as from getter accessors) are valid in the scope of the invoking object. In other words, it is guaranteed that the object will not be released or change value within that scope (unless documented otherwise).
- When an invoking object receives an object from a method such as an accessor, it should not release the object unless it has explicitly retained (or copied) it first.

With these two assumptions in mind, let's look at two possible implementations of a getter and setter accessor for an `NSString` instance variable named `title`. Listing 3-7 shows the first.

Listing 3-7 Implementing accessors for an object instance variable—good technique

```
- (NSString *)title {
```

```

    return title;
}
- (void)setTitle:(NSString *)newTitle {
    if (title != newTitle) {
        [title autorelease];
        title = [newTitle copy];
    }
}

```

Note that the getter accessor simply returns a reference to the instance variable. The setter accessor, on the other hand, is busier. After verifying that the passed-in value for the instance variable isn't the same as the current value, it autoreleases the current value before copying the new value to the instance variable. (Sending `autorelease` to the object is “more thread-safe” than sending it `release`.) However, there is still a potential danger with this approach. What if a client is using the object returned by the getter accessor and meanwhile the setter accessor autoreleases the old `NSString` object and then, soon after, that object is released and destroyed? The client object's reference to the instance variable would no longer be valid.

Listing 3-8 shows another implementation of the accessor methods that works around this problem by retaining and then autoreleasing the instance variable value in the getter method.

Listing 3-8 Implementing accessors for an object instance variable—better technique

```

- (NSString *)title {
    return [[title retain] autorelease];
}
- (void)setTitle:(NSString *)newTitle {
    if (title != newTitle) {
        [title release];
        title = [newTitle copy];
    }
}

```

In both examples of the setter method above (Listing 3-5 and Listing 3-7), the new `NSString` instance variable is copied rather than retained. Why isn't it retained? The general rule is this: When the object assigned to an instance variable is a value object—that is, an object that represents an attribute such as a string, date, number, or corporate record—you should copy it. You are interested in preserving the attribute value, and don't want to risk having it mutate from under you. In other words, you want your own copy of the object.

However, if the object to be stored and accessed is an entity object, such as an `NSView` or an `NSWindow` object, you should retain it. Entity objects are more aggregate and relational, and copying them can be expensive. One way to determine whether an object is an value object or entity object is to decide whether you're interested in the value of the object or in the object itself. If you're interested in the value, it's probably a value object and you should copy it (assuming, of course, the object conforms to the `NSCopying` protocol).

Another way to decide whether to retain or copy an instance variable in a setter method is to determine if the instance variable is an attribute or relationship. This is especially true for model objects, which are objects that represent the data of an application. An attribute is essentially the same thing as a value object: an object that is a defining characteristic of the object that encapsulates it, such as a color (`NSColor` object) or title (`NSString` object). A relationship on the other hand is just that: a relationship with (or reference to) one or more other objects. Generally, in setter methods you copy the values of attributes and you retain relationships. However, relationships have a cardinality; they can be one-to-one or one-to-many. One-to-many relationships, which are typically represented by collection objects such as `NSArray` instances or `NSSet` instances, may require setter methods to do something

other than simply retain the instance variable. See *Model Object Implementation Guide* for details. For more on object properties as either attributes or relationships, see [“Key-Value Mechanisms”](#) (page 131).

Important: If you are implementing a getter accessor method to load the value of an instance variable lazily—that is, when it is first requested—do *not* call the corresponding setter method in your implementation. Instead, set the value of the instance variable directly in the getter method.

If the setter accessors of a class are implemented as shown in either Listing 3-5 or Listing 3-7, to deallocate an instance variable in the class’s `dealloc` method, all you need to do is invoke the appropriate setter method with an argument of `nil`.

Key-Value Mechanisms

Several mechanisms with “key-value” in their names are fundamental parts of Cocoa: key-value binding, key-value coding, and key-value observing. They are essential ingredients of Cocoa technologies such as bindings, which automatically communicate and synchronize values between objects. They also provide at least part of the infrastructure for making applications scriptable—that is, responsive to AppleScript commands. Key-value coding and key-value observing are especially important considerations in the design of a custom subclass.

Mac OS X Note: The controller classes used in bindings (`NSController` and subclasses) and in application scriptability are defined only by the Application Kit.

The term “key-value” refers to the technique of using the name of a property as the key to obtain its value. The term is part of the vocabulary of the object modeling pattern, which was briefly discussed in [“Storing and Accessing Properties”](#) (page 126). Object modeling derives from the entity-relationship modeling used for describing relational databases. In object modeling, objects—particularly the data-bearing model objects of the Model-View-Controller pattern—have properties, which usually (but not always) take the form of instance variables. A property can either be an attribute, such as name or color, or it can be a reference to one or more other objects. These references are known as relationships, and relationships can be one-to-one or one-to-many. The network of objects in a program form an object graph through their relationships with each other. In object modeling you can use key paths—strings of dot-separated keys—to traverse the relationships in an object graph and access the properties of objects.

Note: For a full description of object modeling, see [“Object Modeling”](#) (page 169).

Key-value binding, key-value coding, and key-value observing are enabling mechanisms for this traversal.

- Key-value binding (KVB) establishes bindings between objects, and also removes and advertises those bindings. It makes use of several informal protocols. A binding for a property must specify the object and a key path to the property.

- Key-value coding (KVC), through an implementation of the `NSKeyValueCoding` informal protocol, makes it possible to get and set the values of an object's properties through keys without having to invoke that object's accessor methods directly. (Cocoa provides a default implementation of the protocol.) A key typically corresponds to the name of an instance variable or accessor method in the object being accessed.
- Key-value observing (KVO), through an implementation of the `NSKeyValueObserving` informal protocol, allows objects to register themselves as observers of other objects. The observed object directly notifies its observers when there are changes to one of its properties. Cocoa implements automatic observer notifications for each property of a KVO-compliant object.

To make each property of a subclass compliant with the requirements of key-value coding, do the following:

- For an attribute or a to-one relationship named *key*, implement accessor methods named *key* (getter) and *setKey:* (setter). For example, if you have a property named `salary`, you would have accessor methods `salary` and `setSalary:`. (If the compiler synthesizes a declared property, it follows this pattern, unless instructed otherwise.)
- For a to-many relationship, if the property is based on an instance variable that is a collection (for example, an `NSArray` object) or an accessor method that returns a collection, give the getter method the same name as the property (for example, `employees`). If the property is mutable, but the getter method doesn't return a mutable collection (such as `NSMutableArray`), you must implement `insertObject:inKeyAtIndex:` and `removeObjectFromKeyAtIndex:`. If the instance variable is not a collection and the getter method does not return an collection, you must implement other `NSKeyValueCoding` methods.

To make your object KVO-compliant, simply ensure that your object is KVC-complaint if you are satisfied with automatic observer notification. However, you may chose to implement manual key-value observing, which requires additional work.

Further Reading: Read *Key-Value Coding Programming Guide* and *Key-Value Observing Programming Guide* to learn more about these technologies. For a summary of the Cocoa bindings technology, which uses KVC, KVO, and KVB, see [“Bindings \(Mac OS X\)”](#) (page 190) in [“Communicating With Objects”](#) (page 177).

Object Infrastructure

If a subclass is well-designed, instances of that class will behave in ways expected of Cocoa objects. Code using the object can compare it to other instances of the class, discover its contents (for example, in a debugger), and perform similar basic operations with the object.

Custom subclass should implement most, if not all, of the following root-class and basic protocol methods:

- `isEqual:` and `hash`—Implement these `NSObject` methods to bring some object-specific logic to their comparison. For example, if what individuates instances of your class is their serial number, make that the basis for equality. For further information, see [“Introspection”](#) (page 90).

- **description**—Implement this `NSObject` method to return a string that concisely describes the properties or contents of the object. This information is returned by the `print object` command in the `gdb` debugger and is used by the `%@` specifier for objects in formatted strings. For example, say you have a hypothetical `Employee` class with attributes for name, date of hire, department, and position ID. The description method for this class might look like the following:

```
- (NSString *)description {
    return [NSString stringWithFormat:@"Employee:Name = %@,\nHire Date = %@, Department = %@, Position = %i\n", [self name],\n[[self dateOfHire] description], [self department],\n[self position]];
}
```

- **copyWithZone:**—If you expect clients of your class to copy instances of it, you should implement this method of the `NSCopying` protocol. Value objects, including model objects, are typical candidates for copying; objects such as `UITableView` and `NSColorPanel` are not. If instances of your class are mutable, you should instead conform to the `NSMutableCopying` protocol.
- **initWithCoder:** and **encodeWithCoder:**—If you expect instances of your class to be archived (for example, a model class), you should adopt the `NSCoding` protocol (if necessary) and implement these two methods. See “[Entry and Exit Points](#)” (page 123) for more on the `NSCoding` methods.

If any of the ancestors of your class adopts a formal protocol, you must also ensure that your class properly conforms to the protocol. That is, if the superclass implementation of any of the protocol methods is inadequate for your class, your class should re-implement the methods.

Error Handling

It is axiomatic for any programming discipline that the programmer should properly handle errors. However, what is “proper” often varies by programming language, application environment, and other factors. Cocoa has its own set of conventions and prescriptions for handling errors in subclass code.

- If the error encountered in a method implementation is a system-level or Objective-C runtime error, create and raise an exception, if necessary, and handle it locally, if possible.

In Cocoa, exceptions are typically reserved for programming or *unexpected* runtime errors such as out-of-bounds collection access, attempts to mutate immutable objects, sending an invalid message, and losing the connection to the window server. You usually take care of these errors with exceptions when an application is being created rather than at runtime. Cocoa predefines several exceptions that you can catch with an exception handler. For information on predefined exceptions and the procedure and API for raising and handling exceptions, see *Exception Programming Topics for Cocoa*.

- For other sorts of errors, including *expected* runtime errors, return `nil`, `NO`, `NULL`, or some other type-suitable form of zero to the caller. Examples of these errors include the inability to read or write a file, a failure to initialize an object, the inability to establish a network connection, or a failure to locate an object in a collection. Use an `NSError` object if you feel it necessary to return supplemental information about the error to the sender.

An `NSError` object encapsulates information about an error, including an error code (which can be specific to the Mach, POSIX, or `OSStatus` domains) and a dictionary of program-specific information. The negative value that is directly returned (`nil`, `NO`, and so on) should be the principal indicator of error; if you do communicate more specific error information, return an `NSError` object indirectly in an argument of the method.

- If the error requires a decision or action from the user, display an alert dialog.

For Mac OS X, use an instance of the `NSAlert` class (and related facilities) for displaying an alert dialog and handling the user's response. For iPhone OS, use the facilities of the `UIActionSheet` or `UIAlertView` classes. See *Dialogs and Special Panels* for information.

For further information about `NSError` objects, handling errors, and displaying error alerts, see *Error Handling Programming Guide For Cocoa*.

Resource Management and Other Efficiencies

There are all sorts of things you can do to enhance the performance of your objects and, of course, the application that incorporates and manages those objects. These procedures and disciplines range from multithreading to drawing optimizations and techniques for reducing your code footprint. You can learn more about these things by reading *Cocoa Performance Guidelines* and other performance documents.

Yet you can, before even adopting a more advanced performance technique, improve the performance of your objects significantly by following three simple, common-sense guidelines:

- Don't load a resource or allocate memory until you really need it.

If you load a resource of the program, such as a nib file or an image, and don't use it until much later, or don't use it at all, that is a gross inefficiency. Your program's memory footprint is bloated for no good reason. You should load resources or allocate memory only when there is an immediate need.

For example, if your application's preferences window is in a separate nib file, don't load that file until the user first chooses Preferences from the application menu. The same caution goes for allocating memory for some task; hold off on allocation until the need for that memory arises. This technique of lazy-loading or lazy-allocation is easily implemented. For example, say your application has an image that it loads, upon the first user request, for displaying in the user interface. Listing 3-9 shows one approach: loading the image in the getter accessor for the image.

Listing 3-9 Lazy-loading of a resource

```
- (NSImage *)fooImage {
    if (!fooImage) { // fooImage is an instance variable
        NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"Foo"
ofType:@"jpg"];
        if (!imagePath) return nil;
        fooImage = [[NSImage alloc] initWithContentsOfFile:imagePath];
    }
    return fooImage;
}
```

- Use the Cocoa API; don't dig down into lower-level programming interfaces.

Much effort has gone into making the implementations of the Cocoa frameworks as robust, secure, and efficient as possible. Moreover, these implementations manage interdependencies that you might not be aware of. If, instead of using the Cocoa API for a certain task, you decide to “roll your own” solution using lower-level interfaces, you’ll probably end up writing more code, which introduces a greater risk of error or inefficiency. Also, by leveraging Cocoa, you are better prepared to take advantage of future enhancements while at the same time you are more insulated from changes in the underlying implementations. So use the Cocoa alternative for a programming task if one exists and its capabilities answer your needs.

- Practice good memory-management techniques.

If you decide not to enable garbage collection (which is disabled by default in iPhone OS), perhaps the most important single thing you can do to improve the efficiency and robustness of your custom objects is to practice good memory-management techniques. Make sure every object allocation, `copy` message, or `retain` message is matched with a corresponding `release` message. Become familiar with the policies and techniques for proper memory management and practice, practice, practice them. See *Memory Management Programming Guide for Cocoa* for complete details.

Functions, Constants, and Other C Types

Because Objective-C is a superset of ANSI C, it permits any C type in your code, including functions, `typedef` structures, `enum` constants, and macros. An important design issue is when and how to use these types in the interface and implementation of custom classes.

The following list offers some guidance on using C types in the definition of a custom class:

- Define a function rather than a method for functionality that is often requested but doesn’t need to be overridden by subclasses. The reason for doing this is performance. In these cases, it’s best for the function to be private rather than being part of the class API. You can also implement functions for behavior that is not associated with any class (because it is global) or for operations on simple types (C primitives or structures). However, for global functionality it might be better—for reasons of extensibility—to create a class from which a singleton instance is generated.
- Define a structure type rather than a simple class when the following conditions apply:
 - You don’t expect to augment the list of fields.
 - All the fields are public (for performance reasons).
 - None of the fields is dynamic (dynamic fields might require special handling, such as retaining or releasing).
 - You don’t intend to make use of object-oriented techniques, such as subclassing.

Even when all these conditions apply, the primary justification for structures in Objective-C code is performance. In other words, if there is no compelling performance reason, a simple class is preferable.

- Declare `enum` constants rather than `#define` constants. The former are more suitable for typing, and you can discover their values in the debugger.

When the Class Is Public (Mac OS X)

When you are defining a custom class for your own use, such as a controller class for an application, you have great flexibility because you know the class well and can always redesign it if you have to. However, if your custom class will likely be a superclass for other developers—in other words, if it's a public class—the expectations others have of your class mean that you have to be more careful of your design.

Here are few guidelines for developers of public Cocoa classes:

- Make sure those instance variables that subclasses need to access are scoped as `@protected`.
- Add one or two reserved instance variables to help ensure binary compatibility with future versions of your class (see [“Instance Variables”](#) (page 122)).
- In memory-managed code, implement your accessor methods to properly handle memory management of vended objects (see [“Storing and Accessing Properties”](#) (page 126)). Clients of your class should get back autoreleased objects from your getter accessors.
- Observe the naming guidelines for public Cocoa interfaces as recommended in *Coding Guidelines for Cocoa*.
- Document the class, at least with comments in the interface file.

Multithreaded Cocoa Programs

It has long been a standard programming technique to optimize program performance through multithreading. By having data processing and I/O operations run on their own secondary threads and dedicating the main thread to user-interface management, you can enhance the responsiveness of your applications. But in recent years, multithreading has assumed even greater importance with the emergence of multicore computers and symmetric multiprocessing (SMP). Concurrent processing on such systems involves synchronizing the execution of not only multiple threads on the same processor but multiple threads on multiple processors.

But making your program multithreaded is not without risk or costs. Although threads share the virtual address space of their process, they still add to the process's memory footprint; each thread requires an additional data structure, associated attributes, and its own stack space. Multithreading also requires greater complexity of program design to synchronize access to shared memory among multiple threads. Such designs make a program harder to maintain and debug. Moreover, multithreading designs that overuse locks can actually degrade performance (relative to a single-threaded program) because of the high contention for shared resources. Designing a program for multithreading often involves tradeoffs between performance and protection and requires careful consideration of your data structures and the intended usage pattern for extra threads. Indeed, in some situations the best approach is to avoid multithreading altogether and keep all program execution on the main thread.

Further Reading: To get a solid understanding of issues and guidelines related to multithreading, read *Threading Programming Guide*. This document contains several articles specific to Cocoa multithreading, including “Cocoa Thread Safety,” “Creating Threads in Cocoa,” and “Using Locks in Cocoa.”

Multithreading and Multiprocessing Resources

Cocoa provides several classes that are useful for multithreading programs.

Threads and Run Loops

In Cocoa threads are represented by objects of the `NSThread` class. The implementation of the class is based upon POSIX threads. Although the interface of `NSThread` does not provide as many options for thread management as do POSIX threads, it is sufficient for most multithreading needs.

Run loops are a key component of the Mac OS X architecture for the distribution of events. Run loops have input sources (typically ports or sockets) and timers; they also have input modes for specifying which input sources the run loop listens to. Objects of the `NSRunLoop` class represent run loops in Cocoa. Each thread has its own run loop, and the run loop of the main thread is set up and run automatically when a process begins execution. Secondary threads, however, must run their own run loops.

Operations and Operation Queues

With the `NSOperation` and `NSOperationQueue` classes you can manage the execution of one or more encapsulated tasks that may be concurrent or nonconcurrent. (Note that the word “task” here does not necessarily imply an `NSTask` object; a task, or operation, for example, typically runs on a secondary thread within the same process.) An `NSOperation` object represents a discrete task that can be executed only once. Generally, you use `NSOperationQueue` objects to schedule operations in a sequence determined by priority and interoperation dependency. An operation object that has dependencies does not execute until all of its dependent operation objects finish executing. An operation object remains in a queue until it is explicitly removed or it finishes executing.

Note: The `NSOperation` and `NSOperationQueue` classes were introduced in Mac OS X v10.5. They are not available in earlier versions of the operating system.

Locks and Conditions

Locks act as resource sentinels for contending threads; they prevent threads from simultaneously accessing a shared resource. Several classes in Cocoa provide lock objects of different types.

Classes	Description
<code>NSLock</code>	Implements a mutex lock—a lock that enforces mutually exclusive access to a shared resource. Only one thread can acquire the lock and use the resource, thereby blocking other threads until it releases the lock.

Classes	Description
<code>NSRecursiveLock</code>	Implements a recursive lock—a mutex lock that can be acquired multiple times by the thread that currently owns it. The lock remains in place until each recursive acquisition of the lock has been balanced with a call that releases the lock.
<code>NSConditionLock</code> <code>NSCondition</code>	Both of these classes implement condition locks. This type of lock combines a semaphore with a mutex lock to implement locking behavior that is effective based on a program-defined condition. A thread blocks and waits for another thread to signal the condition, at which point it (or another waiting thread) is unblocked and continues execution. You can signal multiple threads simultaneously, causing them to unblock and continue execution.

`NSConditionLock` and `NSCondition` are similar in purpose and implementation; both are object-oriented implementations of `pthread` condition locks. The implementation of `NSConditionLock` is more thorough but less flexible (although it does offer features such as locking timeouts); it implements both the semaphore signaling *and* the mutex locking for you. The implementation of `NSCondition`, on the other hand, wraps the `pthread` condition variables and mutex locks and requires you to do your own locking, signaling, and predicate state management. However, it is more flexible. This approach closely follows the implementation pattern for `pthread` conditions.

Note: The `NSCondition` class was introduced in Mac OS X v10.5. It is not available in earlier versions of the operating system.

Interthread Communication

When your program has multiple threads, those threads often need a way to communicate with each other. As you might have read in the preceding section, the condition-lock classes use `pthread`-based condition signaling (semaphores) as a way to communicate between threads. But Cocoa offers several other resources for interthread communication.

Objects of the `NSMachPort` and `NSMessagePort` classes enable interthread communication over Mach ports. Objects of the `NSSocketPort` class enable interthread *and* interprocess communication over BSD sockets. You can also use distributed objects as a mechanism for interthread and interprocess communication; for more on this port-based communication abstraction, see *Distributed Objects Programming Topics*.

In addition, the `NSObject` class provides the `performSelectorOnMainThread:withObject:waitUntilDone:` method which allows secondary threads to communicate with and send data to the main thread.

Multithreading Guidelines for Cocoa Programs

Whether your program uses the Cocoa frameworks or something else, there are some common observations that can be made about the appropriateness and wisdom of having multiple threads in your program. You might start by asking yourself the following questions; if you answer “yes” to any of them, you should consider multithreading:

- Is there a CPU- or I/O-intensive task that could block your application’s user interface?

The main thread manages an application's user interface—the view part of the Model-View-Controller pattern. Thus the application can perform work that involves the application's data model on one or more secondary threads. In a Cocoa application, because controller objects are directly involved in the management of the user interface, you would run them on the main thread.

- Does your program have multiple sources of input or multiple outputs, and is it feasible to handle them simultaneously?
- Do you want to spread the work your program performs across multiple CPU cores?

Let's say one of these situations applies to your application. As noted earlier, multithreading can involve costs and risks as well as benefits, so it's worthwhile to consider alternatives to multithreading. For example, you might try asynchronous processing.

Just as there are situations that are appropriate for multithreading, there are situations where it's not:

- The work requires little processing time.
- The work requires steps that must be performed serially.
- Underlying subsystems are not thread safe.

The last item brings up the important issue of thread safety. There are some guidelines you should observe to ensure—as much as possible—that your multithreaded code is thread safe. Some of these guidelines are general and others are specific to Cocoa applications. The general guidelines are as follows:

- Avoid sharing data structures across threads if possible.

Instead you can give each thread its own copy of an object (or other data) and then synchronize changes using a transaction-based model. Ideally you want to minimize resource contention as much as possible.

- Lock data at the right level.

Locks are an essential component of multithreaded code, but they do impose a performance bottleneck and may not work as planned if used at the wrong level.

- Catch local exceptions in your thread.

Each thread has its own call stack and is therefore responsible for catching any local exceptions and cleaning things up. The exception cannot be thrown to the main thread or any other thread for handling. Failure to catch and handle an exception may result in the termination of the thread or the owning process.

- Avoid `volatile` variables in protected code.

The additional guidelines for Cocoa programs are few:

- Deal with immutable objects in your code as much as possible, especially across interface boundaries.

Objects whose value or content cannot be modified—immutable objects—are usually thread safe. Mutable objects are often not. As a corollary to this guideline, respect the declared mutability status of objects returned from method calls; if you receive an object declared as immutable but it is mutable and you modify it, you can cause behavior that is catastrophic to the program.

- The main thread of a Cocoa application is responsible for receiving and dispatching events. Although your application can handle events on a secondary thread instead of the main thread, if it does it must keep all event-handling code on that thread. If you spin off the handling of events to different threads, user events (such as the letters typed on the keyboard) can occur out of sequence.
- If a secondary thread is to draw a view, ensure that all drawing code falls between calls to the `NSView` methods `lockFocusIfCanDraw` and `unlockFocus`.

Note: You can learn more about these guidelines in the “Thread Safety Guidelines” and “Cocoa Thread Safety” articles of *Threading Programming Guide*.

Are the Cocoa Frameworks Thread Safe?

Before you can make your program thread safe, it’s essential to know the thread safety of the frameworks that your program relies on. The core Cocoa frameworks, Foundation and Application Kit, have parts that are thread safe and parts that are not.

Generally, the classes of Foundation whose objects are immutable collections (`NSArray`, `NSDictionary`, and so on) as well as those whose immutable objects encapsulate primitive values or constructs (for example, `NSString`, `NSNumber`, `NSException`) are thread safe. Conversely, objects of the mutable versions of these collection and primitive-value classes are not thread safe. A number of classes whose objects represent system-level entities—`NSTask`, `NSRunLoop`, `NSPipe`, `NSHost`, and `NSPort`, among others—are not thread safe. (`NSThread` and the lock classes, on the other hand, are thread safe.)

In the Application Kit, windows (`NSWindow` objects) are generally thread safe in that you can create and manage them on a secondary thread. Events (`NSEvent` objects), however, are safely handled only on the same thread, whether that be the main thread or a secondary thread; otherwise you run the risk of having events get out of sequence. Drawing and views (`NSView` objects) are generally thread safe, although operations on views such as creating, resizing, moving should happen on the main thread.

All UIKit objects should be used on the main thread only.

For more information on the thread safety of the Cocoa frameworks, see “Cocoa Thread Safety” in *Threading Programming Guide*.

Cocoa Design Patterns

Many of the architectures and mechanisms of the Cocoa environment make effective use of design patterns: abstract designs that solve recurring problems in a particular context. This chapter describes the major implementations of design patterns in Cocoa, focusing in particular on Model-View-Controller and object modeling. Its main purpose is to give you a greater awareness of design patterns in Cocoa and encourage you to take advantage of these patterns in your own software projects.

What Is a Design Pattern?

A design pattern is a template for a design that solves a general, recurring problem in a particular context. It is a tool of abstraction that is useful in fields like architecture and engineering as well as software development. The following sections summarize what design patterns are, explains why they're important for object-oriented design, and looks at a sample design pattern.

A Solution to a Problem in a Context

As a developer, you might already be familiar with the notion of design patterns in object-oriented programming. They were first authoritatively described and cataloged in *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (commonly referred to as the “Gang of Four”). That book, originally published in 1994, was soon followed by other books and articles that further explored and elaborated design patterns in object-oriented systems.

The succinct definition of a design pattern is “a solution to a problem in a context.” Let’s parse this by working backwards through the phrase. The context is a recurring situation in which the pattern applies. The problem is the goal you are trying to achieve in this context as well as any constraints that come with the context. And the solution is what you’re after: a general design for the context that achieves the goal and resolves the constraints.

A design pattern abstracts the key aspects of the structure of a concrete design that has proven to be effective over time. The pattern has a name and identifies the classes and objects that participate in the pattern along with their responsibilities and collaborations. It also spells out consequences (costs and benefits) and the situations in which the pattern can be applied. A design pattern is a kind of template or guide for a particular design; in a sense, a concrete design is an “instantiation” of a pattern.

Design patterns are not absolute. There is some flexibility in how you can apply them, and often things such as programming language and existing architectures can determine how the pattern is applied.

Several themes or principles of design influence design patterns. These design principles are rules of thumb for constructing object-oriented systems, such as “encapsulate the aspects of system structure that vary” and “program to an interface, not an implementation.” They express important insights. For example, if you isolate the parts of a system that vary and encapsulate them, they can vary independently of other parts of the system, especially if you define interfaces for them that are not tied to implementation specifics. You can later alter or extend those variable parts without affecting the other parts of the system. You thus eliminate dependencies and reduce couplings between parts, and consequently the system becomes more flexible and robust to change.

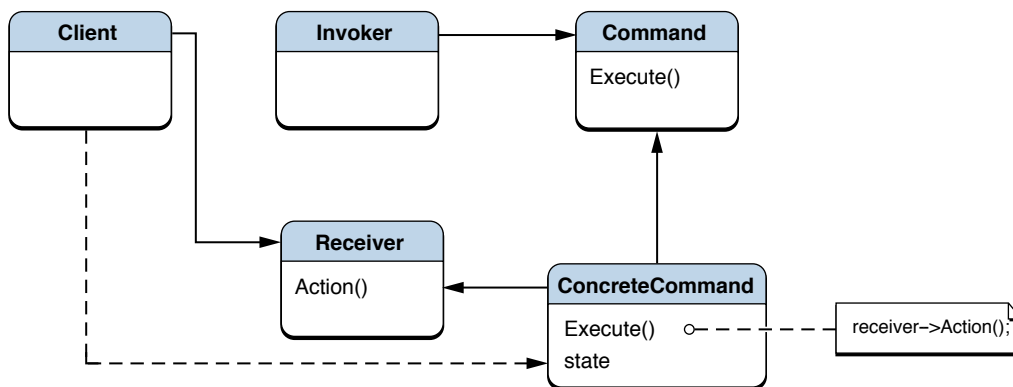
Benefits such as these make design patterns an important consideration when you’re writing software. If you find, adapt, and use patterns in your program’s design, that program—and the objects and classes that it comprises—will be more reusable, more extensible, and easier to change when future requirements demand it. Moreover, programs that are based on design patterns are generally more elegant and efficient than programs that aren’t, as they require fewer lines of code to accomplish the same goal.

An Example: The Command Pattern

Most of the book by the Gang of Four consists of a catalog of design patterns. It categorizes the patterns in the catalog by scope (class or object) and by purpose (creational, structural, or behavioral). Each entry in the catalog discusses the intent, motivation, applicability, structure, participants, collaborations, consequences, and implementation of a design pattern. One of these entries is the Command pattern (an object-behavioral pattern).

The stated intent of the Command pattern is to “encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.” The pattern separates an object sending a message from the objects that receive and evaluate those messages. The originator of the message (the client) encapsulates a request by binding together one or more actions on a specific receiver. The encapsulated message can be passed around between objects, placed in queues or otherwise stored for later invocation, and dynamically modified to vary the receiver or message parameters. Figure 4-1 shows the structure diagram for the pattern.

Figure 4-1 Structure diagram for the Command pattern



For a developer familiar with Cocoa, this sketchy overview of the Command pattern might ring a bell. The pattern perfectly describes a class in the Foundation framework whose purpose is to encapsulate messages: `NSInvocation`. As the pattern's intent states, one of its purposes is to make operations undoable. Invocation objects are used in the Cocoa designs for undo management as well as distributed objects, which is an architecture for interprocess communication. The Command pattern also describes (although less perfectly) the target-action mechanism of Cocoa in which user-interface control objects encapsulate the target and action of the messages they send when users activate them.

In its framework classes and in its languages and runtime, Cocoa has already implemented many of the cataloged design patterns for you. (These implementations are described in [“How Cocoa Adapts Design Patterns”](#) (page 143).) You can satisfy many of your development requirements by using one of these “off-the-shelf” adaptations of a design pattern. Or you may decide your problem and its context demands a brand new pattern-based design of your own. The important thing is to be aware of patterns when you are developing software and to use them in your designs when appropriate.

How Cocoa Adapts Design Patterns

You can find adaptations of design patterns throughout Cocoa, in both its Mac OS X and iPhone OS versions. Mechanisms and architectures based on patterns are common in Cocoa frameworks and in the Objective-C runtime and language. Cocoa often puts its own distinctive spin on a pattern because its designs are influenced by factors such as language capabilities or existing architectures.

This section contains entries for most of the design patterns cataloged in *Design Patterns: Elements of Reusable Object-Oriented Software*. Each entry summarizes the pattern and describes the Cocoa implementations of it. Only patterns that Cocoa implements are listed, and each discussion of a pattern in the following sections takes place within a particular Cocoa context. It is recommended that you become familiar with these patterns, as you will find them useful in Cocoa software development.

Implementations of design patterns in Cocoa come in various forms. Some of the designs described in the following sections—such as protocols and categories—are features of the Objective-C language. In other cases, the “instance of a pattern” is implemented in one class or a group of related classes (for example, class clusters and singleton classes). And in other cases the pattern adaptation is a major framework architecture, such as the responder chain. Some of the pattern-based mechanisms you get almost “for free” while others require some work on your part. And even if Cocoa does not implement a pattern, you are encouraged to do so yourself when the situation warrants it; for example, object composition (Decorator pattern) is often a better technique than subclassing for extending class behavior.

Two design patterns are reserved for later sections, Model-View-Controller (MVC) and object modeling. MVC is a compound or aggregate pattern, meaning that it is based on several catalog patterns. Object modeling has no counterpart in the Gang of Four catalog, instead originating from the domain of relational databases. Yet MVC and object modeling are perhaps the most important and pervasive design patterns or metaphors in Cocoa, and to a large extent they are interrelated patterns. They play a crucial role in the design of several technologies, including bindings, undo management, scripting, and the document architecture. To learn more about these patterns, see [“The Model-View-Controller Design Pattern”](#) (page 161) and [“Object Modeling”](#) (page 169).

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes. The client is decoupled from any of the specifics of the concrete object obtained from the factory.

Class Cluster

A class cluster is an architecture that groups a number of private, concrete subclasses under a public, abstract superclass. The abstract superclass declares methods for creating instances of its private subclasses. The superclass dispenses an object of the proper concrete subclass based on the creation method invoked. Each object returned may belong to different private concrete subclass.

Cocoa limits class clusters to the generation of objects whose storage of data may vary depending on circumstances. The Foundation framework has class clusters for `NSString`, `NSData`, `NSDictionary`, `NSSet`, and `NSArray` objects. The public superclasses include the aforementioned immutable classes as well as the complementary mutable classes `NSMutableString`, `NSMutableData`, `NSMutableDictionary`, `NSMutableSet`, and `NSMutableArray`.

Uses and Limitations

You use one of the public classes of a class cluster when you want to create immutable or mutable objects of the type represented by the cluster. With class clusters there is a trade-off between simplicity and extensibility. A class cluster simplifies the interface to a class and thus makes it easier to learn and use the class. However, it is generally more difficult to create custom subclasses of the abstract class of a class cluster.

Further Reading: “[Class Clusters](#)” (page 98) provides more information about class clusters in Cocoa.

Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. It decouples the client from the class of the targeted object.

Protocols

A protocol is a language-level (Objective-C) feature that makes it possible to define interfaces that are instances of the Adapter pattern. (In Java, “interface” is synonymous with “protocol.”) If you want a client object to communicate with another object, but their incompatible interfaces make that difficult, you can define a protocol, which is essentially a series of method declarations unassociated with a class. The class of the other object then formally adopts the protocol and “conforms” to it by implementing one or more of the methods of the protocol. The protocol may require the conforming class to implement some of its methods and may leave the implementation of others optional. The client object can then send messages to the other object through the protocol interface.

Protocols make a set of method declarations independent of the class hierarchy. They make it possible to group objects on the basis of conformance to a protocol as well as class inheritance. The `NSObject` method `conformsToProtocol:` permits you to verify an object's protocol affiliation.

In addition to formal protocols, Cocoa has the notion of informal protocols. This type of protocol is a category on the `NSObject` class, thus making any object a potential implementer of any method in the category (see “[Categories](#)” (page 151)). The methods in an informal protocol can be selectively implemented. Informal protocols are part of the implementation of the delegation mechanism (see “[Delegation](#)” (page 149)).

Note that the design of protocols does not perfectly match the Adapter pattern. But it is a means for allowing classes with otherwise incompatible interfaces to work together.

Uses and Limitations

You primarily use a protocol to declare an interface that hierarchically unrelated classes are expected to conform to if they wish to communicate. But you can also use protocols to declare an interface of an object while concealing its class. The Cocoa frameworks include many formal protocols that enable custom subclasses to communicate with them for specific purposes. For example, the Foundation framework includes the `NSObject`, `NSCopying`, and `NSCoding` protocols, which are all very important ones. Application Kit protocols include `NSDraggingInfo`, `NSTextInput`, and `NSChangeSpelling`.

Formal protocols implicitly require the conforming class to implement *all* declared methods. However, they can mark single methods or groups of methods with the `@optional` directive, and the conforming class may choose to implement those. They are also fragile; once you define a protocol and make it available to other classes, future changes to it (except for additional optional methods) can break those classes.

Further Reading: More more information on formal protocols, see *Protocols The Objective-C 2.0 Programming Language*.

Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. Each object either handles the request or passes it to the next object in the chain.

Responder Chain

The Application Kit and UIKit frameworks include an architecture known as the responder chain. This chain consists of a series of responder objects (that is, objects inheriting from `NSResponder` or, in UIKit, `UIResponder`) along which an event (for example, a mouse click) or action message is passed and (usually) eventually handled. If a given responder object doesn't handle a particular message, it passes the message to the next responder in the chain. The order of responder objects in the chain is generally determined by the view hierarchy, with the progression from lower-level to higher-level responders in the hierarchy, culminating in the window object that manages the view hierarchy or the delegate of the window object or the global application object. The exact paths of events and action messages up the responder chain is different. An application can have as many responder chains as it has windows (or even local hierarchies of views); but only one responder chain can be active at a time—the one associated with the currently active window.

A similar chain of responders exists for error handling in an application.

The design of the view hierarchy, which is closely related to the responder chain, adapts the Composite pattern ([“Composite”](#) (page 147)). Action messages—messages originating from control objects—are based on the target-action mechanism, which is an instance of the Command pattern ([“Command”](#) (page 146)).

Uses and Limitations

When you construct a user interface for a program either by using Interface Builder or programmatically, you get one or more responder chains “for free.” The responder chain goes hand in hand with a view hierarchy, which you get automatically when you make a view object a subview of a window’s content view. If you have a custom view added to a view hierarchy, it becomes part of the responder chain; if you implement the appropriate `NSResponder` or `UIResponder` methods, you can receive and handle events and action messages. A custom object that is a delegate of a window object or the global application object (`NSApp` in the Application Kit) can also receive and handle those messages.

You can also programmatically inject custom responders into the responder chain and you can programmatically manipulate the order of responders.

Further Reading: The responder chain for events and action messages and the responder chain for handling errors are described (in an Application Kit context) in *Cocoa Event-Handling Guide* and *Error Handling Programming Guide For Cocoa*. This document summarizes the view hierarchy in the section on the [“Composite”](#) (page 147) pattern and is more fully described in [“The Core Application Architecture on Mac OS X”](#) (page 201).

Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. The request object binds together one or more actions on a specific receiver. The Command pattern separates an object making a request from the objects that receive and execute that request.

Invocation Objects

An instance of the `NSInvocation` class encapsulates an Objective-C message. An invocation object contains a target object, method selector, and method arguments. You can dynamically change the target of the message dispatched by the invocation object as well as its arguments; once the invocation is executed, you can also obtain the return value from the object. With a single invocation object, you can repeatedly invoke a message with multiple variations in target and arguments.

The creation of an `NSInvocation` object requires an `NSMethodSignature` object, which is an object that encapsulates type information related to the arguments and return value of a method. An `NSMethodSignature` object, in turn, is created from a method selector. The implementation of `NSInvocation` also makes use of functions of the Objective-C runtime.

Uses and Limitations

`NSInvocation` objects are part of the programmatic interfaces of distributed objects, undo management, message forwarding, and timers. You can also use invocation objects in similar contexts where you need to decouple an object sending a message from the object that receives the message.

Distributed objects is a technology for interprocess communication. See [“Proxy”](#) (page 159) for more on distributed objects.

Further Reading: See *NSInvocation Class Reference* for details of invocation objects. Also, consult the following documents for information about related technologies: *Distributed Objects Programming Topics*, *Undo Architecture*, *Timers*, and the “The Runtime System” in *The Objective-C 2.0 Programming Language*.

The Target-Action Mechanism

The target-action mechanism enables a control object—that is, an object such as a button, slider, or text field—to send a message to another object that can interpret the message and handle it as an application-specific instruction. The receiving object, or the target, is usually a custom controller object. The message—named an action message—is determined by a selector, a unique runtime identifier of a method.

In the Application Kit, the cell object that a control owns typically encapsulates the target and action; when the user clicks or otherwise activates the control, it extracts the information from its cell and sends the message. (A menu item also encapsulates target and action, and sends an action message when the user chooses it.) The target-action mechanism can work on the basis of a selector (and not a method signature) because the signature of an action method in the Application Kit by convention is always the same.

In UIKit, the target-action mechanism does not rely on cells. Instead, a control maps a target and action to one or more multi-touch events that can occur on the control.

Uses and Limitations

When creating a Cocoa application for Mac OS X, you can set a control’s action and target through the Interface Builder application. You thereby let the control initiate custom behavior without writing any code for the control itself. The action selector and target connection are archived in a nib file and are restored when the nib is unarchived. You can also change the target and action dynamically by sending the control or its cell `setTarget:` and `setAction:` messages.

In UIKit you must set the target and action for a control programmatically.

A Cocoa application for Mac OS X can use target-action to instruct a custom controller object to transfer data from the user interface to a model object, or to display data in a model object. The Cocoa bindings technology obviates the need to use target-action for this purpose. See *Cocoa Bindings Programming Topics* for more about this technology.

Controls and cells in the Application Kit and UIKit do not retain their targets. See [“Ownership of Delegates, Observers, and Targets”](#) (page 200) for further information.

Further Reading: See [“The Target-Action Mechanism”](#) (page 184) for further information.

Composite

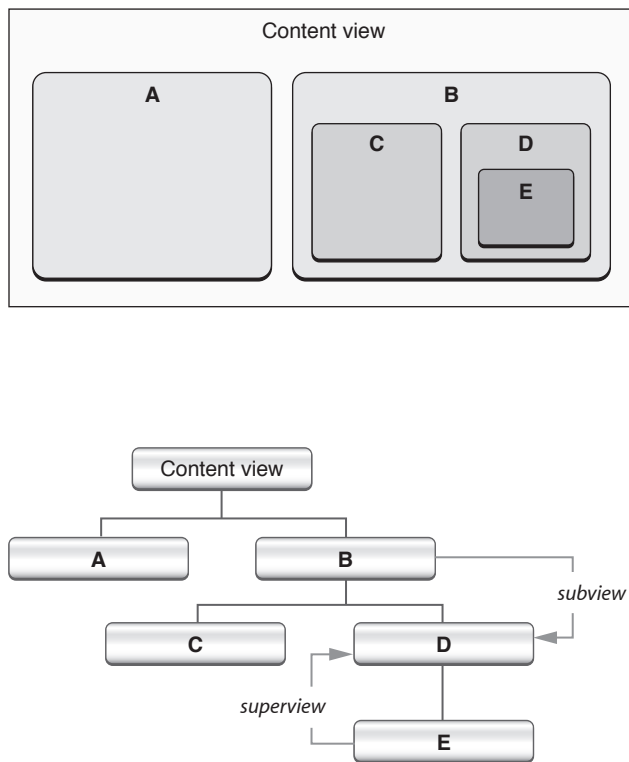
Compose related objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

The Composite pattern is part of the Model-View-Controller aggregate pattern.

View Hierarchy

The views (`NSView` or `UIView` objects) in a window are internally structured into a view hierarchy. At the root of the hierarchy is a window (`NSWindow` or `UIWindow` object) and its content view, a transparent view that fills the window's content rectangle. Views that are added to the content view become subviews of it, and they become the superviews of any views added to them. A view can have one (and only one) superview and zero or more subviews. Visually you apprehend this structure as containment: a superview contains its subviews. Figure 4-2 shows the structural and visual aspects of the view hierarchy.

Figure 4-2 The view hierarchy, structural and visual



The view hierarchy is a structural architecture that plays a part in both drawing and event handling. A view has two bounding rectangles, its frame and its bounds, that affect how graphics operations with the view take place. The frame is the exterior boundary; it locates the view in its superview's coordinate system, defines its size, and clips drawing to its edges. The bounds, the interior bounding rectangle, defines the internal coordinate system of the surface where the view draws itself.

When a window is asked by the windowing system to prepare itself for display, superviews are asked to render themselves before their subviews. When you send some messages to a view—for example, a message that requests it to redraw itself—the message is propagated to subviews. You can thus treat a branch of the view hierarchy as a unified view.

The view hierarchy is also used by the responder chain for handling events and action messages. See the summary of the responder chain in the Chain of Responsibility pattern entry ([“Chain of Responsibility”](#) (page 145)).

Uses and Limitations

You create or modify a view hierarchy whenever you add a view to another view, either programmatically or using Interface Builder. The Application Kit framework automatically handles all the relationships associated with the view hierarchy.

Further Reading: For learn more about the view hierarchy, read [“The View Hierarchy”](#) (page 217) in this document. *Cocoa Drawing Guide* also discusses the view hierarchy.

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. As does subclassing, adaptation of the Decorator pattern allows you to incorporate new behavior without modifying existing code. Decorators wrap an object of the class whose behavior they extend. They implement the same interface as the object they wrap and add their own behavior either before or after delegating a task to the wrapped object. The Decorator pattern expresses the design principle that classes should be open to extension but closed to modification.

General Comments

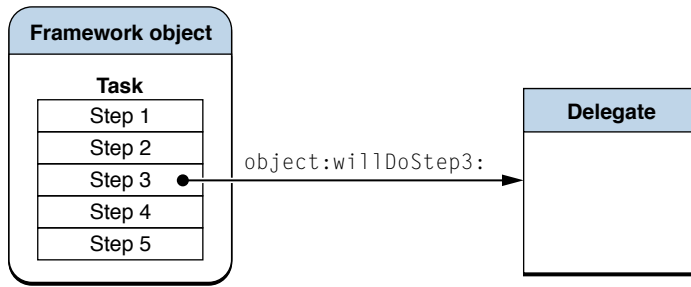
Decorator is a pattern for object composition, which is something that you are encouraged to do in your own code (see [“When to Make a Subclass”](#) (page 117)). Cocoa, however, provides some classes and mechanisms of its own (discussed in the sections below) that are based on the pattern. In these implementations, the extending object does not completely duplicate the interface of the object that it wraps, although the implementations use different techniques for interface sharing.

Cocoa uses the Decorator pattern in the implementation of several of its classes, including `NSAttributedString`, `NSScrollView`, and `UIDatePicker`. The latter two classes are examples of compound views, which group together simple objects of other view classes and coordinate their interaction.

Delegation

Delegation is a mechanism in which a host object embeds an weak reference (“weak” in the sense that it’s a simple pointer reference, unretained) to another object—its delegate—and periodically sends messages to the delegate when it requires its input for a task. The host object is generally an “off-the-shelf” framework object (such as a `NSWindow` or `NSXMLParser` object) that is seeking to accomplish something, but can only do so in a generic fashion. The delegate, which is almost always an instance of a custom class, acts in coordination with the host object, supplying program-specific behavior at certain points in the task (see Figure 4-3). Thus delegation makes it possible to modify or extend the behavior of another object without the need for subclassing.

Figure 4-3 Framework object sending a message to its delegate



Delegation, in the simple sense of one object delegating a task to another object, is a common technique in object-oriented programming. However, Cocoa implements delegation in a unique way. A host class uses a formal protocol or an informal protocol to define an interface that the delegate object may choose to implement. All the methods in the informal protocol are optional, and the formal protocol may declare optional methods, allowing the delegate to implement only some of the methods in the protocol. Before it attempts to send a message to its delegate, the host object determines whether it implements the method (via a `respondsToSelector:` message) to avoid runtime exceptions. For more on formal and informal protocols, see [“Protocols”](#) (page 144).

Some classes in the Cocoa frameworks also send messages to their data sources. A data source is identical in all respects to a delegate, except that the intent is to provide the host object with data to populate a browser, a table view, or similar user-interface view. A data source also may be required to implement some methods of the protocol, unlike a delegate.

Delegation is not a strict implementation of the Decorator pattern. The host (delegating) object does not wrap an instance of the class it wants to extend; indeed, it’s the other way around, in that the delegate is specializing the behavior of the delegating framework class. There is no sharing of interface either, other than the delegation methods declared by the framework class.

Delegation in Cocoa is also part of the Template Method pattern ([“Template Method”](#) (page 160)).

Uses and Limitations

Delegation is a common design in the Cocoa frameworks. Many classes in the Application Kit and UIKit frameworks send messages to delegates, including `NSApplication`, `UIApplication`, `UITableView`, and several subclasses of `NSView`. Some classes in the Foundation framework, such as `NSXMLParser` and `NSStream`, also maintain delegates. You should always use a class’s delegation mechanism instead of subclassing the class, unless the delegation methods do not allow you to accomplish your goal.

Although you can dynamically change the delegate, only one object can be a delegate at a time. Thus if you want multiple objects to be informed of a particular program event at the same time, you cannot use delegation. However, you can use the notification mechanism for this purpose. A delegate automatically receives notifications from its delegating framework object as long as the delegate implements one or more of the notification methods declared by the framework class. See the discussion of notifications in the Observer pattern ([“Observer”](#) (page 157)).

Delegating objects in the Application Kit do not retain their delegates or data sources. See [“Ownership of Delegates, Observers, and Targets”](#) (page 200) for further information.

Further Reading: For further information on delegation, see [“Delegates and Data Sources”](#) (page 179).

Categories

A category is a feature of the Objective-C language that enables you to add methods (interface and implementation) to a class without having to make a subclass. There is no runtime difference—within the scope of your program—between the original methods of the class and the methods added by the category. The methods in the category become part of the class type and are inherited by all the class’s subclasses.

As with delegation, categories are not a strict adaptation of the Decorator pattern, fulfilling the intent but taking a different path to implementing that intent. The behavior added by categories is a compile-time artifact, and is not something dynamically acquired. Moreover, categories do not encapsulate an instance of the class being extended.

Uses and Limitations

The Cocoa frameworks define numerous categories, most of them informal protocols (which are summarized in [“Protocols”](#) (page 144)). Often they use categories to group related methods. You may implement categories in your code to extend classes without subclassing or to group related methods. However, you should be aware of these caveats:

- You cannot add instance variables to the class.
- If you override existing methods of the class, your application may behave unpredictably.

Further Reading: See [“Defining a Class”](#) in *The Objective-C 2.0 Programming Language* for more information on categories.

Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use by reducing complexity and hiding the communication and dependencies between subsystems.

NSImage

The `NSImage` class provides a unified interface for loading and using images that can be bitmap-based (such as JPEG, PNG, or TIFF format) or vector-based (EPS or PDF format). `NSImage` can keep more than one representation of the same image; each representation is a kind of `NSImageRep` object. `NSImage` automates the choice of the representation that is appropriate for a particular type of data and for a given display device. It also hides the details of image manipulation and selection so that the client can use many different underlying representations interchangeably.

Uses and Limitations

Because `UIImage` supports several different representations of what an image is, some requested attributes might not apply. For example, asking a image for the color of a pixel will not work if the underlying image representation is vector-based and device-independent.

Note: See *Cocoa Drawing Guide* for a discussion of `UIImage` and image representations.

Iterator

Provide a way to access the elements of an aggregate object (that is, a collection) sequentially without exposing its underlying representation. The Iterator pattern transfers the responsibility for accessing and traversing the elements of a collection from the collection itself to an iterator object. The iterator defines an interface for accessing collection elements and keeps track of the current element. Different iterators can carry out different traversal policies.

Enumerators

The `NSEnumerator` class in the Foundation framework implements the Iterator pattern. The private, concrete subclass of the abstract `NSEnumerator` class returns enumerator objects that sequentially traverse collections of various types—arrays, sets, dictionaries (values and keys)—returning objects in the collection to clients.

`NSDirectoryEnumerator` is a distantly related class. Instances of this class recursively enumerate the contents of a directory in the file system.

Uses and Limitations

The collection classes such as `NSArray`, `NSSet`, and `NSDictionary` include methods that return an enumerator appropriate to the type of collection. All enumerators work in the same manner. You send a `nextObject` message to the enumerator object in a loop that exits when `nil` is returned instead of the next object in the collection.

Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. These objects can thus remain more reusable.

A Mediator object centralizes complex communication and control logic between objects in a system. These objects tell a Mediator object when their state changes and, in turn, respond to requests from Mediator.

Controller Classes in the Application Kit

The Model-View-Controller design pattern assigns roles to the objects in an object-oriented system such as an application. They can be model objects, which contain the data of the application and manipulate that data; they can be view objects, which present the data and respond to user actions; or they can be controller objects, which mediate between the model and view objects. Controller objects fit the Mediator pattern.

In Cocoa, controller objects can be of two general types: mediating controllers or coordinating controllers. Mediating controllers mediate the flow of data between view objects and model objects in an application. Mediating controllers are typically `NSController` objects. Coordinating controllers implement the centralized communication and control logic for an application, acting as delegates for framework objects and as targets for action messages. They are typically `NSWindowController` objects or instances of custom `NSObject` subclasses. Because they are so highly specialized for a particular program, coordinating controllers tend not to be reusable.

The abstract class `NSController` and its concrete subclasses in the Application Kit framework are part of the Cocoa technology of bindings, which automatically synchronizes the data contained in model objects and displayed and edited in view objects. For example, if a user edits a string in a text field, bindings communicates that change—through a mediating controller—to the appropriate property of the bound model object. All programmers need to do is properly design their model objects and, using Interface Builder, establish bindings between the view, controller, and model objects of a program.

Instances of the concrete public controller classes are available on Interface Builder palette and hence are highly reusable. They provide services such as the management of selections and placeholder values. These objects perform the following specific functions:

- `NSObjectController` manages a single model object.
- `NSArrayController` manages an array of model objects and maintains a selection; it also allows you to add objects to and remove objects from the array.
- `NSTreeController` enables you add, remove, and manage model objects in a hierarchical tree structure.
- `NSUserDefaultsController` provides a convenient interface to the preferences (user defaults) system.

Uses and Limitations

Generally you use `NSController` objects as mediating controllers because these objects are designed to communicate data between the view objects and model objects of an application. To use a mediating controller, you typically drag the object from an Interface Builder palette, specify the model-object property keys, and establish the bindings between view and model objects using the Bindings pane of the Interface Builder Info window. You can also subclass `NSController` or one of its subclasses to get more specialized behavior.

You can potentially make bindings between almost any pair of objects as long as those objects comply with the `NSKeyValueCoding` and `NSKeyValueObserving` informal protocols. But it's recommended that you make bindings through mediating controllers to get all the benefits `NSController` and its subclasses give you.

Coordinating controllers centralize communication and control logic in an application by

- Maintaining outlets to model and view objects (outlets are instance variables that hold connections or references to other objects)
- Responding to user manipulations of view objects through target-action (see [“The Target-Action Mechanism”](#) (page 147))
- Acting as a delegate for messages sent by framework objects (see [“Delegation”](#) (page 149)))

You usually make all of the above connections—outlets, target-action, and delegate—in Interface Builder, which archives them in the application’s nib file.

Further Reading: See [“The Model-View-Controller Design Pattern”](#) (page 161) for a discussion of mediating controllers, coordinating controllers, and design decisions related to controllers. *Cocoa Bindings Programming Topics* describes the mediating controller classes in detail.

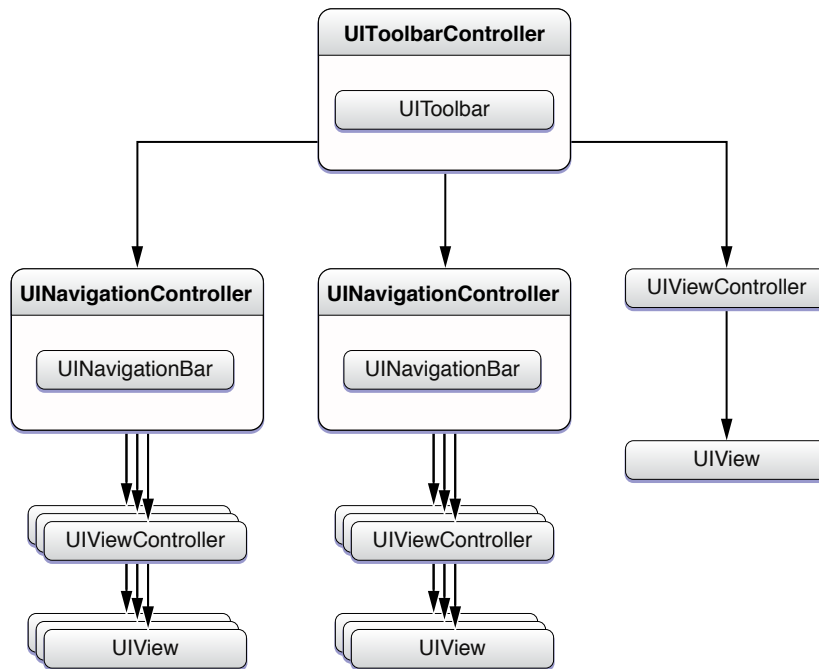
View Controllers in UIKit

Applications running on iPhone OS frequently use a modal and navigational user-interface design for presenting screen-size chunks of the application’s data model. An application may have a navigation bar and a toolbar, and between these objects is the current view of application data. Users can tap buttons on the toolbar to select a mode, tap buttons on the navigation bar, and tap controls in the current view to traverse a hierarchy of model (data) objects; at each level the central view presents more detail. At the end of this hierarchy is often an item that the user can inspect or edit. (An application, of course, is free to use just a navigation bar or just a toolbar.)

View controllers—that inherit from `UIViewController`—are central to this design. `UIViewController` is an abstract class that you can subclass to manage a particular view. The UIKit framework also provides `UIViewController` subclasses for managing navigation bar and toolbar objects: `UINavigationController` and `UITabBarController`. A toolbar controller can manage a number of navigation controllers, which in turn can manage one or more view controllers, each with its associated view object. In addition to managing views (including overlay views), a view controller specifies the buttons and titles that are displayed in the navigation bar.

To learn more about view controllers, see *View Controller Programming Guide for iPhone OS*.

Figure 4-4 View controllers in UIKit



Memento

The Memento pattern captures and externalizes an object’s internal state—without violating encapsulation—so that the object can be restored to this state later. The Memento pattern keeps the important state of a key object external from that object to maintain cohesion.

Archiving

Archiving converts the objects in a program, along with those objects’ properties (attributes and relationships) into an archive that can be stored in the file system or transmitted between processes or across a network. The archive captures the object graph of a program as an architecture-independent stream of bytes that preserves the identity of the objects and the relationships among them. Because an object’s type is stored along with its data, an object decoded from a stream of bytes is normally instantiated using the same class of the object that was originally encoded.

Uses and Limitations

Generally, you want to archive those objects in your program whose state you want to preserve. Model objects almost always fall into this category. You write an object to an archive by encoding it, and you read that object from an archive by decoding it. Encoding and decoding are operations that you perform using an `NSCoder` object, preferably using the keyed archiving technique (requiring you to invoke methods of the `NSKeyedArchiver` and `NSKeyedUnarchiver` classes). The object being encoded and decoded must conform to the `NSCoding` protocol; the methods of this protocol are invoked during archiving.

Further Reading: See *Archives and Serializations Programming Guide for Cocoa* for further information about archiving.

Property List Serialization

A property list is a simple, structured serialization of an object graph that uses only objects of the following classes: `NSDictionary`, `NSArray`, `NSString`, `NSData`, `NSDate`, and `NSNumber`. These objects are commonly referred to as property list objects. Several Cocoa framework classes offer methods to serialize these property list objects and define special formats for the data stream recording the contents of the objects and their hierarchical relationship. The `NSPropertyListSerialization` class provides class methods that serialize property list objects to and from an XML format or an optimized binary format.

Uses and Limitations

If the objects in an object graph are simple, property list serialization is a flexible, portable, and adequate means to capture and externalize an object and its state. However, this form of serialization has its limitations. It does not preserve the full class identity of objects, only the general kind (array, dictionary, string, and so on). Thus an object restored from a property list might be of a different class than its original class. This is especially an issue when the mutability of an object can vary. Property list serialization also doesn't keep track of objects that are referenced multiple times in an object, potentially resulting in multiple instances upon deserialization that was a single instance in the original object graph.

Further Reading: See *Archives and Serializations Programming Guide for Cocoa* for further information on property list serialization.

Core Data

Core Data is a Cocoa framework for Mac OS X that defines an architecture for managing object graphs and making them persistent. It is this second capability—object persistence—that makes Core Data an adaptation of the Memento pattern.

In the Core Data architecture, a central object called the managed object context manages the various model objects in an application's object graph. Below the managed object context is the persistence stack for that object graph—a collection of framework objects that mediate between the model objects and external data stores, such as XML files or relational databases. The persistence-stack objects map between data in the store and corresponding objects in the managed data context and, when there are multiple data stores, present them to the managed object context as a single aggregate store.

The design of Core Data is also heavily influenced by the Model-View-Controller and object modeling patterns.

Important: The Core Data framework was introduced in Mac OS X v10.4.

Uses and Limitations

Core Data is particularly useful in the development of enterprise applications where complex graphs of model objects must be defined, managed, and transparently archived and unarchived to and from data stores. The Xcode development environment includes project templates and design tools that reduce the programming effort required to create the two general types of Core Data applications, those that are document-based and those that are not document-based. The Interface Builder application also includes configurable Core Data framework objects on its palettes.

Further Reading: In this document, [“Other Cocoa Architectures on Mac OS X”](#) (page 241) contains a summary of Core Data. To learn more about Core Data, read *Core Data Programming Guide*. The tutorials *NSPersistentDocument Core Data Tutorial* and *Low-Level Core Data Tutorial* step you through the basic procedure for creating document-based and non-document-based Core Data applications.

Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The Observer pattern is essentially a publish-and-subscribe model in which the subject and its observers are loosely coupled. Communication can take place between the observing and observed objects without either needing to know much about the other.

Notifications

The notification mechanism of Cocoa implements one-to-many broadcast of messages according to the Observer pattern. Objects in a program add themselves or other objects to a list of observers of one or more notifications, each of which is identified by a global string (the notification name). The object that wants to notify other objects—the observed object—creates a notification object and posts it to a notification center. The notification center determines the observers of a particular notification and sends the notification to them via a message. The methods invoked by the notification message must conform to a certain single-parameter signature. The parameter of the method is the notification object, which contains the notification name, the observed object, and a dictionary containing any supplemental information.

Posting a notification is a synchronous procedure. The posting object doesn't regain control until the notification center has broadcast the notification to all observers. For asynchronous behavior, you can put the notification in a notification queue; control returns immediately to the posting object and the notification center broadcasts the notification when it reaches the top of the queue.

Regular notifications—that is, those broadcast by the notification center—are intraprocess only. If you want to broadcast notifications to other processes, you can use the distributed notification center and its related API.

Uses and Limitations

You can use notifications for a variety of reasons. For example, you could broadcast a notification to change how user-interface elements display information based on a certain event elsewhere in the program. Or you could use notifications as a way to ensure that objects in a document save their state before the document window is closed. The general purpose of notifications is to inform other objects of program events so they can respond appropriately.

But objects receiving notifications can react only after the event has occurred. This is a significant difference from delegation. The delegate is given a chance to reject or modify the operation proposed by the delegating object. Observing objects, on the other hand, cannot directly affect an impending operation.

The notification classes are `NSNotification` (for notification objects), `NSNotificationCenter` (to post notifications and add observers), `NSNotificationQueue` (to enqueue notifications), and `NSDistributedNotificationCenter`. Many Cocoa framework classes publish and post notifications that any object can observe.

Further Reading: “[Notifications](#)” (page 193) describes notification mechanism in greater detail and offers guidelines for its use.

Key-Value Observing

Key-value observing is a mechanism that allows objects to be notified of changes to specific properties of other objects. It is based on the `NSKeyValueObserving` informal protocol. Observed properties can be simple attributes, to-one relationships, or to-many relationships. In the context of the Model-View-Controller pattern, key-value observing is especially important because it enables view objects to observe—via the controller layer—changes in model objects. It is thus an essential component of the Cocoa bindings technology (see “[Controller Classes in the Application Kit](#)” (page 153)).

Cocoa provides a default “automatic” implementation of many `NSKeyValueObserving` methods that gives all complying objects a property-observing capability.

Uses and Limitations

Key-value observing is similar to the notification mechanism but is different in important respects. In key-value observing there is no central object that provides change notification for all observers. Instead, notifications of changes are directly transmitted to observing objects. Key-value observing is also directly tied to the values of specific object properties. The notification mechanism, on the other hand, is more broadly concerned with program events.

Objects that participate in key-value observing (KVO) must comply with certain requirements—or, in shorthand, be KVO-compliant. For automatic observing, this requires compliance with the requirements of key-value coding (KVC-compliance) and using the KVC-compliance methods (that is, accessor methods). Key-value coding is a related mechanism (based on a related informal protocol) for automatically getting and setting the values of object properties.

You can refine KVO notifications by disabling automatic observer notifications and implementing manual notifications using the methods of the `NSKeyValueObserving` informal protocol and associated categories.

Further Reading: See *Key-Value Observing Programming Guide* to learn more about the mechanism and underlying protocol. Also see the related documents *Key-Value Coding Programming Guide* and *Cocoa Bindings Programming Topics*.

Proxy

Provide a surrogate or placeholder for another object to control access to it. You use this pattern to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing. This pattern is structurally similar to the Decorator pattern but it serves a different purpose; Decorator adds behavior to an object whereas Proxy controls access to an object.

NSProxy

The `NSProxy` class defines the interface for objects that act as placeholders or surrogates for other objects, even for objects that don't yet exist. A proxy object typically forwards a message sent to it to the object that it represents, but it can also respond to the message by loading the represented object or transforming itself into it. Although `NSProxy` is an abstract class, it implements the `NSObject` protocol and other fundamental methods expected of a root object; it is, in fact, the root class of a hierarchy just as the `NSObject` class is.

Concrete subclasses of `NSProxy` can accomplish the stated goals of the Proxy pattern such as lazy instantiation of expensive objects or acting as sentry objects for security. `NSDistantObject`, a concrete subclass of `NSProxy` in the Foundation framework, implements a remote proxy for transparent distributed messaging. `NSDistantObject` objects are part of the architecture for distributed objects. By acting as proxies for objects in other processes or threads, they help to enable communication between objects in those threads or processes.

`NSInvocation` objects, which are an adaptation of the Command pattern, are also part of the distributed objects architecture (see [“Invocation Objects”](#) (page 146)).

Uses and Limitations

Cocoa employs `NSProxy` objects only in distributed objects. The `NSProxy` objects are specifically instances of the concrete subclasses `NSDistantObject` and `NSProtocolChecker`. You can use distributed objects not only for interprocess messaging (on the same or different computers) but you can also use it to implement distributed computing or parallel processing. If you want to use proxy objects for other purposes, such as the creation of expensive resources or security, you have to implement your own concrete subclass of `NSProxy`.

Further Reading: To learn more about Cocoa proxy objects and the role they play in distributed messaging, read *Distributed Objects Programming Topics*.

Singleton

Ensure a class only has one instance, and provide a global point of access to it. The class keeps track of its sole instance and ensures that no other instance can be created. Singleton classes are appropriate for situations where it makes sense for a single object to provide access to a global resource.

Framework Classes

Several Cocoa framework classes are singletons. They include `NSFileManager`, `NSWorkspace`, `NSApplication`, and, in `UIKit`, `UIApplication`. A process is limited to one instance of these classes. When a client asks the class for an instance, it gets a shared instance, which is lazily created upon the first request.

Uses and Limitations

Using the shared instance returned by a singleton class is no different from using an instance of a non-singleton class, except that you are prevented from copying, retaining, or releasing it (the related methods are re-implemented as null operations). You can create your own singleton classes if the situation warrants it.

Further Reading: [“Cocoa Objects”](#) (page 57) explains how to create a singleton class.

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. The Template Method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

Overridden Framework Methods

The Template Method pattern is a fundamental design of Cocoa, and indeed of object-oriented frameworks in general. The pattern in Cocoa lets custom components of a program hook themselves into an algorithm, but the framework components determine when and how they are needed. The programmatic interfaces of Cocoa classes often include methods that are meant to be overridden by subclasses. At runtime, the framework invokes these so-called generic methods at certain points in the task it is carrying out. The generic methods provide a structure for custom code to contribute program-specific behavior and data to the task being executed and coordinated by framework classes.

Uses and Limitations

To make use of the Cocoa adaptation of the Template Method pattern, you must create a subclass and override those methods that the framework invokes to insert application-specific input into the algorithm it is executing. If you are writing your own framework, you should probably include the pattern in the design.

Note: [“Adding Behavior to a Cocoa Program”](#) (page 109) discusses the Cocoa adaptation of the Template Method pattern, especially in [“Inheriting From a Cocoa Class”](#) (page 115).

Document Architecture

The document architecture defined by the Application Kit is a particular—and important—instance of the general design of overridden framework methods as an adaptation of the Template Method pattern. Cocoa applications that can create and manage multiple documents, each in its own window, are almost always based on the document architecture. In this architecture there are cooperating

objects of three framework classes: `NSDocument`, `NSWindowController`, and `NSDocumentController`. `NSDocument` objects manage the model objects that represent the data of a document; upon user requests, they write that data to files and reload the data, recreating the model objects with it. `NSWindowController` objects manage the user interface of particular documents. The `NSDocumentController` object of a document-based application tracks and manages all open documents and otherwise coordinates the activities of the application. At runtime, each of these objects receives messages from the Application Kit requesting it to perform specific operations. The application developer must override many of the methods invoked by these messages to add application-specific behavior.

The design of the document architecture of Cocoa is also heavily influenced by the Model-View-Controller pattern.

Uses and Limitations

You can create a project for a document-based Cocoa application by choosing the Cocoa Document-based Application template from the New Project assistant in Xcode. Then you must implement a custom subclass of `NSDocument` and may choose to implement custom subclasses of `NSWindowController` and `NSDocumentController`. The Application Kit provides much of the document-management logic of the application for you.

Note: In this document [“Other Cocoa Architectures on Mac OS X”](#) (page 241) contains a summary of the document architecture. For the definitive documentation for this adaptation of the Template Method pattern, see *Document-Based Applications Overview*.

The Model-View-Controller Design Pattern

The Model-View-Controller design pattern (MVC) is quite old. Variations of it have been around at least since the early days of Smalltalk. It is a high-level pattern in that it concerns itself with the global architecture of an application and classifies objects according to the general roles they play in an application. It is also a compound pattern in that it comprises several, more elemental patterns.

Object-oriented programs benefit in several ways by adapting the MVC design pattern for their designs. Many objects in these programs tend to be more reusable and their interfaces tend to be better defined. The programs overall are more adaptable to changing requirements—in other words, they are more easily extensible than programs that are not based on MVC. Moreover, many technologies and architectures in Cocoa—such as bindings, the document architecture, and scriptability—are based on MVC and require that your custom objects play one of the roles defined by MVC.

Roles and Relationships of MVC Objects

The MVC design pattern considers there to be three types of objects: model objects, view objects, and controller objects. The pattern defines the roles that these types of objects play in the application and their lines of communication. When designing an application, a major step is choosing—or creating custom classes for—objects that fall into one of these three groups. Each of the three types of objects is separated from the others by abstract boundaries and communicates with objects of the other types across those boundaries.

Model Objects Encapsulate Data and Basic Behaviors

Model objects represent special knowledge and expertise. They hold an application's data and define the logic that manipulates that data. A well-designed MVC application has all its important data encapsulated in model objects. Any data that is part of the persistent state of the application (whether that persistent state is stored in files or databases) should reside in the model objects once the data is loaded into the application. Because they represent knowledge and expertise related to a specific problem domain, they tend to be reusable.

Ideally, a model object has no explicit connection to the user interface used to present and edit it. For example, if you have a model object that represents a person (say you are writing an address book), you might want to store a birth date. That's a good thing to store in your Person model object. However, storing a date format string or other information on how that date is to be presented is probably better off somewhere else.

In practice, this separation is not always the best thing, and there is some room for flexibility here, but in general a model object should not be concerned with interface and presentation issues. One example where a bit of an exception is reasonable is a drawing application that has model objects that represent the graphics displayed. It makes sense for the graphic objects to know how to draw themselves because the main reason for their existence is to define a visual thing. But even in this case, the graphic objects should not rely on living in a particular view or any view at all, and they should not be in charge of knowing when to draw themselves. They should be asked to draw themselves by the view object that wants to present them.

Further Reading: *Model Object Implementation Guide* discusses the proper design and implementation of model objects.

View Objects Present Information to the User

A view object knows how to display and might allow users to edit the data from the application's model. The view should not be responsible for storing the data it is displaying. (This does not mean the view never actually stores data it's displaying, of course. A view can cache data or do similar tricks for performance reasons). A view object can be in charge of displaying just one part of a model object, or a whole model object, or even many different model objects. Views come in many different varieties.

View objects tend to be reusable and configurable, and they provide consistency between applications. In Cocoa, the Application Kit defines a large number of view objects and provides many of them on Interface Builder palettes. By reusing the Application Kit's view objects, such as `NSButton` objects, you guarantee that buttons in your application behave just like buttons in any other Cocoa application, assuring a high level of consistency in appearance and behavior across applications.

A view should ensure it is displaying the model correctly. Consequently, it usually needs to know about changes to the model. Because model objects should not be tied to specific view objects, they need a generic way of indicating that they have changed.

Controller Objects Tie the Model to the View

A controller object acts as the intermediary between the application's view objects and its model objects. Controllers are often in charge of making sure the views have access to the model objects they need to display and act as the conduit through which views learn about changes to the model. Controller objects can also perform set-up and coordinating tasks for an application and manage the life cycles of other objects.

In a typical Cocoa MVC design, when users enter a value or indicate a choice through a view object, that value or choice is communicated to a controller object. The controller object might interpret the user input in some application-specific way and then either may tell a model object what to do with this input—for example, "add a new value" or "delete the current record"; or it may have the model object reflect a changed value in one of its properties. Based on this same user input, some controller objects might also tell a view object to change an aspect of its appearance or behavior, such as disabling a button. Conversely, when a model object changes—say, a new data source is accessed—the model object usually communicates that change to a controller object, which then requests one or more view objects to update themselves accordingly.

Controller objects can be either reusable or non-reusable, depending on their general type. [“Types of Cocoa Controller Objects”](#) (page 164) describes the different types of controller objects in Cocoa.

Combining Roles

One can merge the MVC roles played by an object, making an object, for example, fulfill both the controller and view roles—in which case, it would be called a view-controller. In the same way, you can also have model-controller objects. For some applications, combining roles like this is an acceptable design.

A model-controller is a controller that concerns itself mostly with the model layer. It “owns” the model; its primary responsibilities are to manage the model and communicate with view objects. Action methods that apply to the model as a whole will typically be implemented in a model-controller. The document architecture provides a number of these methods for you; for example, an `NSDocument` object (which is a central part of the document architecture) automatically handles action methods related to saving files.

A view-controller is a controller that concerns itself mostly with the view layer. It “owns” the interface (the views); its primary responsibilities are to manage the interface and communicate with the model. Action methods concerned with data displayed in a view will typically be implemented in a view-controller. An `NSWindowController` object (also part of the document architecture) is an example of view-controller.

[“Design Guidelines for MVC Applications”](#) (page 167) offers some design advice concerning objects with merged MVC roles.

Further Reading: *Document-Based Applications Overview* discusses the distinction between model-controller and view-controller from another perspective.

Types of Cocoa Controller Objects

The section “[Controller Objects Tie the Model to the View](#)” (page 163) sketches the abstract outline of a controller object, but in practice the picture is far more complex. In Cocoa there are two general kinds of controller objects: mediating controllers and coordinating controllers. Each kind of controller object is associated with a different set of classes and each provides a different range of behaviors.

A **mediating controller** is typically an object that inherits from the `NSController` class. Mediating controller objects are used in the Cocoa bindings technology. They facilitate—or mediate—the flow of data between view objects and model objects.

iPhone OS Note: The Application Kit implements the `NSController` class and its subclasses. These classes and the bindings technology are not available on iPhone OS.

Mediating controllers are typically ready-made objects that you drag from Interface Builder palettes. You can configure these objects to establish the bindings between properties of view objects and properties of the controller object, and then between those controller properties and specific properties of a model object. As a result, when users change a value displayed in a view object, the new value is automatically communicated to a model object for storage—via the mediating controller; and when a property of a model changes its value, that change is communicated to a view for display. The abstract `NSController` class and its concrete subclasses—`NSObjectController`, `NSArrayController`, `NSUserDefaultsController`, and `NSTreeController`—provide supporting features such as the ability to commit and discard changes and the management of selections and placeholder values.

A **coordinating controller** is typically an `NSWindowController` or `NSDocumentController` object (available only in the Application Kit), or an instance of a custom subclass of `NSObject`. Its role in an application is to oversee—or coordinate—the functioning of the entire application or of part of the application, such as the objects unarchived from a nib file. A coordinating controller provides services such as:

- Responding to delegation messages and observing notifications
- Responding to action messages
- Managing the life cycle of “owned” objects (for example, releasing them at the proper time)
- Establishing connections between objects and performing other set-up tasks

`NSWindowController` and `NSDocumentController` are classes that are part of the Cocoa architecture for document-based applications. Instances of these classes provide default implementations for several of the services listed above, and you can create subclasses of them to implement more application-specific behavior. You can even use `NSWindowController` objects to manage windows in an application that is not based on the document architecture.

A coordinating controller frequently owns the objects archived in a nib file. As File’s Owner, it is external to the objects in the nib file and manages those objects. These owned objects include mediating controllers as well as window objects and view objects. See “[MVC as a Compound Design Pattern](#)” (page 165) for more on coordinating controllers as File’s Owner.

Instances of custom `NSObject` subclasses can be entirely suitable as coordinating controllers. These kinds of controller objects combine both mediating and coordinating functions. For their mediating behavior, they make use of mechanisms such as target-action, outlets, delegation, and notifications to facilitate the movement of data between view objects and model objects. They tend to contain a lot of "glue" code and, because that code is exclusively application-specific, they are the least reusable kind of object in an application.

Further Reading: For more on controller objects in their role as mediators, see the information on the [“Mediator”](#) (page 152) design pattern. For more on the Cocoa bindings technology, see *Cocoa Bindings Programming Topics*.

MVC as a Compound Design Pattern

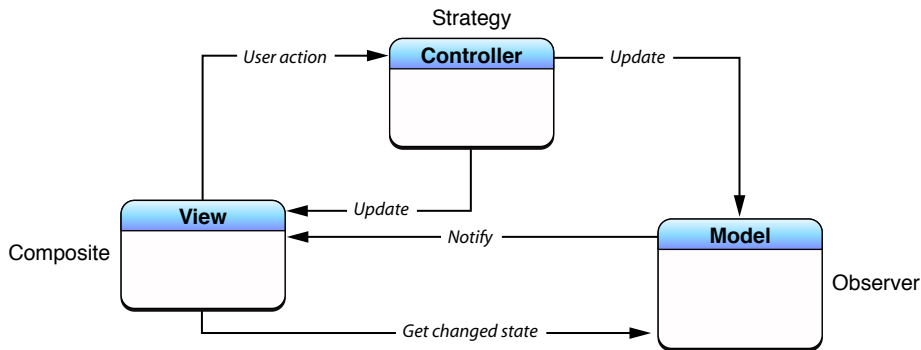
Model-View-Controller is a design pattern that is composed of several more basic design patterns. These basic patterns work together to define the functional separation and paths of communication that are characteristic of an MVC application. However, the traditional notion of MVC assigns a different set of basic patterns than does Cocoa. The difference primarily lies in the roles given to the controller and view objects of an application.

In the original (Smalltalk) conception, MVC is made up of the Composite, Strategy, and Observer patterns.

- **Composite**—The view objects in an application are actually a composite of nested views that work together in a coordinated fashion (that is, the view hierarchy). These display components range from a window to compound views, such as a table view, to individual views, such as buttons. User input and display can take place at any level of the composite structure.
- **Strategy**—A controller object implements the strategy for one or more view objects. The view object confines itself to maintaining its visual aspects and delegates to the controller all decisions about the application-specific meaning of the interface behavior.
- **Observer**—A model object keeps interested objects in an application—usually view objects—advised of changes in its state.

These patterns work together in the way depicted by Figure 4-5: The user manipulates a view at some level of the composite structured and, as a result, an event is generated. A controller object receives the event and interprets it in an application-specific way—that is, it applies a strategy. This strategy can be to request (via message) a model object to change its state or to request a view object (at some level of the composite structure) to change its behavior or appearance. The model object, in turn, notifies all objects who have registered as observers when its state changes; if the observer is a view object, it may update its appearance accordingly.

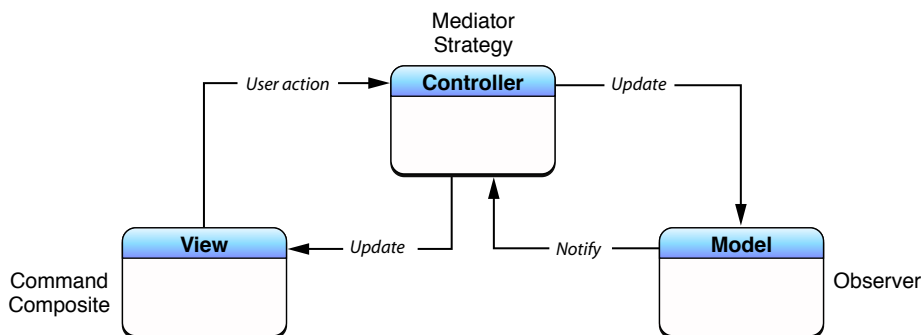
Figure 4-5 Traditional version of MVC as a compound pattern



The Cocoa version of MVC as a compound pattern has some similarities to the traditional version, and in fact it is quite possible to construct a working application based on the diagram in Figure 4-5. By using the bindings technology, you can easily create a Cocoa MVC application whose views directly observe model objects to receive notifications of state changes. However, there is a theoretical problem with this design. View objects and model objects should be the most reusable objects in an application. View objects represent the "look and feel" of an operating system and the applications that system supports; consistency in appearance and behavior is essential, and that requires highly reusable objects. Model objects by definition encapsulate the data associated with a problem domain and perform operations on that data. Design-wise, it's best to keep model and view objects separate from each other, because that enhances their reusability.

In most Cocoa applications, notifications of state changes in model objects are communicated to view objects *through* controller objects. Figure 4-6 shows this different configuration, which appears much cleaner despite the involvement of two more basic design patterns.

Figure 4-6 Cocoa version of MVC as compound design pattern



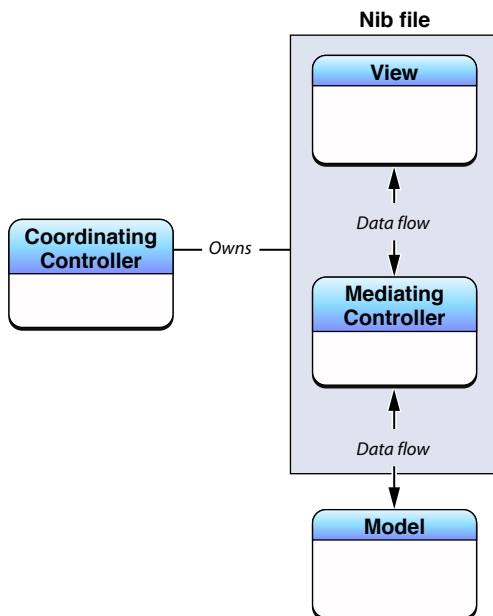
The controller object in this compound design pattern incorporates the Mediator pattern as well as the Strategy pattern; it mediates the flow of data between model and view objects in both directions. Changes in model state are communicated to view objects through the controller objects of an application. In addition, view objects incorporate the Command pattern through their implementation of the target-action mechanism.

Note: The target-action mechanism, which enables view objects to communicate user input and choices, can be implemented in both coordinating and mediating controller objects. However, the design of the mechanism differs in each controller type. For coordinating controllers, you connect the view object to its target (the controller object) in Interface Builder and specify an action selector that must conform to a certain signature. Coordinating controllers, by virtue of being delegates of windows and the global application object, can also be in the responder chain. The bindings mechanism used by mediating controllers also connects view objects to targets and allows action signatures with a variable number of arguments of arbitrary types. Mediating controllers, however, aren't in the responder chain.

There are practical reasons as well as theoretical ones for the revised compound design pattern depicted in Figure 4-6, especially when it comes to the Mediator design pattern. Mediating controllers derive from concrete subclasses of `NSController`, and these classes, besides implementing the Mediator pattern, offer many features that applications should take advantage of, such as the management of selections and placeholder values. And if you opt not to use the bindings technology, your view object could use a mechanism such as the Cocoa notification center to receive notifications from a model object. But this would require you to create a custom view subclass to add the knowledge of the notifications posted by the model object.

In a well-designed Cocoa MVC application, coordinating controller objects often "own" mediating controllers, which are archived in nib files. Figure 4-7 shows the relation of the two types of controller objects.

Figure 4-7 Coordinating controller as the owner of a nib file



Design Guidelines for MVC Applications

The following guidelines apply to Model-View-Controller considerations in the design of applications:

- Although you can use an instance of a custom subclass of `NSObject` as a mediating controller, there's no reason to go through all the work required to make it one. Use instead one of the ready-made `NSController` objects designed for the Cocoa bindings technology; that is, use an instance of `NSObjectController`, `NSArrayController`, `NSUserDefaultsController`, or `NSTreeController`—or a custom subclass of one of these concrete `NSController` subclasses.

However, if the application is very simple and you feel more comfortable writing the "glue code" needed to implement mediating behavior using outlets and target-action, feel free to use an instance of a custom `NSObject` subclass as a mediating controller. In a custom `NSObject` subclass, you can also implement a mediating controller in the `NSController` sense, using key-value coding, key-value observing, and the editor protocols

- Although you can combine MVC roles in an object, the best overall strategy is to keep the separation between roles. This separation enhances the reusability of objects and the extensibility of the program they're used in. If you are going to merge MVC roles in a class, pick a predominant role for that class and then (for maintenance purposes) use categories in the same implementation file to extend the class to play other roles.
- A goal of a well-designed MVC application should be to use as many objects as possible that are (theoretically, at least) reusable. In particular, view objects and model objects should be highly reusable. (The ready-made mediating controller objects, of course, are reusable.) Application-specific behavior is frequently concentrated as much as possible in controller objects.
- Although it is possible to have views directly observe models to detect changes in state, it is not recommended that they do so. A view object should always go through a mediating controller object to learn about changes in an model object. The intent is two-fold:
 - ❑ If you use the bindings mechanism to have view objects directly observe the properties of model objects, you bypass all the advantages `NSController` and its subclasses give your application: selection and placeholder management as well as the ability to commit and discard changes.
 - ❑ If you don't use the bindings mechanism, you have to subclass an existing view class to add the ability to observe change notifications posted by a model object.
- Strive to limit code dependency in the classes of your application. The greater the dependency a class has on another class, the less reusable it is. Specific recommendations vary by the MVC roles of the two classes involved:
 - ❑ A view class shouldn't depend on a model class (although this may be unavoidable with some custom views).
 - ❑ A view class shouldn't have to depend on a mediating controller class.
 - ❑ A model class shouldn't depend on anything other than other model classes.
 - ❑ A mediating controller class should not depend on a model class (although, like views, this may be necessary if it's a custom controller class).
 - ❑ A mediating controller class shouldn't depend on view classes or on coordinating controller classes.
 - ❑ A coordinating controller class depends on classes of all MVC role types.
- If Cocoa offers an architecture that assigns MVC roles to objects of specific types, use that architecture. It will be much easier to put your project together if you do. The document architecture, for example, includes an Xcode project template that configures an `NSDocument` object (per-nib controller-model) as File's Owner.

Model-View-Controller in Cocoa (Mac OS X)

The Model-View-Controller design pattern is fundamental to many Cocoa mechanisms and technologies. As a consequence, the importance of using MVC in object-oriented design goes beyond attaining greater reusability and extensibility for your own applications. If your application is to incorporate a Cocoa technology that is MVC-based, your application will work best if its design also follows the MVC pattern. It should be relatively painless to use these technologies if your application has a good MVC separation, but it will take more effort to use such a technology if you don't have a good separation.

Cocoa includes the following architectures, mechanisms, and technologies that are based on Model-View-Controller:

- **Document architecture.** In this architecture, a document-based application consists of a controller object for the entire application (`NSDocumentController`), a controller object for each document window (`NSWindowController`), and an object that combines controller and model roles for each document (`NSDocument`).
- **Bindings.** As has been indicated in the foregoing discussion, MVC is central to the bindings technology of Cocoa. The concrete subclasses of the abstract `NSController` provide ready-made controller objects that you can configure to establish bindings between view objects and properly designed model objects.
- **Application scriptability.** When designing an application to make it scriptable, it is essential not only that it follow the MVC design pattern but that your application's model objects are properly designed. Scripting commands that access application state and request application behavior should usually be sent to model objects or controller objects.
- **Core Data.** The Core Data framework manages graphs of model objects and ensures the persistence of those objects by saving them to (and retrieving them from) a persistent store. Core Data is tightly integrated with the Cocoa bindings technology. The MVC and object modeling design patterns are essential determinants of the Core Data architecture.
- **Undo.** In the undo architecture, model objects once again play a central role. The primitive methods of model objects (which are often its accessor methods) are often where you implement undo and redo operations. The view and controller objects of an action may also be involved in these operations; for example, you might have such objects give specific titles to the undo and redo menu items, or you might have them undo selections in a text view.

Further Reading: “[Other Cocoa Architectures on Mac OS X](#)” (page 241) provides overviews of the document architecture, application scriptability, and Core Data and includes cross-references to the documents describing these technologies in detail.

Object Modeling

This section defines terms and presents examples of object modeling and key-value coding that are specific to Cocoa bindings and the Core Data framework. Understanding terms such as key paths is fundamental to using these technologies effectively. This section is recommended reading if you are new to object-oriented design or key-value coding.

When using the Core Data framework—which is available only on Mac OS X—you need a way to describe your model objects that does not depend on views and controllers. In a good reusable design, views and controllers need a way to access model properties without imposing dependencies between them. The Core Data framework solves this problem by borrowing concepts and terms from database technology—specifically, the entity-relationship model.

Entity-relationship modeling is a way of representing objects typically used to describe a data source's data structures in a way that allows those data structures to be mapped to objects in an object-oriented system. Note that entity-relationship modeling isn't unique to Cocoa; it's a popular discipline with a set of rules and terms that are documented in database literature. It is a representation that facilitates storage and retrieval of objects in a data source. A data source can be a database, a file, a web service, or any other persistent store. Because it is not dependent on any type of data source it can also be used to represent any kind of object and its relationship to other objects.

Cocoa uses a modified version of the traditional rules of entity-relationship modeling referred to in this article as **object modeling**. Object modeling is particularly useful in representing model objects in the Model-View-Controller (MVC) design pattern. This is not surprising since even in a simple Cocoa application, models are typically persistent—stored in some kind of data container, for example, a file.

Entities

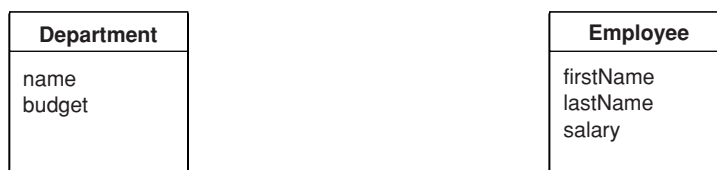
In the MVC design pattern, models are the objects in your application that encapsulate specified data and provide methods that operate on that data. Models are usually persistent but more importantly, models are not dependent on how the data is displayed to the user.

In the entity-relationship model, models are called **entities**, the components of an entity are called **attributes**, and the references to other models are called **relationships**. Together, attributes and relationships are known as **properties**. With these three simple components (entities, attributes, and relationships), arbitrarily complex systems can be modeled.

For example, an object model can be used to depict a company's customer base, a library of books, or a network of computers. A library book has attributes—such as the book title, ISBN number, and copyright date—and relationships to other objects—such as the author and library member. In theory, if the parts of a system can be identified, the system can be expressed as an object model.

Figure 4-8 shows an example object model used in an employee management application. In this model, a Department models a department and Employee models an employee.

Figure 4-8 Employee management application object diagram

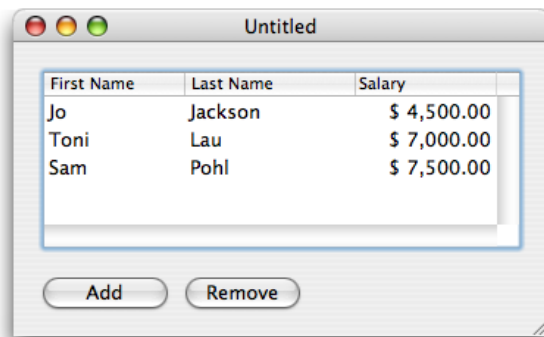


Attributes

Attributes represent structures that contain data. An attribute of an object may be a simple value, such as a scalar (for example, an integer, float, or double), but can also be a C structure (for example an array of char or an `NSPoint`) or an instance of a primitive class (such as, `NSNumber`, `NSData`, or `NSColor` in Cocoa). Immutable objects such as `NSColor` are usually considered attributes too. (Note that Core Data natively supports only a specific set of attribute types, as described in "NSAttributeDescription". You can, however, use additional attribute types, as described in "Non-Standard Attributes" in *Core Data Programming Guide*.)

In Cocoa, an attribute typically corresponds to a model's instance variable or accessor method. For example, `Employee` has `firstName`, `lastName`, and `salary` instance variables. In an employee management application, you might implement a table view to display a collection of `Employee` objects and some of their attributes, as shown in Figure 4-9. Each row in the table corresponds to an instance of `Employee`, and each column corresponds to a attribute of `Employee`.

Figure 4-9 Employees table view



Relationships

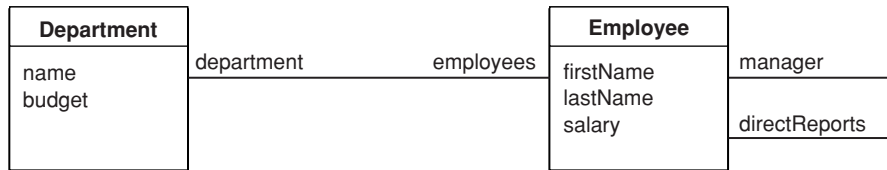
Not all properties of a model are attributes—some properties are **relationships** to other objects. Your application is typically modeled by multiple classes. At runtime, your object model is a collection of related objects that make up an **object graph**. These are typically the persistent objects that your users create and save to some data container or file before terminating the application (as in a document-based application). The relationships between these model objects can be traversed at runtime to access the properties of the related objects.

For example, in the employee management application, there are relationships between an employee and the department in which they work, and between an employee and the employee's manager. The latter is an example of a **reflexive relationship**—a relationship from an entity to itself.

Relationships are inherently bidirectional, so conceptually at least there are also relationships between a department and the employees that work in the department, and an employee and their direct reports. Figure 4-10 (page 172) illustrates the relationships between a `Department` and an `Employee` entity, and the `Employee` reflexive relationship. In this example, the `Department` entity's "employees" relationship is the **inverse** of the `Employee` entity's "department" relationship. It is possible, however, for relationships to be navigable in only one direction—for there to be no inverse relationship. If, for example, you are never interested in finding out from a department object what employees are

associated with it, then you do not have to model that relationship. (Note that although this is true in the general case, Core Data may impose additional constraints over general Cocoa object modeling—not modeling the inverse should be considered an extremely advanced option.)

Figure 4-10 Relationships in the employee management application



Relationship Cardinality and Ownership

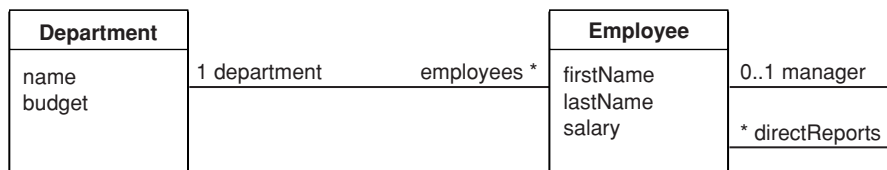
Every relationship has a **cardinality**; the cardinality tells you how many destination objects can (potentially) resolve the relationship. If the destination object is a single object, then the relationship is called a **to-one relationship**. If there may be more than one object in the destination, then the relationship is called a **to-many relationship**.

Relationships can be **mandatory** or **optional**. A mandatory relationship is one where the destination is required, for example, every employee must be associated with a department. An optional relationship is, as the name suggests, optional—for example, not every employee has direct reports.

It is also possible to specify a range for the cardinality. An optional to-one relationship has a range 0-1. An employee may have an arbitrary number of direct reports, or a range that specifies a minimum and a maximum, for example, 0-15, which also illustrates an optional to-many relationship.

Figure 4-11 illustrates the cardinalities in the employee management application. The relationship between an Employee object and a Department object is a mandatory to-one relationship—an employee must belong to one, and only one, department. The relationship between a Department and its Employee objects is an optional to-many relationship (represented by a “*”). The relationship between an employee and a manager is an optional to-one relationship (denoted by the range 0-1)—top-ranking employees do not have managers.

Figure 4-11 Relationship cardinality



Note also that destination objects of relationships are sometimes owned and sometimes shared.

Accessing Properties

In order for models, views, and controllers to be independent of each other, you need to be able to access properties in a way that is independent of a model’s implementation. This is accomplished by using key-value pairs.

Keys

You specify properties of a model using a simple **key**, often a string. The corresponding view or controller uses the key to look up the corresponding attribute **value**. The “value for an attribute” construction enforces the notion that the attribute itself doesn’t necessarily contain the data—the value can be indirectly obtained or derived.

Key-value coding is used to perform this lookup—it is a mechanism for accessing an object’s properties indirectly and, in certain contexts, automatically. Key-value coding works by using the names of the object’s properties—typically its instance variables or accessor methods—as keys to access the values of those properties.

For example, you might obtain the name of a `Department` object using a `name` key. If the `Department` object either has an instance variable or method called `name` then a value for the key can be returned (if it doesn’t, this results in an error). Similarly, you might obtain `Employee` attributes using the `firstName`, `lastName`, and `salary` keys.

Values

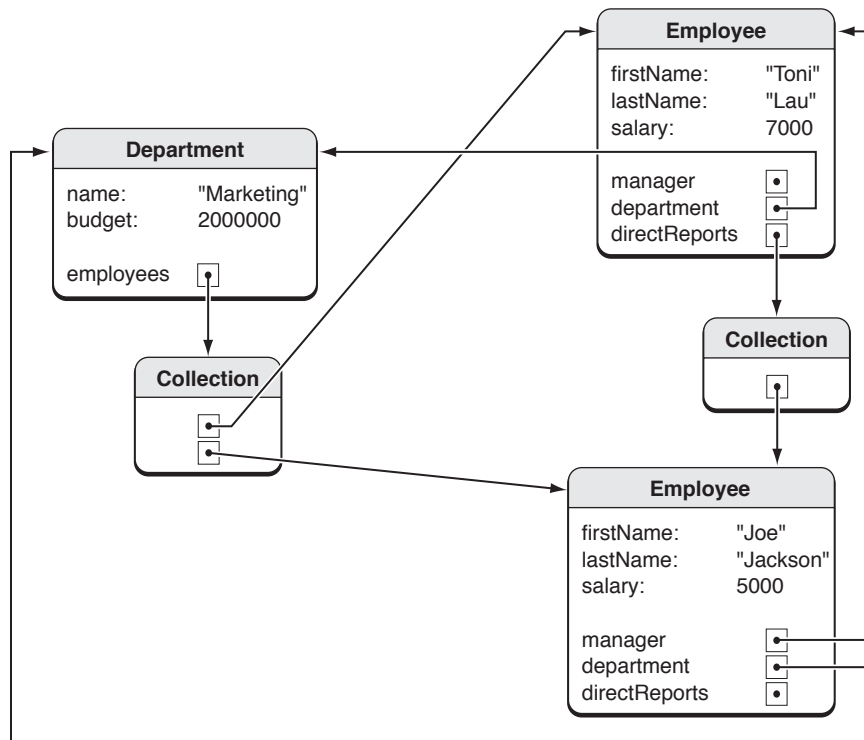
All values for a particular attribute of a given entity are of the same data type. The data type of an attribute is specified in the declaration of its corresponding instance variable or in the return value of its accessor method. For example, the data type of the `Department` object `name` attribute may be an `NSString` object in Objective-C.

Note that key-value coding only returns object values. If the return type or the data type for the specific accessor method or instance variable used to supply the value for a specified key is not an object, then an `NSNumber` or `NSValue` object is created for that value and returned in its place. If the `name` attribute of `Department` is of type `NSString`, then using key-value coding the value returned for the `name` key of a `Department` object is an `NSString` object. If the `budget` attribute of `Department` is of type `float`, then using key-value coding the value returned for the `budget` key of a `Department` object is an `NSNumber` object.

Similarly, when you set a value using key-value coding, if the data type required by the appropriate accessor or instance variable for the specified key is not an object, then the value is extracted from the passed object using the appropriate `-<type>Value` method.

The value of a to-one relationship is simply the destination object of that relationship. For example, the value of the `department` property of an `Employee` object is a `Department` object. The value of a to-many relationship is the collection object (it may be a set or an array—if you use Core Data it is a set otherwise it is typically an array) that contains the destination objects of that relationship. For example, the value of the `employees` property of an `Department` object is a collection containing `Employee` objects. Figure 4-12 shows an example object graph for the employee management application.

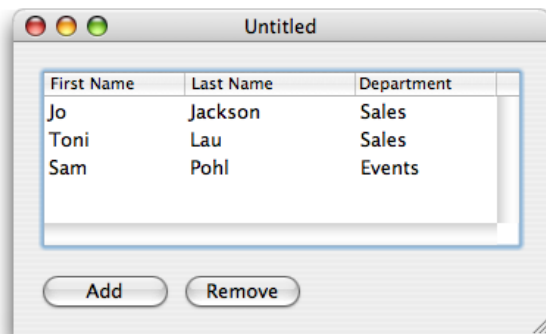
Figure 4-12 Object graph for the employee management application



Key Paths

A **key path** is a string of dot-separated keys that specify a sequence of object properties to traverse. The property of the first key is determined by, and each subsequent key is evaluated relative to, the previous property. Key paths allow you to specify the properties of related objects in a way that is independent of the model implementation. Using key paths you can specify the path through an object graph, of arbitrary depth, to a specific attribute of a related object.

The key-value coding mechanism implements the lookup of a value given a key path similar to key-value pairs. For example, in the accounting application you might access the name of a **Department** via an **Employee** object using the `department.name` key path where `department` is a relationship of **Employee** and `name` is an attribute of **Department**. Key paths are useful if you want to display an attribute of a destination entity. For example, the employee table view in Figure 4-13 is configured to display the name of the employee's department object, not the department object itself. Using Cocoa bindings, the value of the "Department" column is bound to `department.name` of the employee objects in the displayed array.

Figure 4-13 Employees table view showing department name

Not every relationship in a key path necessarily has a value. For example, the `manager` relationship can be `nil` if the employee is the CEO. In this case, the key-value coding mechanism does not break—it simply stops traversing the path and returns an appropriate value, such as `nil`.

Communicating With Objects

Several of the Cocoa adaptations of design patterns assist communication between objects in an application. These mechanisms and paradigms include delegation, notification, target-action, and the bindings technology. This chapter describes those mechanisms and paradigms.

Communication in Object-Oriented Programs

With Cocoa and Objective-C, the object-oriented language for Cocoa, one way of adding the behavior that is specific to your program is through inheritance. You create a subclass of an existing class that either augments the attributes and behavior of an instance of that class or modifies them in some way. But there are other ways of adding the special logic that characterizes your program. There are other mechanisms for reusing and extending the capabilities of Cocoa objects.

The relationships between objects in a program exist in more than one dimension. There is the hierarchical structure of inheritance, but objects in a program also exist dynamically, in a network of other objects that must communicate with one another at runtime to get the work of the program done. In a fashion similar to a musician in orchestra, each object in a program has a role, a limited set of behaviors it contributes to the program. It displays an oval surface that responds to mouse clicks, or it manages a collection of objects, or it coordinates the major events in the life of a window. It does what it is designed to do, and nothing more. But for its contributions to be realized in the program, it must be able to communicate them to other objects. It must be able to send messages to other objects or be able to receive messages from other objects.

Before your object can send a message to another object, it must either have a reference to it or have some delivery mechanism it can rely on. Cocoa gives objects many ways to communicate with each other. These mechanisms and techniques, which are based on design patterns described in “[Cocoa Design Patterns](#)” (page 141), make it possible to construct robust applications efficiently. They range from the simple to the slightly more elaborate, and often are a preferable alternative to subclassing. You can configure them programmatically and sometimes graphically in Interface Builder.

Outlets

Object composition is a dynamic pattern that requires an object to somehow acquire references to its constituent objects so that it can send messages to them. It typically holds these other objects as instance variables. These variables must be initialized with the appropriate references at some point during the execution of the program.

An outlet is such an object instance variable, but it is special in that the reference is configured and archived through Interface Builder. The connections between the containing object and its outlets are reestablished every time the containing object is unarchived from its nib file. The containing object holds an outlet as an instance variable with the type qualifier of `IBOutlet`. For example:

```
@interface ApplicationController : NSObject
{
    IBOutlet NSArray *keywords;
}
```

Because it is an instance variable, an outlet becomes part of an object's encapsulated data. But an outlet is more than a simple instance variable. The connection between an object and its outlets is archived in a nib file; when the nib file is loaded, each connection is unarchived and reestablished, and is thus always available whenever it becomes necessary to send messages to the other object. The type qualifier `IBOutlet` is a tag applied to an instance-variable declaration so that the Interface Builder application can recognize the instance variable as an outlet and synchronize the display and connection of it with Xcode.

You connect an outlet in Interface Builder, but the procedure starts in Xcode. The following steps describe the procedure:

1. When defining your custom class, declare an outlet by tagging the instance variable with the `IBOutlet` qualifier.
2. In Interface Builder, drag a generic object into the top level of the nib file window (if one doesn't already exist for your custom class).

If an instance of your custom class is to be the File's Owner for the nib file, this step is not necessary. Also, if you're defining a custom `NSView` object, select that object instead.

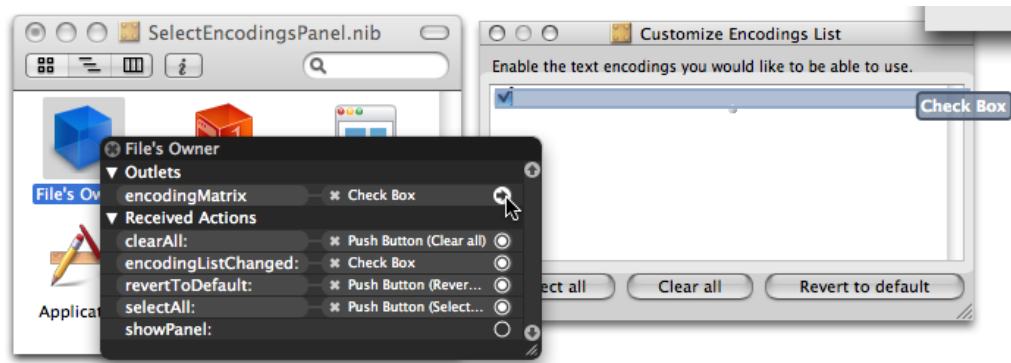
3. Import the custom class into Interface Builder.

With the generic object (or File's Owner) selected, type the name of your custom class in the Class field of the Identify pane of the Interface Builder inspector. This assigns your custom class as the class of the selected object. This step has to be done only once.

4. Select your custom instance (or File's Owner).
5. Right-click or Control-click this object to display the connections panel.
6. Find your outlet under Outlets and drag a connection line *from* the circle next to the outlet *to* the object on the user interface that you want that outlet to reference.

Figure 5-1 illustrates what an outlet connection looks like in the connections panel when you've completed these steps. (Note that the circle next to the outlet is filled in and the type of the object is specified.)

Figure 5-1 Connecting an outlet in Interface Builder



An application typically sets outlet connections between its custom controller objects and objects on the user interface, but they can be made between any objects that can be represented as instances in Interface Builder, even between two custom objects. As with any instance variable, you should be able to justify its inclusion in a class; the more instance variables an object has, the more memory it takes up. If there are other ways to obtain a reference to an object, such as finding it through its index position in a matrix, or through its inclusion as a function parameter, or through use of a tag (an assigned numeric identifier), you should do that instead.

Delegates and Data Sources

A delegate is an object that acts on behalf of, or in coordination with, another object when that object encounters an event in a program. The delegating object is often a responder object—that is, an object inheriting from `NSResponder` in Application Kit or `UIResponder` in UIKit—that is responding to a user event. The delegate is an object that is delegated control of the user interface for that event, or is at least asked to interpret the event in an application-specific manner.

To better appreciate the value of delegation, it helps to consider an off-the-shelf Cocoa object such as a window (an instance of `NSWindow` or `UIWindow`) or a table view (an instance of `NSTableView` or `UITableView`). These objects are designed to fulfill a specific role in a generic fashion; a window object in the Application Kit, for example, responds to mouse manipulations of its controls and handles such things as closing, resizing, and moving the physical window. This restricted and generic behavior necessarily limits what the object can know about how an event affects (or will affect) something elsewhere in the application, especially when the affected behavior is specific to your application. Delegation provides a way for your custom object to communicate application-specific behavior to the off-the-shelf object.

The programming mechanism of delegation gives objects a chance to coordinate their appearance and state with changes occurring elsewhere in a program, changes usually brought about by user actions. More importantly, delegation makes it possible for one object to alter the behavior of another object without the need to inherit from it. The delegate is almost always one of your custom objects, and by definition it incorporates application-specific logic that the generic and delegating object cannot possibly know itself.

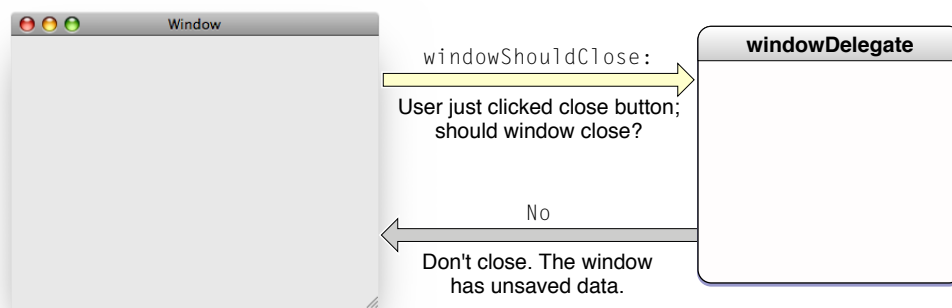
How Delegation Works

The design of the delegation mechanism is simple (Figure 5-2). The delegating class has an outlet or property, usually one that is named `delegate`; if it is an outlet, it includes methods for setting and accessing the value of the outlet. It also declares, without implementing, one or more methods that constitute a formal protocol or an informal protocol. A formal protocol that uses optional methods—a new feature of Objective-C 2.0—is the preferred approach, but both kinds of protocols are used by the Cocoa frameworks for delegation.

In the informal protocol approach, the delegating class declares methods on a category of `NSObject`, and the delegate implements only those methods where it has an interest in coordinating itself with the delegating object or affecting that object's default behavior. If the delegating class declares a formal protocol, the delegate may choose to implement those methods marked optional, but it must implement the required ones.

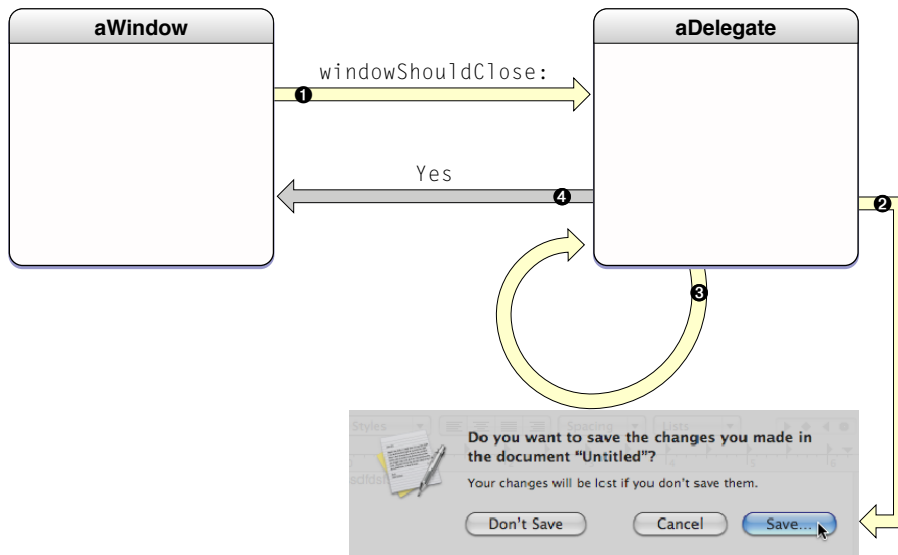
After this detail, delegation follows a common design, illustrated by Figure 5-2.

Figure 5-2 The mechanism of delegation



The methods of the protocol mark significant events handled or anticipated by the delegating object. This object wants either to communicate these events to the delegate or, for impending events, to request input or approval from the delegate. For example, when a user clicks the close button of a window in Mac OS X, the window object sends the `windowShouldClose:` message to its delegate; this gives the delegate the opportunity to veto or defer the closing of the window if, for example, the window has associated data that must be saved (see Figure 5-3).

Figure 5-3 A more realistic sequence involving a delegate



The delegating object sends a message only if the delegate implements the method. It makes this discovery by invoking the `NSObject` method `respondsToSelector:` in the delegate first.

The Form of Delegation Messages

Delegation methods have a conventional form. They begin with the name of the Application Kit or UIKit object doing the delegating—application, window, control, and so on; this name is in lower-case and without the “NS” or “UI” prefix. Usually (but not always) this object name is followed by an auxiliary verb indicative of the temporal status of the reported event. This verb, in other words, indicates whether the event is about to occur (“Should” or “Will”) or whether it has just occurred (“Did” or “Has”). This temporal distinction helps to categorize those messages that expect a return value and those that don’t. Listing 5-1 includes a few Application Kit delegation methods that expect a return value.

Listing 5-1 Sample delegation methods with return values

```
- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename; //
NSApplication
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url; //
UIApplicationDelegate
- (NSApplicationTerminateReply)applicationShouldTerminate:(NSApplication *)sender; //
NSApplication
- (UITableViewIndexSet *)tableView:(NSTableView *)tableView
willSelectRows:(UITableViewIndexSet *)selection; // UITableViewDelegate
- (NSRect>windowWillUseStandardFrame:(NSWindow *)window defaultFrame:(NSRect)newFrame;
// NSWindow
```

The delegate that implements these methods can block the impending event (by returning `NO` in the first two methods), defer the impending event (by returning `NSTerminateLater` in the `applicationShouldTerminate:` method), or alter a suggested value (the frame rectangle and the index set in the last two methods).

Other delegation methods are invoked by messages that don't expect a return value and so are typed to return `void`. These messages are purely informational, and the method names often contain "Did" or some other indication of a transpired event. Listing 5-2 shows a few examples of this kind of delegation method.

Listing 5-2 Sample delegation methods returning `void`

```
- (void)tableView:(NSTableView*)tableView mouseDownInHeaderOfTableColumn:(NSTableColumn
*)tableColumn; // NSTableView
- (void>windowDidMove:(NSNotification *)notification; // NSWindow
- (void)application:(UIApplication *)application
willChangeStatusBarFrame:(CGRect)newStatusBarFrame; // UIApplication
- (void)applicationWillBecomeActive:(NSNotification *)notification; // NSApplication
```

There are couple of things to note about this last group of messages. The first is that an auxiliary verb of "Will" (as in the third method) does not necessarily mean that a return value is expected. In this case, the event is imminent and cannot be blocked, but the message gives the delegate an opportunity to prepare the program for the event.

The other point of interest concerns the second and last method declarations in Listing 5-2. The sole parameter of each of these methods is an `NSNotification` object, which means that these methods are invoked as the result of the posting of a particular notification. For example, the `windowDidMove:` method is associated with the `NSWindow` method `NSWindowDidMoveNotification`. The section "Notifications" (page 193) discusses notifications in detail, but here it's important to understand the relation of notifications to delegation messages in the Application Kit. The delegating object automatically makes its delegate an observer of all notifications it posts. All the delegate needs to do is implement the associated method to get the notification.

To make an instance of your custom class the delegate of an Application Kit object, simply connect the instance to the `delegate` outlet or property in Interface Builder. Or you can set it programmatically through the delegating object's `setDelegate:` method or `delegate` property, preferably early on, such as in the `awakeFromNib` or `applicationDidFinishLaunching:` method.

Delegation and the Cocoa Application Frameworks

The delegating object in a Cocoa application is often a responder object such as a `UIApplication`, `NSWindow`, or `NSTableView` object. The delegate object itself is typically, but not necessarily, an object, often a custom object, that controls some part of the application (that is, a coordinating controller object). Table 5-1 lists the Application Kit classes that define a delegate.

Table 5-1 Application Kit classes with delegates

<code>NSApplication</code>	<code>NSFontManager</code>	<code>NSSplitView</code>	<code>NSTextField</code>
<code>NSBrowser</code>	<code>NSFontPanel</code>	<code>NSTableView</code>	<code>NSTextView</code>
<code>NSControl</code>	<code>NSMatrix</code>	<code>NSTabView</code>	<code>NSWindow</code>
<code>NSDrawer</code>	<code>NSOutlineView</code>	<code>NSText</code>	

The UIKit framework also uses delegation extensively and always implements it using formal protocols. The application delegate is extremely important in an application running on iPhone OS, as it must respond to application-launch, application-quit, low-memory, and other messages from the application object. The application delegate must adopt the `UIApplicationDelegate` protocol.

Delegating objects do not (and should not) retain their delegates. However, clients of delegating objects (applications, usually) are responsible for ensuring that their delegates are around to receive delegation messages. To do this, they may have to retain the delegate in memory-managed code. This precaution applies equally to data sources, notification observers, and targets of action messages.

Some Application Kit classes have a more restricted type of delegate called a modal delegate. Objects of these classes (`NSOpenPanel`, for example) run modal dialogs that invoke a handler method in the designated delegate when the user clicks the dialog's OK button. Modal delegates are limited in scope to the operation of the modal dialog.

The existence of delegates has other programmatic uses. For example, with delegates it is easy for two coordinating controllers in the same program to find and communicate with each other. For example, the object controlling the application overall can find the controller of the application's inspector window (assuming it's the current key window) using code similar to the following:

```
id winController = [[NSApp keyWindow] delegate];
```

And your code can find the application-controller object—by definition, the delegate of the global application instance—by doing something similar to the following:

```
id appController = [NSApp delegate];
```

Data Sources

A data source is like a delegate except that, instead of being delegated control of the user interface, it is delegated control of data. A data source is an outlet held by `NSView` and `UIView` objects such as table views and outline views that require a source from which to populate their rows of visible data. The data source for a view is usually the same object that acts as its delegate, but it can be any object. As with the delegate, the data source must implement one or more methods of an informal protocol to supply the view with the data it needs and, in more advanced implementations, to handle data that users directly edit in such views.

As with delegates, data sources are objects that must be present to receive messages from the objects requesting data. The application that uses them must ensure their persistence, retaining them if necessary in memory-managed code.

Data sources are responsible for the persistence of the objects they hand out to user-interface objects. In other words, they are responsible for the memory management of those objects. However, whenever a view object such as an outline view or table view accesses the data from a data source, it retains the objects as long as it uses the data. But it does not use the data very long. Typically it holds on to the data only long enough to display it.

Implementing a Delegate for a Custom Class

To implement a delegate for your custom class, complete the following steps:

- Declare the delegate accessor methods in your class header file.

```
- (id)delegate;
- (void)setDelegate:(id)newDelegate;
```

- Implement the accessor methods. In a memory-managed program, to avoid retain cycles, the setter method should not retain or copy your delegate.

```
- (id)delegate {
    return delegate;
}

- (void)setDelegate:(id)newDelegate {
    delegate = newDelegate;
}
```

In a garbage-collected environment, where retain cycles are not a problem, you should not make the delegate a weak reference (by using the `__weak` type modifier). For more on retain cycles, see “Object Ownership and Disposal” in *Memory Management Programming Guide for Cocoa*. For more on weak references in garbage collection, see “Garbage Collection for Cocoa Essentials.”

- Declare a formal or informal protocol containing the programmatic interface for the delegate. Informal protocols are categories on the `NSObject` class. If you declare a formal protocol for your delegate, make sure you mark groups of optional methods with the `@optional` directive.

“[The Form of Delegation Messages](#)” (page 181) gives advice for naming your own delegation methods.

- Before invoking a delegation method make sure the delegate implements it by sending it a `respondsToSelector:` message.

```
- (void)someMethod {
    if ( [delegate respondsToSelector:@selector(operationShouldProceed)] ) {
        if ( [delegate operationShouldProceed] ) {
            // do something appropriate
        }
    }
}
```

The Target-Action Mechanism

While delegation, bindings, and notification are useful for handling certain forms of communication between objects in a program, they are not particularly suitable for the most visible sort of communication. A typical application's user interface consists of a number of graphical objects, and perhaps the most common of these objects are controls. A control is a graphical analogue of a real-world or logical device (button, slider, check boxes, and so on); as with a real-world control, such as a radio tuner, you use it to convey your intent to some system of which it is a part—in our case, an application.

The role of a control on a user interface is simple: It interprets the intent of the user and instructs some other object to carry out that request. When a user acts on the control by, say, clicking it or pressing the Return key, the hardware device generates a raw event. The control accepts the event (as appropriately packaged for Cocoa) and translates it into an instruction that is specific to the application. However, events by themselves don't give much information about the user's intent; they merely tell you that the user clicked a mouse button or pressed a key. So some mechanism must be called upon to provide the translation between event and instruction. This mechanism is called target-action.

Cocoa uses the target-action mechanism for communication between a control and another object. This mechanism allows the control and, in Mac OS X, its cell or cells, to encapsulate the information necessary to send an application-specific instruction to the appropriate object. The receiving object—typically an instance of a custom class—is called the target. The action is the message that the control sends to the target. The object that is interested in the user event—the target—is the one that imparts significance to it, and this significance is usually reflected in the name it gives to the action.

The Target

A target is a receiver of an action message. A control or, more frequently, its cell holds the target of its action message as an outlet (see “Outlets” (page 178)). The target usually is an instance of one of your custom classes, although it can be any Cocoa object whose class implements the appropriate action method.

You can also set a cell's or control's target outlet to `nil` and let the target object be determined at run time. When the target is `nil`, the application object (`NSApplication` or `UIApplication`) searches for an appropriate receiver in a prescribed order:

1. It begins with the first responder in the key window and follows `nextResponder` links up the responder chain to the window object's (`NSWindow` or `UIWindow`) content view.

Note: A key window in Mac OS X responds to key presses for an application and is the receiver of messages from menus and dialogs. An application's main window is the principal focus of user actions and often has key status as well.

2. It tries the window object and then the window object's delegate.
3. If the main window is different from the key window, it then starts over with the first responder in the main window and works its way up the main window's responder chain to the window object and its delegate.
4. Next, the application object tries to respond itself. If it can't respond, it tries its own delegate. The application object and its delegate are the receivers of last resort.

Control objects do not (and should not) retain their targets. However, clients of controls sending action messages (applications, usually) are responsible for ensuring that their targets are available to receive action messages. To do this, they may have to retain their targets in memory-managed environments. This precaution applies equally to delegates and data sources.

The Action

An action is the message a control sends to the target or, from the perspective of the target, the method it implements to respond to the action. A control or—as is frequently the case in the Application Kit—a control's cell stores an action as an instance variable of type `SEL`. `SEL` is an Objective-C data type used to specify the signature of a message. An action message must have a simple, distinct signature. The method it invokes returns nothing and has a sole argument of type `id`. This argument, by convention, is named `sender`. Here is an example from the `NSResponder` class, which defines a number of action methods:

```
- (void)capitalizeWord:(id)sender;
```

Action methods declared by Cocoa classes can also have the equivalent signature:

```
- (IBAction) deleteRecord:(id)sender;
```

In this case, `IBAction` does not designate a data type for a return value; no value is returned. `IBAction` is a type qualifier that Interface Builder notices during application development to synchronize actions added programmatically with its internal list of action methods defined for a project.

iPhone OS Note: In UIKit, action selectors can also take two other forms. See [“Target-Action in UIKit”](#) (page 189) for details.

The `sender` parameter usually identifies the control sending the action message (although it can be another object substituted by the actual sender). The idea behind this is similar to a return address on a postcard. The target can query the sender for more information if it needs to. If the actual sending object substitutes another object as sender, you should treat that object in the same way. For example, say you have a text field and when the user enters text, the (arbitrarily named) action method `nameEntered:` is invoked in the target:

```
- (void)nameEntered:(id) sender {
    NSString *name = [sender stringValue];
    if (![name isEqualToString:@""]) {
        NSMutableArray *names = [self nameList];
        [names addObject:name];
        [sender setStringValue:@""];
    }
}
```

Here the responding method extracts the contents of the text field, adds the string to an array cached as an instance variable, and clears the field. Other possible queries to sender would be asking an `NSMatrix` object for its selected row (`[sender selectedRow]`), asking an `NSButton` object for its state (`[sender state]`), and asking any cell associated with a control for its tag (`[[sender cell] tag]`), a tag being an arbitrary identifier.

Target-Action in the Application Kit

Controls, Cells, and Menu Items

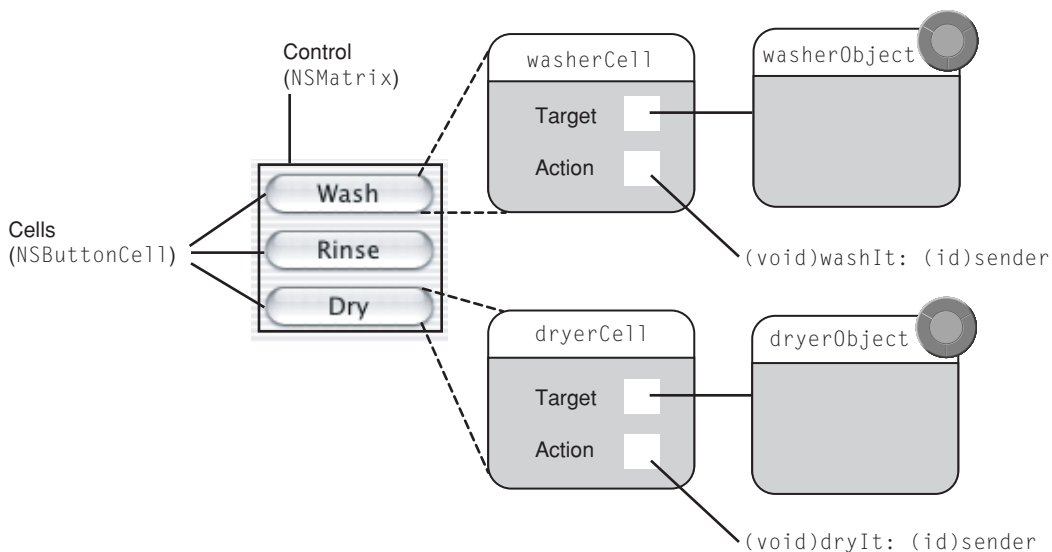
Most controls in the Application Kit are objects that inherit from the `NSControl` class. Although a control has the initial responsibility for sending an action message to its target, it rarely carries the information needed to send the message. For this, it usually relies on its cell or cells.

A control almost always has one or more cells—objects that inherit from `NSCell`—associated with it. Why is there this association? A control is a relatively “heavy” object because it inherits all the combined instance variables of its ancestors, which include the `NSView` and `NSResponder` classes. Because controls are expensive, cells are used to subdivide the screen real estate of a control into various functional areas. Cells are lightweight objects that can be thought of as overlaying all or part of the control. But it's not only a division of area, it's a division of labor. Cells do some of the drawing that controls would otherwise have to do, and cells hold some of the data that controls would otherwise have to carry. Two items of this data are the instance variables for target and action. [Figure 5-4](#) (page 187) depicts the control-cell mechanism.

Being abstract classes, `NSControl` and `NSCell` both incompletely handle the setting of the target and action instance variables. By default, `NSControl` simply sets the information in its associated cell, if one exists. (`NSControl` itself supports only a one-to-one mapping between itself and a cell; subclasses of `NSControl` such as `NSMatrix` support multiple cells.) In its default implementation, `NSCell` simply raises an exception. You must go one step further down the inheritance chain to find the class that really implements the setting of target and action: `NSActionCell`.

Objects derived from `NSActionCell` provide target and action values to their controls so the controls can compose and send an action message to the proper receiver. An `NSActionCell` object handles mouse (cursor) tracking by highlighting its area and assisting its control in sending action messages to the specified target. In most cases, the responsibility for an `NSControl` object's appearance and behavior is completely given over to a corresponding `NSActionCell` object. (`NSMatrix`, and its subclass `NSForm`, are subclasses of `NSControl` that don't follow this rule.)

Figure 5-4 How the target–action mechanism works



When users choose an item from a menu, an action is sent to a target. Yet menus (`NSMenu` objects) and their items (`NSMenuItem` objects) are completely separate, in an architectural sense, from controls and cells. The `NSMenuItem` class implements the target–action mechanism for its own instances; an `NSMenuItem` object has both target and action instance variables (and related accessor methods) and sends the action message to the target when a user chooses it.

Note: “Controls and Menus” (page 230) discusses control-cell and menu architecture in greater detail. Also see *Control and Cell Programming Topics for Cocoa* and *Application Menu and Pop-up List Programming Topics for Cocoa*.

Setting the Target and Action

You can set the targets and actions of cells and controls programmatically or by using Interface Builder. For most developers and most situations, Interface Builder is the preferred approach. When you use it to set controls and targets, Interface Builder provides visual confirmation, allows you to lock the connections, and archives the connections to a nib file. The procedure, which is covered in more detail in the development-tools documentation, is simple:

1. When defining your custom class in Xcode, declare an action method using the `IBAction` qualifier.
2. In Interface Builder, drag a generic object into the top level of the nib file window (if one doesn't already exist for your custom class).

If an instance of your custom class is to be the File's Owner for the nib file, this step is not necessary. Also, if you're defining a custom `NSView` object, select that object instead.

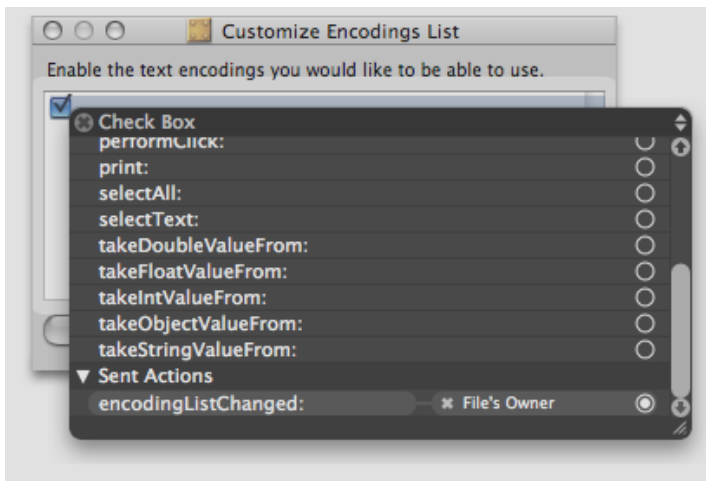
3. Import the custom class into Interface Builder.

With the generic object (or File's Owner) selected, type the name of your custom class in the Class field of the Identify pane of the Interface Builder inspector. This assigns your custom class as the class of the selected object. This step is performed once.

4. Select the control or cell that is to send the action message to your custom object.
5. Right-click or Control-click this object to display the connections panel.
6. Find your action method under Sent Actions. Then drag a connection line *from* the circle next to the action method *to* the icon representing the instance of your custom class.

Figure 5-5 illustrates what an action connection looks like in the connections panel when you've completed these steps.

Figure 5-5 Setting target and action in Interface Builder



If the action is handled by a superclass of your custom class or by an off-the-shelf Application Kit or UIKit class, you can make the connection without declaring any action method. Of course, if you declare an action method yourself, you must be sure to implement it.

To set the action and the target programmatically, use the following methods to send messages to a control or cell object:

```
- (void)setTarget:(id)anObject;  
- (void)setAction:(SEL)aSelector;
```

The following example shows how you might use these methods:


```
[aCell setTarget:myController];
[aControl setAction:@selector(deleteRecord)];
[aMenuItem setAction:@selector(showGuides)];
```

Programmatically setting the target and action does have its advantages and indeed in certain situations it is the only possible approach. For example, you might want the target or action to vary according to some runtime condition, such as whether a network connection exists or whether an inspector window has been loaded. Another example is when you are dynamically populating the items of a pop-up menu, and you want each pop-up item to have its own action.

Actions Defined by the Application Kit

The Application Kit not only includes many `NSActionCell`-based controls for sending action messages, it defines action methods in many of its classes. Some of these actions are connected to default targets when you create a Cocoa application project. For example, the Quit command in the application menu is connected to the `terminate:` method in the global application object (`NSApp`).

The `NSResponder` class also defines many default action messages (also known as standard commands) for common operations on text. This allows the Cocoa text system to send these action messages up an application's responder chain—a hierarchical sequence of event-handling objects—where it can be handled by the first `NSView`, `NSWindow`, or `NSApplication` object that implements the corresponding method.

Target-Action in UIKit

The UIKit framework also declares and implements a suite of control classes; the control classes in this framework inherit from the `UIControl` class, which defines most of the target-action mechanism for iPhone OS. However there are some fundamental differences in how the two frameworks implement target-action. One of these differences is that UIKit does not have any true cell classes. Controls in UIKit do not rely upon their cells for target and action information.

A larger difference in how the two frameworks implement target-action lies in the nature of the event model. In the Application Kit, the user typically uses a mouse and keyboard to register events for handling by the system. These events—such as the click of a mouse pointer on a button—are limited and discrete. Consequently, a control object in the Application Kit usually recognizes a single physical event as the trigger for the action it sends to its target. (In the case of buttons, this is a mouse-up event.) On iPhone OS, the user's fingers are what originate events instead of mouse clicks, mouse drags, or physical keystrokes. There can be more than one of these fingers touching an object on the screen at one time, and these “touches” can even be going in different directions.

To account for this Multi-Touch event model, UIKit declares a set of “control event” constants in `UIControl.h` that specify various physical gestures that users can make on controls, such as lifting a finger from a control, dragging a finger into a control, and touching down within a text field. You can configure a control object so that it responds to one or more of these touch events by sending an action message to a target. Many of the control classes in UIKit are implemented to generate certain control events; for example, instances of the `UISlider` class generate a `UIControlEventValueChanged` control event, which you can use to send an action message to a target object.

You set up a control so that it sends an action message to a target object by associating both target and action with one or more control events. To do this, send `addTarget:action:forControlEvents:` to the control for each target-action pair you wish to specify. When the user touches the control in a designated fashion, the control forwards the action message to the global `UIApplication` object in a

`sendAction:to:from:forEvent: message`. As in the Application Kit, the global application object is the centralized dispatch point for action messages. If the control specifies a `nil` target for an action message, the application queries objects in the responder chain until it finds one that is willing to handle the action message—that is, one implementing a method corresponding to the action selector.

In contrast with the Application Kit, where an action method may have only one or perhaps two valid signatures, the UIKit framework allows three different forms of action selector:

- (void)action
- (void)action:(id)sender
- (void)action:(id)sender forEvent:(UIEvent *)event

To learn more about the target-action mechanism in UIKit, read *UIControl Class Reference*.

Bindings (Mac OS X)

Bindings are a Cocoa technology that you can use to synchronize the display and storage of data in a Cocoa application created for Mac OS X. They are an important tool in the Cocoa toolbox for enabling communication between objects. The technology is an adaptation of both the Model-View-Controller and object modeling design patterns. (“[The Model-View-Controller Design Pattern](#)” (page 161) introduced bindings in its discussion of controller objects.) It allows you to establish a mediated connection—a binding—between the attribute of a view object that displays a value and a model-object property that stores that value; when a change occurs in the value in one side of the connection, it is automatically reflected in the other. The controller object that mediates the connection provides additional support, including selection management, placeholder values, and sortable tables.

How Bindings Work

Bindings inhabit the conceptual space defined by the Model-View-Controller (MVC) and object modeling design patterns. An MVC application assigns objects general roles and maintains separation between objects based on these roles. Objects can be view objects, model objects, or controller objects whose roles can be briefly stated as follows:

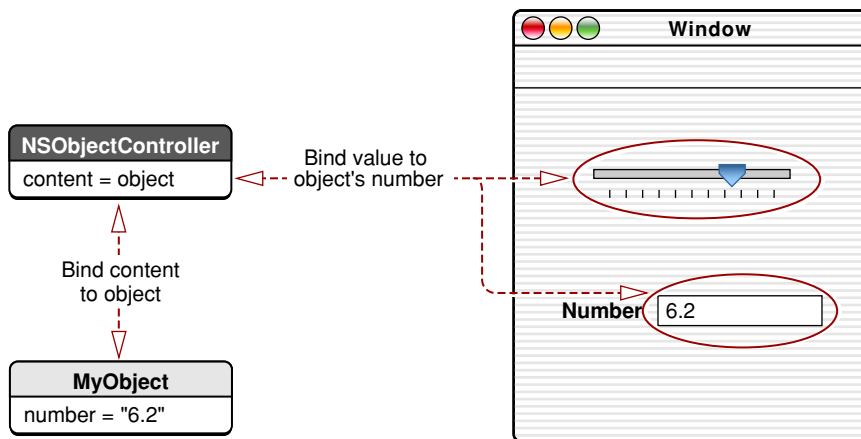
- View objects display the data of the application.
- Model objects encapsulate and operate on application data. They are typically the persistent objects that your users create and save while an application is running.
- Controller objects mediate the exchange of data between view and model objects and also perform “command and control” services for the application.

All objects, but most importantly model objects, have defining components or characteristics called properties. Properties can be of two sorts: attributes—values such as strings, scalars, and data structures—and relationships to other objects. Relationships can be of two sorts: one-to-one and one-to-many. They can also be bidirectional and reflexive. The objects of an application thus have various relationships with each other, and this web of objects is called an object graph. A property has an identifying name called a key. Using key paths—period-separated sequences of keys—one can traverse the relationships in an object graph to access the attributes of related objects.

The bindings technology makes use of this object model to establish bindings among the view, model, and controller objects of an application. With bindings you can extend the web of relationships from the object graph of model objects to the controller and view objects of an application. You can establish a binding between an attribute of a view object and a property of a model object (typically through a mediating property of a controller object). Any change in the displayed attribute value is automatically propagated through the binding to the property where the value is stored. And any internal change in the value of the property is communicated back to the view for display.

For example, [Figure 5-6](#) (page 191) shows a simplified set of bindings between the displayed values of a slider and a text field (attributes of those view objects) and the `number` attribute of a model object (`MyObject`) through the `content` property of a controller object. With these bindings established, if a user moves the slider, the change in value would be applied to the `number` attribute and communicated back to the text field for display.

Figure 5-6 Bindings between view, controller, and model objects



The implementation of bindings rests on the enabling mechanisms of key-value coding, key-value observing, and key-value binding. See ["Key-Value Mechanisms"](#) (page 131) for overviews of these mechanisms and their associated informal protocols. The discussion of the Observer pattern in ["Cocoa Design Patterns"](#) (page 141) also describes key-value observing.

You can establish a binding between any two objects. The only requirement is that the objects comply with the conventions of key-value coding and key-value observing. However, you generally want to establish the binding *through* a mediating controller because such controller objects offer bindings-related services such as selection management, placeholder values, and the ability to commit or discard pending changes. Mediating controllers are instances of several `NSController` subclasses; they are available on the Interface Builder Controllers palette (see ["How You Establish Bindings,"](#) below). You can also create custom mediating-controller classes to acquire more specialized behavior. For a discussion of mediating controllers and `NSController` objects, see ["Types of Cocoa Controller Objects"](#) (page 164) in ["The Model-View-Controller Design Pattern"](#) and the section on the Mediator pattern in ["Cocoa Design Patterns"](#) (page 141).

Further Reading: To learn more about the design patterns summarized above, see [“The Model-View-Controller Design Pattern”](#) (page 161) and [“Object Modeling”](#) (page 169).

How You Establish Bindings

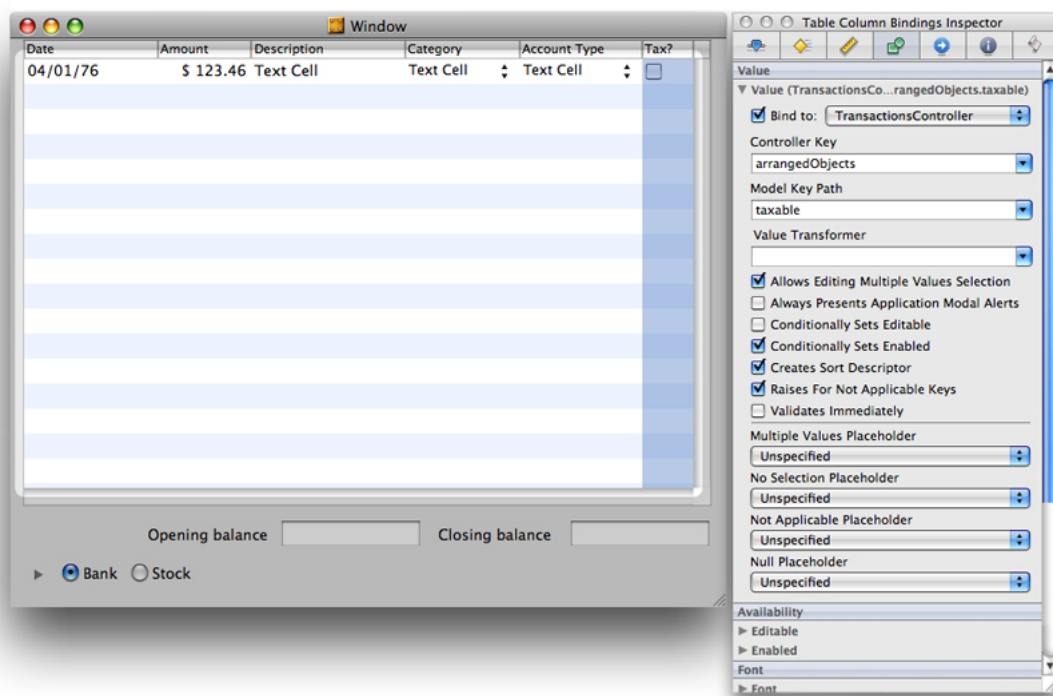
If the only custom classes of your application are model classes, the only requirement for establishing bindings is that those classes be compliant with key-value coding conventions for any properties you want to bind. If you are using a custom view or custom controller, you should also ensure that it is key-value observing compliant. See [“Key-Value Mechanisms”](#) (page 131) for a summary of the requirements for compliance with both key-value coding and key-value observing.

Note: Most of the classes of the Cocoa frameworks are compliant with key-value coding and (where appropriate) key-value observing.

You can also establish bindings programmatically but for most situations you use the Interface Builder application to establish bindings. In Interface Builder, you start by dragging `NSController` objects from a palette into your nib file. Then you use the Bindings pane of the Info window to specify the relationships between the properties of the view, controller, and model objects of your application and the attributes you want bound.

Figure 5-7 gives an example of a binding. It shows the “Tax?” column of the top table view bound to the model attribute `taxable` through the property `arrangedObjects` of the `Entrees` controller (an `NSArrayController` object), which itself is bound to an array of model objects (not shown).

Figure 5-7 Establishing a binding in Interface Builder



Further Reading: Read *Cocoa Bindings Programming Topics* to learn more about the bindings technology and how to use Interface Builder to establish bindings. Also see *Key-Value Coding Programming Guide* and *Key-Value Observing Programming Guide* for complete descriptions of these mechanisms.

Notifications

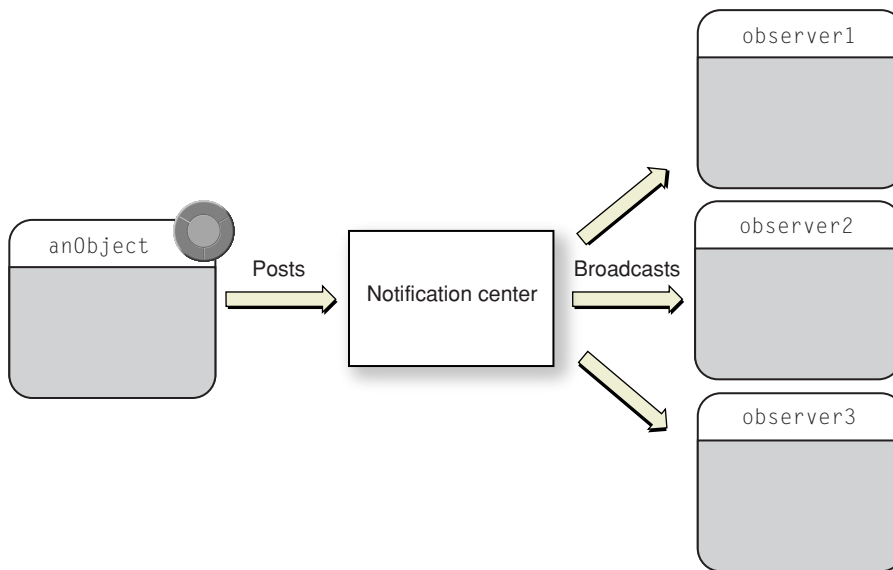
The standard way to pass information between objects is message passing—one object invokes the method of another object. However, message passing requires that the object sending the message know who the receiver is and what messages it responds to. This requirement is true of delegation messages as well as other types of messages. At times, this tight coupling of two objects is undesirable—most notably because it would join together what might be two otherwise independent subsystems. And it is impractical, because it would require hard-coded connections between many disparate objects in an application.

For cases where standard message-passing just won't do, Cocoa offers the broadcast model of notification. By using the notification mechanism, one object can keep other objects informed of what it is doing. In this sense, it is similar to delegation, but the differences are important. The key distinction between delegation and notification is that the former is a one-to-one communication path (between the delegating object and its delegate). But notification is a potentially one-to-many form of communication—it is a broadcast. An object can have only one delegate, but it can have many *observers*, as the recipients of notification are known. And the object doesn't have to know what those observers are. Any object can observe an event indirectly via notification and adjust its own appearance, behavior, and state in response to the event. Notification is a powerful mechanism for attaining coordination and cohesion in a program.

How the notification mechanism works is conceptually straightforward. A process has an object called a notification center, which kind of acts as a clearing house and broadcast center for notifications. Objects that need to know about an event elsewhere in the application register with the notification center to let it know they want to be notified when that event happens. An example of this is a controller object that needs to know when a pop-up selection is made so it can reflect this change in the user interface. When the event does happen, the object handling the event posts a notification to the notification center, which then dispatches it to all of its observers. Figure 5-8 depicts this mechanism.

Note: The notification center delivers a notification to its observers synchronously. The posting object does not get back control until all notifications are sent. To post notifications asynchronously, you must use a notification queue (see “[Notification Queues](#)” (page 197)). A notification queue posts notifications to the notification center after it delays specified notifications and coalesces notifications that are similar according to some specified criteria.

Figure 5-8 Posting and broadcasting a notification



Any object can post a notification and any object can register itself with the notification center as an observer of a notification. The object posting the notification, the object that the posting object includes in the notification, and the observer of the notification may all be different objects or the same object. (Having the posting and observing object be the same does have its uses, such as in idle-time processing.) Objects that post notifications need not know anything about the observers. On the other hand, observers need to know at least the notification name and the keys to any dictionary encapsulated by the notification object. (“[The Notification Object](#)” (page 196) describes what a notification object consists of.)

Further Reading: For a thorough discussion of the notification mechanism, see *Notification Programming Topics for Cocoa*.

When and How to Use Notifications

As with delegation, the notification mechanism is a great tool for enabling communication between objects in an application. Notifications allow objects within an application to learn about changes that occur elsewhere in that application. Generally, an object registers to be an observer of a notification because it wants to make adjustments when a certain event occurs or is about to occur. For example, if a custom view wants to change its appearance when its window is resized, it can observe the `NSNotificationDidResizeNotification` posted by that window object. Notifications also permit information to be passed between objects because a notification can include a dictionary of data related to the event.

But there are differences between notification and delegation, and these differences dictate what these mechanisms should be used for. As noted earlier, the main difference between the notification model and the delegation model is that the former is a broadcast mechanism whereas delegation is a one-to-one relationship. Each model has its advantages; with notifications they include the following:

- The posting object does not have to know the identity of the observing objects.
- An application is not limited to the notifications declared by the Cocoa frameworks; any class can declare notifications for its instances to post.
- Notifications are not limited to intra-application communication; with distributed notifications, one process can notify another process about events that occur.

iPhone OS Note: Distributed notifications are not available on iPhone OS.

But the one-to-one model of delegation has its advantages too. A delegate is given the opportunity to affect an event by returning a value to the delegating object. A notification observer, on the other hand, must play a more passive role; it can affect only itself and its environment in response to the event. Notification methods must have the following signature:

```
- (void)notificationHandlerName:(NSNotification *);
```

This requirement precludes the observing object from affecting the original event in any direct way. A delegate, however, can often affect how the delegating object will handle an event. Moreover, the delegate of an Application Kit object is automatically registered as an observer of its notifications. All it need do is implement the notification methods defined by the framework class for its notifications.

The notification mechanism is not the only Cocoa alternative for observing changes in object state, and indeed for many situations should not be the preferred one. The Cocoa bindings technology, and specifically its enabling key-value observing (KVO) and key-value binding (KVB) protocols, also allow objects in an application to observe changes in the properties of other objects. The bindings mechanism accomplishes this function more efficiently than do notifications. In bindings, the communication between observed and observing object is direct, and does not require an intermediary object such as the notification center. Moreover, the bindings mechanism imposes no performance penalty for unobserved changes, as do regular notifications.

However, there can be situations where it makes sense to prefer notifications over bindings. You may want to observe events other than a change in object properties. Or it might be impractical to implement KVO and KVB compliance, especially when the notifications to be posted and observed are few.

Even if the situation warrants the use of notifications, you should be aware of the performance implications. When you post a notification, it is eventually dispatched to observing objects synchronously by the local notification center. This occurs regardless of whether the posting was done synchronously or asynchronously. If there are many observers or each observer does a lot of work while handling the notification, your program could experience a significant delay. Therefore you should be careful about overusing notifications or using them inefficiently. The following guidelines for notification usage should help toward this end:

- Be selective about which notifications your application should observe.
- Be specific about notification names and posting objects when you register for notifications.
- Implement the methods that handle notifications to be as efficient as possible.

- Refrain from adding and removing numerous observers; it is much better to have a few "intermediary" observers that can communicate the results of notifications to the objects they have access to.

Further Reading: For detailed information about the efficient use of notifications, see "Notifications" in *Cocoa Performance Guidelines*.

The Notification Object

A notification is an object, an instance of `NSNotification`. This object encapsulates information about an event, such as a window gaining focus or a network connection closing. When the event does happen, the object handling the event posts the notification to the notification center, which immediately broadcasts the notification to all registered objects.

An `NSNotification` object contains a name, an object, and an optional dictionary. The name is a tag identifying the notification. The object is any object that the poster of the notification wants to send to observers of that notification (typically it is the object that posted the notification). It is similar to the sender object in delegation messages, allowing the receiver to query the object for more information. The dictionary stores any information related to the event.

Notification Centers

A notification center manages the sending and receiving of notifications. It notifies all observers of notifications meeting specific criteria. The notification information is encapsulated in `NSNotification` objects. Client objects register themselves with the notification center as observers of specific notifications posted by other objects. When an event occurs, an object posts an appropriate notification to the notification center. The notification center dispatches a message to each registered observer, passing the notification as the sole argument. It is possible for the posting object and the observing object to be the same.

Cocoa includes two types of notification centers:

- A notification center (an instance of `NSNotificationCenter`) manages notifications within a single task.
- A distributed notification center (an instance of `NSDistributedNotificationCenter`) manages notifications across multiple tasks on a single computer.

Note that in contrast to many other Foundation classes, `NSNotificationCenter` is not toll-free bridged to its Core Foundation counterpart (`CFNotificationCenterRef`).

NSNotificationCenter

Each task has a default notification center that you access with the `NSNotificationCenter` class method `defaultCenter`. The notification center handles notifications within a single task. For communication between tasks on the same machine, use a distributed notification center (see "[NSDistributedNotificationCenter](#)" (page 197)).

A notification center delivers notifications to observers synchronously. In other words, when posting a notification, control does not return to the poster until all observers have received and processed the notification. To send notifications asynchronously use a notification queue, which is described in [“Notification Queues”](#) (page 197).

In a multithreaded application, notifications are always delivered in the thread in which the notification was posted, which may not be the same thread in which an observer registered itself.

NSDistributedNotificationCenter

Each task has a default distributed notification center that you access with the `NSDistributedNotificationCenter` class method `defaultCenter`. This distributed notification center handles notifications that can be sent between tasks on a single machine. For communication between tasks on different machines, use distributed objects (see *Distributed Objects Programming Topics*).

Posting a distributed notification is an expensive operation. The notification gets sent to a system-wide server that then distributes it to all the tasks that have objects registered for distributed notifications. The latency between posting the notification and the notification's arrival in another task is unbounded. In fact, if too many notifications are being posted and the server's queue fills up, notifications can be dropped.

Distributed notifications are delivered via a task's run loop. A task must be running a run loop in one of the “common” modes, such as `NSDefaultRunLoopMode`, to receive a distributed notification. If the receiving task is multithreaded, do not depend on the notification arriving on the main thread. The notification is usually delivered to the main thread's run loop, but other threads could also receive the notification.

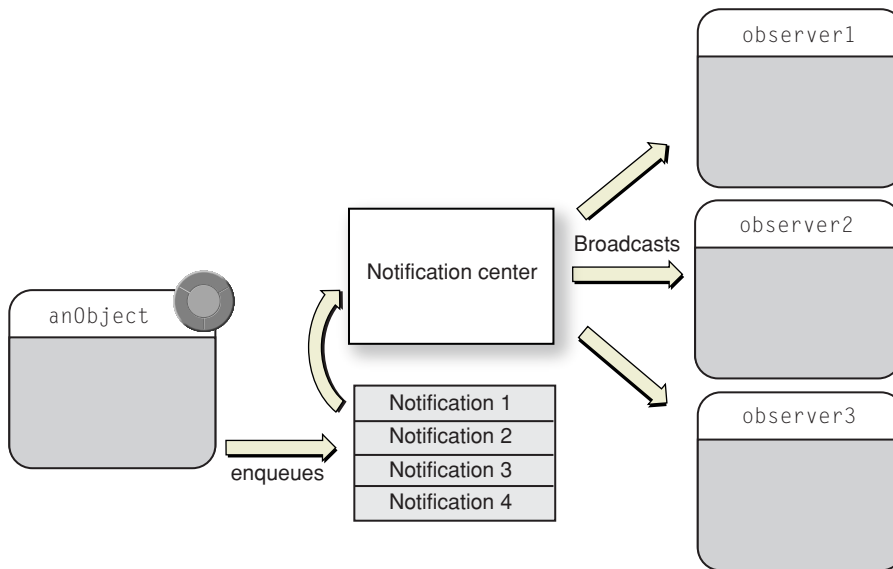
Whereas a regular notification center allows any object to be the notification object (that is, the object encapsulated by the notification), a distributed notification center is restricted to having an `NSString` object as its notification object. Because the posting object and the observer may be in different tasks, notifications cannot contain pointers to arbitrary objects. Therefore, a distributed notification center requires notifications to use a string as the notification object. Notification matching is done based on this string, rather than an object pointer.

Notification Queues

`NSNotificationQueue` objects (or simply, notification queues) act as buffers for notification centers (instances of `NSNotificationCenter`). A notification queue maintains notifications (instances of `NSNotification`) generally in a First In First Out (FIFO) order. When a notification rises to the front of the queue, the queue posts it to the notification center, which in turn dispatches the notification to all objects registered as observers.

Every thread has a default notification queue, which is associated with the default notification center for the task. Figure 5-9 illustrates this association. You can create your own notification queues and have multiple queues per center and thread.

Figure 5-9 A notification queue and notification center



Coalescing Notifications

The `NSNotificationQueue` class contributes two important features to the Foundation Kit's notification mechanism: the coalescing of notifications and asynchronous posting. Coalescing is a process that removes notifications in the queue that are similar to the notification just queued. If the new item is similar to a notification already queued, the new one isn't queued and all similar notifications (except the first one in the queue) are removed. However, you should not depend on this particular coalescing behavior.

You indicate the criteria for similarity by specifying one or more of the following constants in the third argument of the `enqueueNotification:postingStyle:coalesceMask:forModes:` method.

```
NSNotificationNoCoalescing
NSNotificationCoalescingOnName
NSNotificationCoalescingOnSender
```

You can perform a bitwise-OR operation with the `NSNotificationCoalescingOnName` and `NSNotificationCoalescingOnSender` constants to specify coalescing using both the notification name and notification object. In this case, all notifications having the same name and sender as the one enqueued are coalesced.

Asynchronously Posting Notifications

With the `NSNotificationCenter` method `postNotification:` and its variants, you can post a notification immediately to a notification center. However, the invocation of the method is synchronous: Before the posting object can resume its thread of execution, it must wait until the notification center dispatches the notification to all observers and returns. With the `NSNotificationQueue` methods `enqueueNotification:postingStyle:` and `enqueueNotification:postingStyle:coalesceMask:forModes:` methods, however, you can post a notification asynchronously by putting it in a queue. These methods immediately return to the invoking object after putting the notification in the queue.

The notification queue is emptied and its notifications posted based on the posting style and run loop mode specified in the enqueueing method. The mode argument specifies the run loop mode in which the queue will be emptied. For example, if you specify `NSModalPanelRunLoopMode`, the notifications will be posted only when the run loop is in this mode. If the run loop is not currently in this mode, the notifications wait until the next time that mode is entered.

Posting to a notification queue can occur in one of three different styles: `NSPostASAP`, `NSPostWhenIdle`, and `NSPostNow`. These styles are described in the following sections.

Posting As Soon As Possible

Any notification queued with the `NSPostASAP` style is posted to the notification center when the current iteration of the run loop completes, assuming the current run loop mode matches the requested mode. (If the requested and current modes are different, the notification is posted when the requested mode is entered.) Because the run loop can make multiple callouts during each iteration, the notification may or may not get delivered as soon as the current callout exits and control returns to the run loop. Other callouts may take place first, such as a timer or source firing or other asynchronous notifications delivered.

You typically use the `NSPostASAP` posting style for an expensive resource, such as the display server. When many clients draw on the window buffer during a callout from the run loop, it is expensive to flush the buffer to the display server after every draw operation. In this situation, each `draw...` method enqueues some notification such as “FlushTheServer” with coalescing on name and object specified and with a posting style of `NSPostASAP`. As a result, only one of those notifications is dispatched at the end of the run loop and the window buffer is flushed only once.

Posting When Idle

A notification queued with the `NSPostWhenIdle` style is posted only when the run loop is in a wait state. In this state, there’s nothing in the run loop’s input channels, be it timers or other asynchronous events. Note that a run loop that is about to exit (which occurs when all of the input channels have expired) is not in a wait state and thus will not post a notification.

Posting Immediately

A notification queued with `NSPostNow` is posted immediately after coalescing to the notification center. You queue a notification with `NSPostNow` (or post one with the `NSNotificationCenter` method `postNotification:`) when you do not require asynchronous calling behavior. For many programming situations, synchronous behavior is not only allowable but desirable: You want the notification center to return after dispatching so you can be sure that observing objects have received and processed the notification. Of course, you should use `enqueueNotification...` with `NSPostNow` rather than use `postNotification:` when there are similar notifications in the queue that you want to remove through coalescing.

Ownership of Delegates, Observers, and Targets

Delegating objects are not considered to own their delegates or data sources. Similarly, controls and cells are not considered to own their targets, and the notification center does not own the observers of notifications. Consequently, for memory-managed code these framework objects follow the convention of *not* retaining their targets, observers, delegates, and data sources; instead, they simply store a pointer to the object.

Note: In memory management, a non-retained object reference is known as a weak reference, which is something altogether different from a weak reference in a garbage-collected environment. In the latter, all references to objects are considered strong by default and are thus visible to the garbage collector; weak references, which must be marked with the `__weak` type modifier, are not visible. In garbage collection, retain cycles are not a problem.

The object-ownership policy in memory management recommends that owned objects should be retained and archived unconditionally, and that referenced (but not owned) objects should not be retained and should be archived conditionally. The practical intent of this ownership policy is to avoid circular references, a situation where two objects retain each other. (This is often called a “retain cycle.”) Retaining an object creates a strong reference, and an object cannot be deallocated until all of its strong references are released. If two objects retain each other, neither object ever gets deallocated because the connection between them cannot be broken.

If you create a subclass from a Cocoa framework class with a delegate, data source, observer, or target, you should never explicitly retain the object in your subclass. You should create a non-retained reference to it and archive it conditionally.

Further Reading: For more on the ownership policy, weak references, and circular references in memory management, see “Object Ownership and Disposal” in *Memory Management Programming Guide for Cocoa*. For a summary of strong references and weak references in garbage collection, see “Garbage Collection for Cocoa Essentials” in *Garbage Collection Programming Guide*.

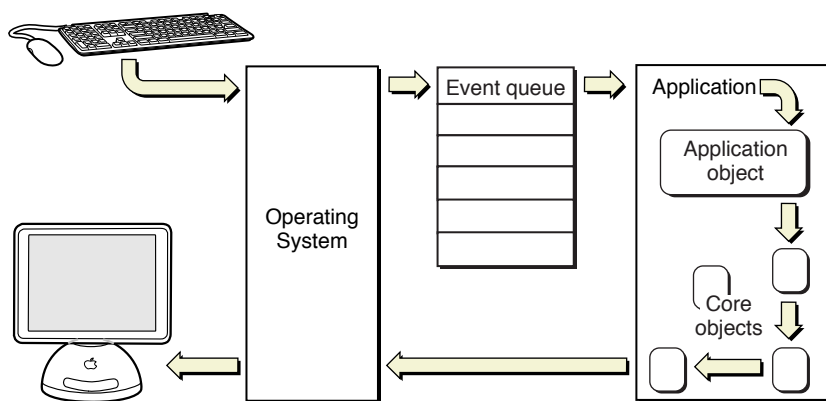
The Core Application Architecture on Mac OS X

When a user launches a Cocoa application, a network of objects is put into place. This runtime network consists of Application Kit objects of different types, each performing a specific role. These objects are related to each other in various ways defined by ownership, dependency, and cooperation. This chapter looks at this application architecture and examines the roles played by the core objects, their primary attributes, and the relationships they have with each other.

The Event-Drawing Cycle, Revisited

The overall purpose of the network of objects is to expedite the work of the event and drawing cycle. Figure A-1 (which you may recall from [“Adding Behavior to a Cocoa Program”](#) (page 109)) illustrates this cycle.

Figure A-1 The event and drawing cycle



The application presents a graphical user interface; users interact with that interface by using the mouse and keyboard to enter data or indicate a choice; and this input is transformed into an event that is routed to the application and placed in its event queue. For each event in its queue, the application locates the object or objects best suited to handle the event, and after the event is handled the application may appropriately modify what it displays to users. Then the application gets the next event in its queue and the cycle begins again.

The core objects that participate in this architecture are direct or indirect descendants of the `NSApplication`, `NSWindow`, and `NSView` classes.

The Global Application Object

Every Cocoa application is managed by a single `NSApplication` object known by the name of the global variable, `NSApp`. This singleton instance of `NSApplication` (or of a custom subclass thereof) has a major responsibility: obtaining user and system events targeted at the application and dispatching them to the appropriate objects. It is also responsible for managing the application's windows, including tracking their current status as key or main window (see [“Window Status”](#) (page 214)).

The Main Event Loop

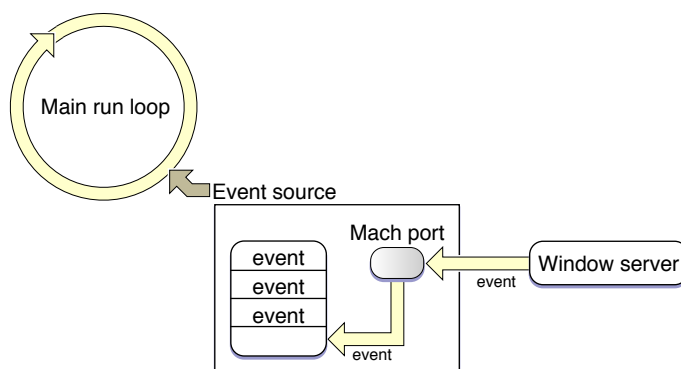
Most of the work that `NSApp` does takes place in the application's main event loop, which is the basis of the event and drawing cycle. To understand how the main event loop operates, it helps to consider what takes place in a Cocoa application's main entry point. In a standard Xcode Cocoa project, `main` has a single function call, `NSApplicationMain`. This function does three important things, in the following order:

1. It obtains the shared application object (`NSApp`) by invoking the class method `sharedApplication`.
2. It loads the application's main nib file into memory.
3. It runs the application (`[NSApp run]`).

Let's examine these steps in a little more detail, starting with some background information. A running application is essentially a process, and each process has a main thread and possibly one or more secondary threads. Each thread always has a run loop created for it; a run loop is a mechanism that monitors sources of input to the process and dispatches control to it when sources become ready for processing.

The main thing that the `sharedApplication` method does—outside of ensuring a singleton instance—is to set up the programmatic infrastructure for receiving and processing events from the window server. In initializing the global application object, `NSApplication` establishes a connection to the window server by creating an event source for receiving events (implemented as a Mach port). It also sets up the application's event queue, a FIFO mechanism that enqueues events fetched from the event source as they arrive. Finally, `NSApplication` initializes the main run loop—the run loop on the main thread—with the event source as an input source (see Figure A-2).

Figure A-2 Main event loop, with event source



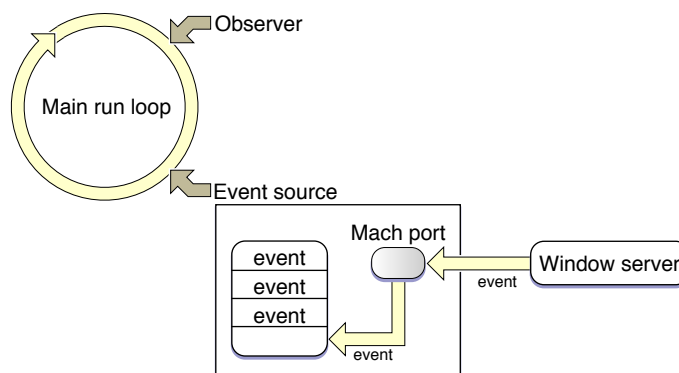
The window server receives events from I/O Kit device drivers and dispatches them to the appropriate processes. The process receives the event from the event source in the run loop and places it on a queue.

When the application loads the main nib file, the objects in the file are unarchived along with the connections between these objects. The main nib file always contains the application menu and may contain one or more window objects (with their constituent views). Other nib files may be unarchived at launch time as well; document-based applications, for instance, may unarchive the nib files of documents when a user double-clicks a file in the Finder. This initial graphical user interface is needed so users can begin making requests—by, for instance, choosing a menu item—and consequently generating events for the application to handle.

The `NSApplication` `run` method is the workhorse of the main event loop. It first registers Apple event handlers for the application (see [“Handling Apple Events”](#) (page 205)). Then in a closed `while` loop, `run` does the following until the application terminates:

1. It services pending window-display observers on the run loop (see Figure A-3, which results in the redrawing of regions in windows that are marked as "dirty."

Figure A-3 Main event loop, with run-loop observer



2. It gets the next event in the event queue (`nextEventMatchingMask:untilDate:inMode:dequeue:`).
3. It dispatches the event to the object that should next handle it, which in most cases is an `NSWindow` object (`sendEvent:`). See [“More About Event Dispatch”](#) (page 204) for clarification.

Ultimately, numerous objects may become involved in handling the event, and the call stack may grow significantly. After the event is handled, control returns to `run`.

The funnel point of the main event loop is the `nextEventMatchingMask:untilDate:inMode:dequeue:` method. If there are events in the event queue, this method fetches the topmost event in the queue and converts the structure to an `NSEvent` object. If there are no events in the queue, the method blocks. During the period the method blocks, new events from the window server are processed and placed in the queue. The presence of new events "wakes" `nextEventMatchingMask:untilDate:inMode:dequeue:` and it returns the first matching event in the queue.

If a window has its auto-display feature enabled and it has views marked for display, it also installs an observer (an input source) on the run loop. Just before `NSApp` processes the next event in the queue, this observer fires, leading to the redrawing of those views. For more on this automatic displaying of window content, see [“Windows and Drawing”](#) (page 213).

More About Event Dispatch

In its `sendEvent:` implementation, `NSApp` looks at the type of event passed in and dispatches it accordingly. Usually the target of the dispatch is one of the application's windows, and the application forwards the event by invoking the `sendEvent:` method of that `NSWindow` object. For the two major types of input events, key events and mouse events, `NSApp` uses a different approach to find the appropriate window.

When users press keys on the keyboard, key events are generated. `NSApp` forwards these events on to the key window—the window of the application that is currently accepting key presses. (In some cases, such as key equivalents, the application handles `NSKeyDown` events specially instead of dispatching them.)

When users click an object on a window with a mouse, such as a graphical shape in a drawing program, mouse events are generated. `NSApp` dispatches mouse events to the window in which the mouse event occurred.

If the object that is clicked or otherwise manipulated is a control object, such as a button or a slider, that object sends an additional sort of message to the application—an action message. That message invokes the `sendAction:to:from:` method of `NSApplication`. If a target of the message is not specified, this method searches the responder chains of the application's key and (if necessary) main windows, to find a suitable target. Then it sends the message to the target of the action message.

`NSApp` deals with other types of events in `sendEvent:`. Some events are specific to the application itself (such as activation and deactivation), and `NSApp` handles these itself. Other events are related to one or more windows—for example, a window is exposed or a change is made in screen resolution. For these, `NSApp` invokes the appropriate method in the affected windows.

Further Reading: The responder chain is described in [“Responders and the Responder Chain”](#) (page 228). To learn more about key windows, see [“Window Status”](#) (page 214). For a discussion of `NSControl` objects and how they work together with `NSCell` objects, read [“Controls and Menus”](#) (page 230).

Window Management

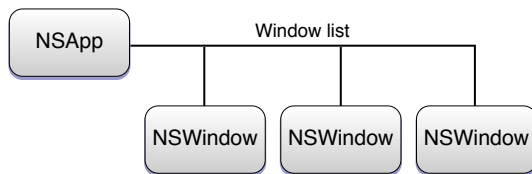
Part of the job of the global application object is to manage the windows of the application. `NSApp` performs the following window-management tasks:

- It keeps track of the current key and main windows.
- It hides and unhides windows.
- It deallocates windows at application termination.
- It manages window activation and deactivation during application switching.

And, as noted in [“More About Event Dispatch”](#) (page 204), `NSApp` dispatches window-specific event and action messages to the affected windows. It also maintains the application's Window menu and manages modal windows and panels. (See [“Modal Windows”](#) (page 215) for a description.)

To help it locate windows, the application object keeps a reference to them called its window list (see Figure A-4). For some window-management tasks (such as deallocation) it iterates through this list. (You can obtain the window list at any time by sending `windows` to `NSApp`.) The application object also finds windows by using the window number obtained from the current `NSEvent` object in a call to `windowWithWindowNumber:`.

Figure A-4 The application's window list



Note: `NSApp` doesn't have anything to do with the layering of windows on the screen (Z-order). That is managed by the window server. For a discussion of window Z-order, see [“Window Z-Order and Levels”](#) (page 208).

Handling Apple Events

Not all events that an application must deal with come from the event queue. Other processes on a Mac OS X system, such as the Finder and Launch Services, use Apple events to communicate with other processes. They occasionally send an application Apple events to notify it (for example) when users double-click a document in a Finder window or choose Shut Down from the Apple menu, requiring the application to terminate.

When a Cocoa application starts up, one of the first things it does is to register a number of Apple event handlers. When an Apple event is sent to the application, the appropriate handler is invoked. A Cocoa application registers handlers for the following Apple events:

Apple event ID	Description
<code>kAEOpenApplication</code>	Launches an application.
<code>kAReopenApplication</code>	Reopens an application. It is sent, for example, when an application is running and a user clicks the application icon in the Dock.
<code>kAEOpenDocuments</code>	Provides an application with a list of documents to open. This event is commonly sent when the user selects one or more documents in the Finder and double-clicks them.
<code>kAEPrintDocuments</code>	Provides an application with a list of documents to print. This event is commonly sent when a user selects one or more documents in the Finder and chooses Print from the File menu.

Apple event ID	Description
kAEOpenContents	Provides an application with dragged content, such as text or an image. This event is often sent when a user drags a file onto an application's icon in the Dock.
kAEQuitApplication	Requests the application to terminate.

Note: For more information on this subject, see "How Cocoa Applications Handle Apple Events".

Windows

An application uses windows to claim an area of the screen in which to display content and respond to user actions. Windows are essential to both drawing and event handling.

Further Reading: *Window Programming Guide for Cocoa* discusses many of the topics covered in this section in greater detail.

The Windows of an Application

Although it's possible to have an application with no visible windows (for example, a background application) such applications are rare. Generally, there are two types of applications based on the number of windows they can potentially display:

- Document-based—An application capable of creating multiple documents, each in its own window. Examples are word-processing and drawing applications. In a document-based application, users choose a menu option (typically File > New) to create a new document. Most document-based applications in Cocoa are based on its document architecture. (You can find an overview of this architecture in [“Other Cocoa Architectures on Mac OS X”](#) (page 241).)
- Single-window—An application that displays no more than one window at any time. Examples in Mac OS X are iSync and Font Book. When a single-window application launches, it displays its window. Often closing the window causes the application to terminate.

Any application can also have secondary windows, also known as dialogs and panels. These windows are subordinate to the current document window or, in the case of single-window applications, to the main window. They support the document or main window in various ways—for example, allowing selection of fonts and color, allowing the selection of tools from a palette, or displaying a warning. A secondary window is often modal. See [“Panels”](#) (page 215) for more information

NSWindow and the Window Server

In Cocoa an `NSWindow` object represents a physical window. The window server creates physical windows and ultimately manages them on the screen. It assigns each window a unique number as an identifier. The connection between an `NSWindow` object and its physical window is established through this window number.

When the window server creates a window, it obtains a window graphics context and initializes the graphics-state stack. It also creates the window's backing store, a memory region holding the pixel values of the window to be placed into the display device's frame buffer.

Note: It's possible to create a window without a backing store. This is called a deferred window. The backing store of a deferred window is created when the window is ordered on-screen.

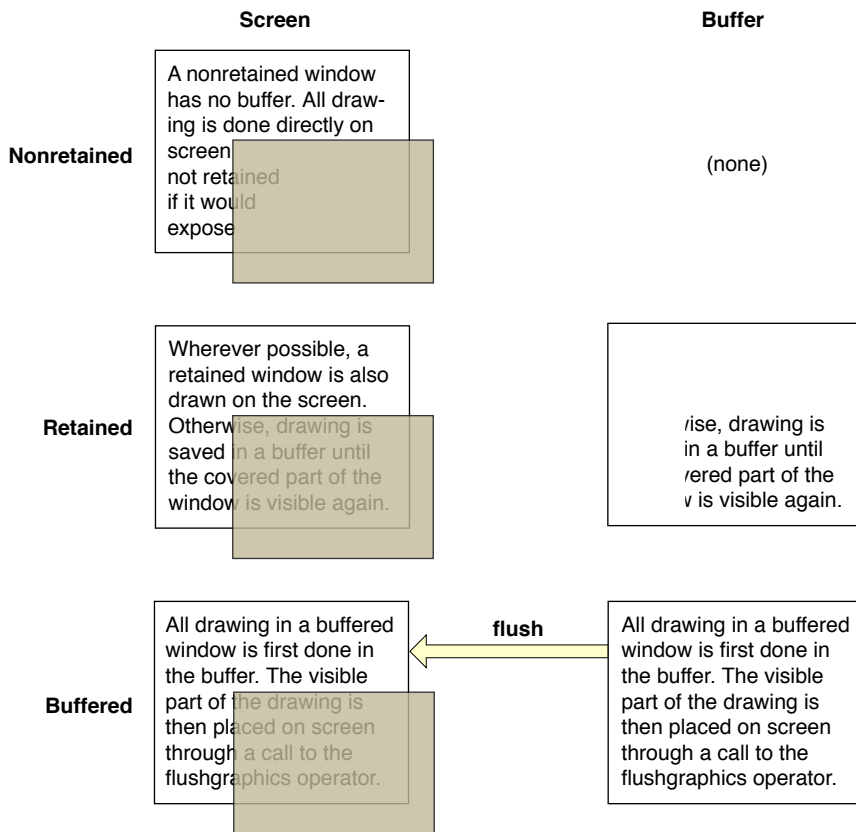
Window Buffering

Windows often overlap when there's more than one on the screen. When a group of windows are stacked one over another, a mouse click on a partially obscured window usually brings it to the front. In the process, its obscured contents become visible. If the window has a display buffer, the window server can automatically make those contents visible. Otherwise, the application has to redraw them.

There are two types of window buffering. A window's display buffer stores pixel values either for the entire window or for those portions of the window that are currently obscured. When an obscured window with a display buffer is made frontmost, the window server copies the exposed area from the display buffer to the screen. When you create an `NSWindow` object you can specify one of three buffering schemes, which are depicted in Figure A-5.

- **Buffered.** The buffer contains an exact duplicate of what's in backing store. In fact, drawing is done to the display buffer, and then its contents are flushed to the screen, composited with overlapping windows if there are shadows or transparency involved. If a window is obscured and then later brought frontmost, the entire display buffer is copied to the screen.
- **Retained.** Although a retained window has a display buffer, drawing is done directly on-screen when the window is unobscured and no alpha component is to be drawn. When any part of the window is obscured, the displayed contents are copied to the buffer, and drawing is done to the buffer. Modified parts of the buffer are copied to the display. When the obscured areas of the window on the display are later revealed, the contents of the display buffer are copied to the screen.
- **Nonretained.** No display buffer is provided. All drawing occurs directly onscreen—that is, the pixel values are written directly to the frame buffer. If part of the window is covered by another window, the bits for that part of the window's content area are lost. When the obscured window is exposed, the application must redraw the obscured portion; if it doesn't draw, that portion is displayed in the window's background color.

Figure A-5 Window buffering schemes



Only buffered windows support transparency effects, including the use of an alpha channel in drawing. Because the Quartz drawing system relies extensively on the use of alpha in drawing, windows to be used with Quartz should be buffered windows, and components that rely on Quartz, such as the Application Kit, should always use buffered windows. Nonretained windows might be used for transitory images or simple connection lines, such as are used in Interface Builder for target-action connections. However, the use of nonretained and retained windows is in general discouraged, because the presence of such windows interacts poorly with the modern user interface and may cause performance degradation.

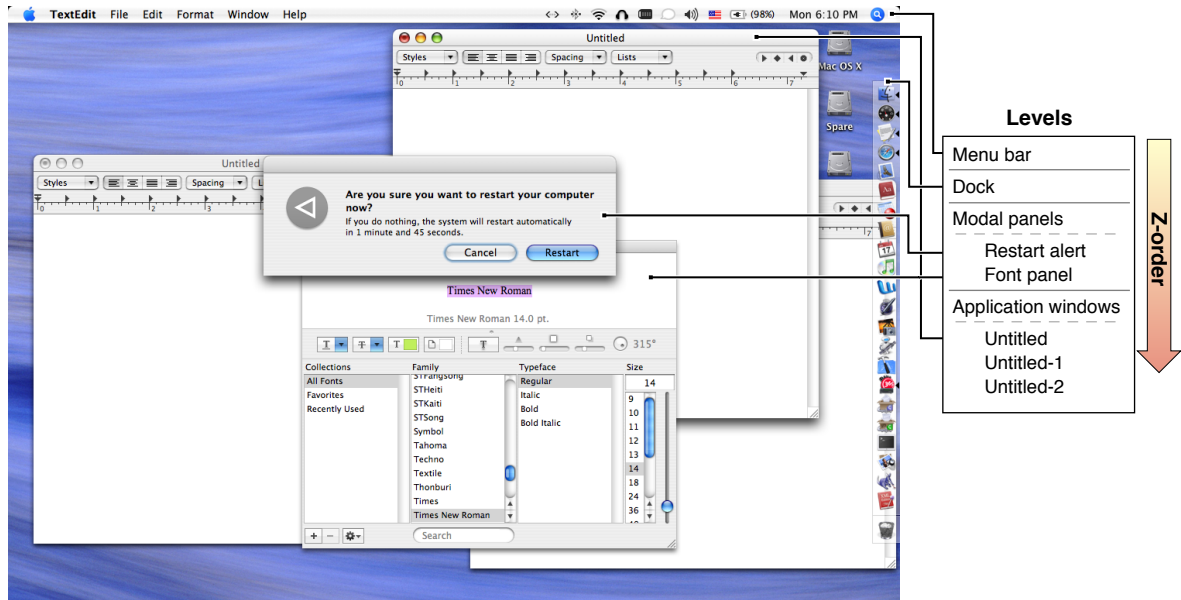
Window Z-Order and Levels

The window server maintains the windows it displays on a screen in a front-to-back order known as the screen list or the Z-order. Each window has a unique position in the list. A window can be at the top of the list—that is, the frontmost window. Or it can be at the bottom of the list—that is, the window potentially overlapped by every other window. Or it can be in any other position where it is above and below other windows. The Z-order is changed every time a user clicks a window, which makes that window frontmost and puts the previously frontmost window behind it.

The concept of window layering is complicated by window levels. A level is a layering subset of windows of a particular functional type. Window levels are in a hierarchical order, with those windows in higher levels displayed above windows in lower levels. This scheme guarantees that a window of

one particular type always appears above or below a window in another level. For example, a modal system dialog always appears above all application windows. Thus Z-order is really maintained within windows levels (see Figure A-6).

Figure A-6 Window levels and Z-order



The window server supports several levels of windows, in the following order:

1. Screen savers. (Screen savers use a window the size of the screen but without a title bar to display their content.) Screen-saver windows are displayed above all other windows.
2. The menu bar (which includes application menus).
3. The Dock.
4. Modal windows and panels (see [“Modal Windows”](#) (page 215)).
5. Contextual menus.
6. Floating windows (for example, palette windows for a drawing application).
7. Any other types of windows, including application windows.

A window can be explicitly taken off the screen list, in which case it's known as an offscreen window. A window disappears when it's removed from the list, and it's restored to the screen when it's put back on the list. Events are not dispatched to offscreen windows. Moving windows off the screen list is the basis for an application's window-hiding capability. An offscreen window must be buffered or retained in order for it to be drawn into.

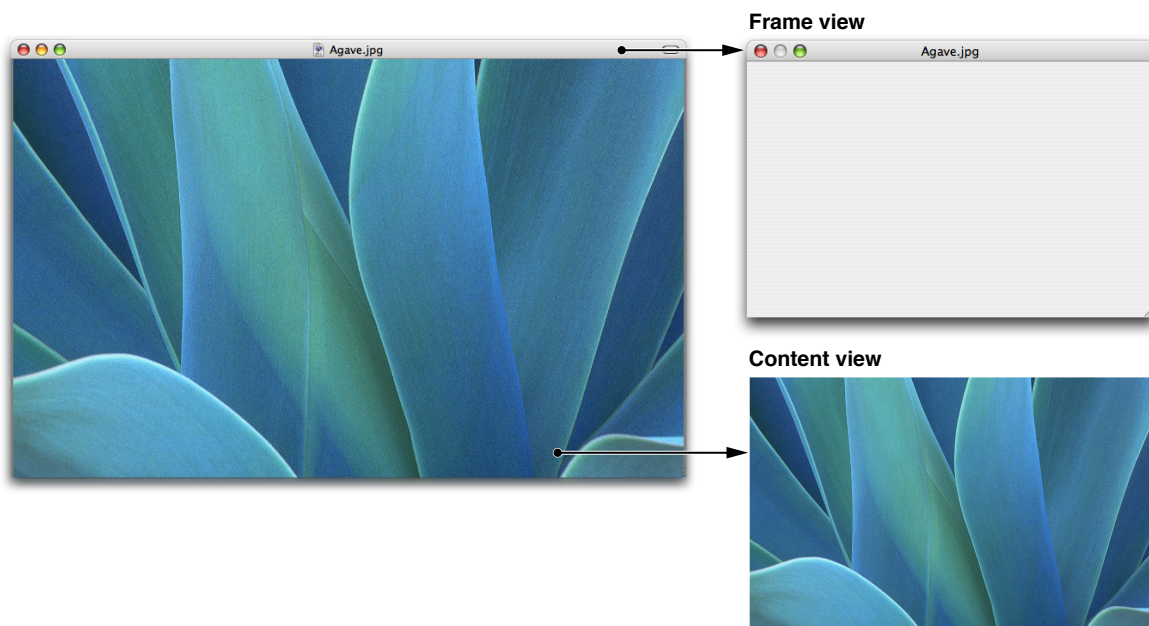
Note: You can also remove a borderless window from the screen by setting its frame coordinates outside of the visible coordinates of the screen. You cannot do this with other windows, however; the Application Kit constrains these windows to remain at least partially visible within the screen bounds.

`NSWindow` defines methods for manipulating the Z-order of application windows, for taking windows off and on the screen list, and for setting the level of windows. See the `NSWindow` class reference for details.

Parts of a Window

A window has two major parts: a frame area and a content area, as shown in Figure A-7. These areas are views—specifically, instances of a subclass of `NSView` (see “Views” (page 216)). The frame view surrounds the entire window area, and draws the window's border and title bar. The frame view is a private object created by `NSWindow`; it is not open to alteration through subclassing. However, when you create a window, you can specify which controls and other features—close button, miniaturize button, resizing triangle, title—you want the frame view to have.

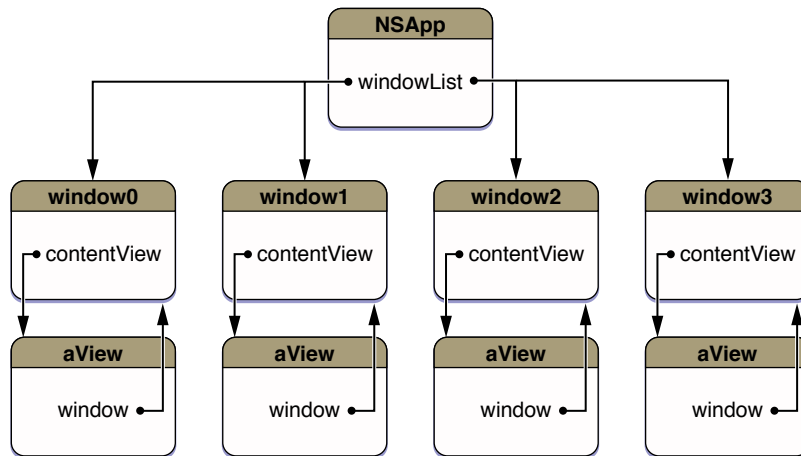
Figure A-7 A window's frame view and content view



The content view is inset into the frame view, usually occupying the entire area except for the title bar and the window border. Figure A-7 shows the content view relative to the frame view. (One can create windows that don't have a title bar or visible border, but they still have a frame view.) The content view is the sole public subview of the frame view. It is not a private view so you can replace it with your own custom view, if you wish. Although the content view has a superview (that is, a view that owns and encloses it in the hierarchy of views), that superview is a private object. The content view is thus the root of the hierarchy of views for the window. For more on the view hierarchy, see “Views” (page 216).

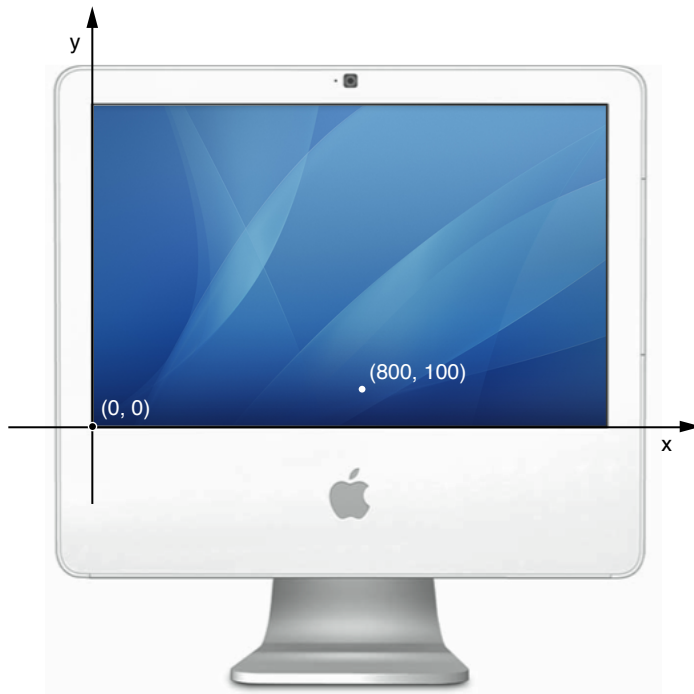
As do all view objects, the content view keeps a reference to its window, accessible through the `window` method. Figure A-8 depicts `NSApp` and its window list, the content view of each of the windows in the list, and the relationships between those objects.

Figure A-8 Relationships among `NSApp`, windows, and content views



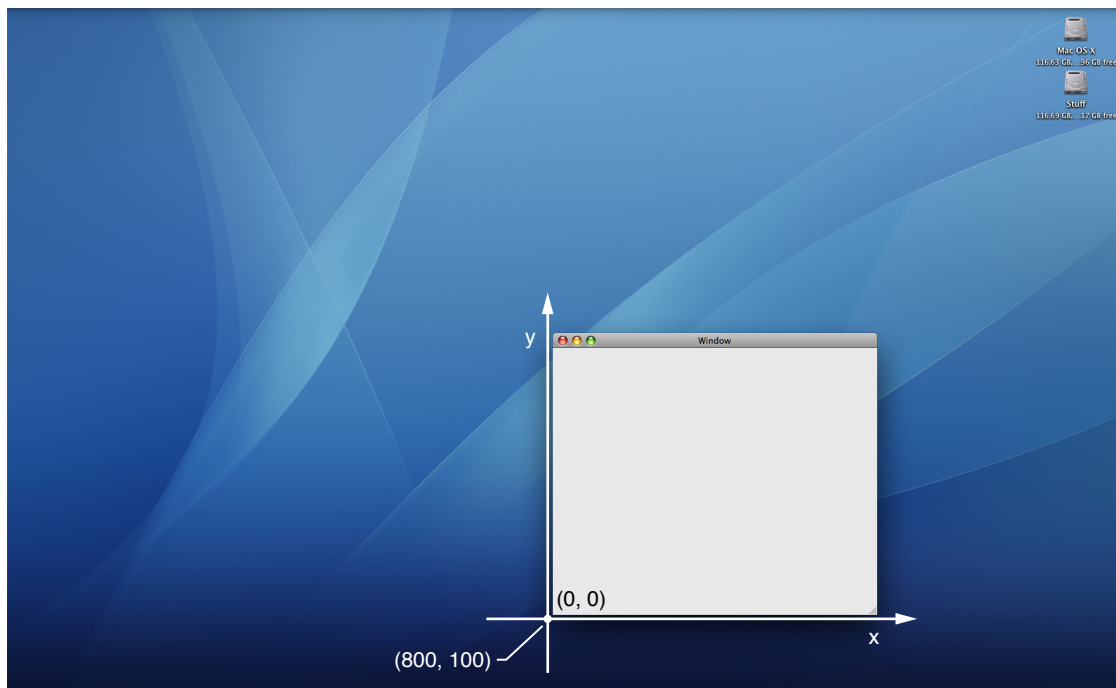
Window Coordinates

The coordinates of a window are related to the coordinates of the screen. The entire screen can be thought of as the upper-right quadrant of a two-dimensional coordinate grid, with the origin in the lower-left corner and the positive x-axis extending horizontally to the right and the positive y-axis extending vertically upward (see Figure A-9). You can locate a point in the screen by reference to this grid.

Figure A-9 Screen coordinates

The primary function of the screen coordinate system is to position windows on the screen. When your application creates a new window and puts it on the screen, it must specify the window's initial size and location in screen coordinates. However, for *drawing* in a window, it uses a different coordinate system, one that's specific to the window: its base coordinate system (see Figure A-10). This coordinate system differs from the screen coordinate system in two respects:

- It applies only to a particular window; each window has its own base coordinate system.
- The origin is at the lower-left corner of the window rather than at the lower-left corner of the screen. If the window moves, the origin and the entire coordinate system move with it. An image retains its position within a base coordinate system no matter where the window is located.

Figure A-10 Base coordinate system

The base coordinate system is the reference point for defining the individual coordinate systems of the window's views. The frame view draws the window's border and title bar directly in the base coordinate system. The content view and its subviews draw in coordinate systems that are transformations of the base coordinate system.

Windows and Drawing

Windows do not draw themselves. That job is left to the views they contain. However, an `NSWindow` object plays an important role in coordinating the drawing of its views.

Normally, during a pass of the event loop objects in an application may mark views (or regions of views) as needing to be redrawn. An `NSWindow` object collects references to these "dirty" views in a list, the order of which is determined by position in the view hierarchy. Shortly afterward—which is usually at the end of an event cycle—the Application Kit iterates through this list starting with the topmost view (that is, the view closest to the content view) and requests each view to draw itself. In this manner, views that are a background to other views are drawn first.

This automatic drawing of a window's views takes place only if the window's auto-display feature is turned on, which it is by default (see the `setAutodisplay:` method). If you turn off this feature, the application is responsible for updating window content when necessary. In addition, you can bypass the auto-display mechanism by sending an `display`, `displayRect:`, or `displayRectIgnoringOpacity:` message to any `NSView` object; as a result that view and its subviews are immediately redrawn. See [“Displaying a View”](#) (page 223) for further information.

You can redraw an entire window with the `NSWindow` `display` and `displayIfNeeded` methods. These methods force an immediate display of the window's views, although the latter method iterates only through the list of invalidated views. The `display` method causes each view in the window's view

hierarchy to redraw itself, starting with the content view. This method is invoked on a one-shot window (that is, a window whose backing store is released when the window is offscreen) before the window is shown onscreen. In addition, you can call `setFrame:display:` with a second argument of `YES` to cause the window to resize and to redraw all of its views.

A mechanism related to window display is window updating. At every pass through the event loop, `NSApp` sends an `update` message to each window in its window list. The default implementation of `update` does nothing, but subclasses of `NSWindow` can override this method to examine the state of the application and modify the behavior or appearance of the window as needed.

Window Status

Each window of an application has a status related to the user's interaction with it. The appearance of a window is an indication of this status. Inactive windows are open and may be visible, but they are not in the foreground. The controls and title in the title bars of inactive windows are grayed out. Users cannot interact with these windows unless they click on them to bring them forward.

Active windows are windows that are currently the focus of user input or attention. They are in the foreground, their controls have color, and their titles are in a black font. Active windows can have two statuses: main and key. The active window that is currently the focus of user attention is the main window. It is also the key window much of the time; a window is key if it currently accepts keyboard events.

However, sometimes the main window and the key window are separate windows. While the main window is still the focus of user attention, another window is the input focus for key events. The key window must have an object such as a text field in which the user can enter characters by typing at the keyboard. In this case, the key window is typically a dialog or panel (for example, the Find dialog) in which the user specifies data relevant to the main window.

The application object (`NSApp`) maintains the main and key status of the application's windows. A window's status often determines the events it receives and dispatches (see [“More About Event Dispatch”](#) (page 204)).

Further Reading: For more on main and key windows, see the section on window behavior in *Apple Human Interface Guidelines*.

Windows and Event Handling

`NSWindow` is involved with events and event handling in two principal ways. In one activity—event dispatch—it is actively involved. In the other, it is the passive recipient of a constrained stream of events.

Event Dispatch

As described in [“More About Event Dispatch”](#) (page 204), the application dispatches most events it receives by sending `sendEvent:` to the `NSWindow` objects to which the event “belongs.” The window object, in turn, locates the `NSView` object that should receive the event and sends the appropriate `NSResponder` message to that view, passing in the `NSEvent` object. For example, if the event is a

key-down event, it sends `keyDown:` to the view; if it's a mouse-dragged event (left button), it sends `mouseDragged:`. How the window object locates the target view generally differs between key and mouse events:

- The window sends key events to the first responder of the view hierarchy.
- The window sends mouse events to the view in which the mouse event occurred.

The key-down (`NSKeyDown`) and left-mouse-down (`NSLeftMouseDown`) events require the most handling. Before `NSWindow` attempts to send the `keyDown:` message to the first-responder view, it feeds the key character (or characters) to the system input manager, which interprets the input as text to insert or commands to perform. Before it sends a `mouseDown:` message, `NSWindow` attempts to make the target view the first responder. If the appropriate modifier keys are pressed, it doesn't send the message at all, instead handling it by displaying contextual help or a contextual menu.

Modal Windows

Normally an application distributes events to all of its windows, guided primarily by where user actions occur. But sometimes an application may run a window modally, thus requiring users to complete a task—for example, selecting a file, entering a name, or even clicking an OK button—before they can dismiss the window. Modal windows are common in Mac OS X; they include error-message dialogs and panels for opening and printing documents.

`NSWindow` objects are passive participants in the modal-window mechanism; it's the application that programmatically initiates and manages modal behavior. To run a window modally, `NSApp` uses its normal event loop machinery but restricts input to a specific window or panel. In the loop, it fetches events but if the events do not meet certain criteria—most importantly their association with the modal window—it tosses them.

`NSApplication` offers a couple of ways to run a window modally:

- **Blocking**—The application blocks until the user dismisses the modal window.
- **Nonblocking (modal session)**—An application initiates the modal session and runs the window modally for one pass through the event loop. The modal-session code can continue to run the window modally in a loop until some condition is satisfied.

Important: In a non-blocking loop, your code should do some work in between the invocations that run the window modally. Otherwise, it puts the application in a tight polling loop rather than a blocking loop.

Other Application Kit classes also provide methods to run windows and panels modally.

Panels

Panels are secondary windows that act in a supporting role to an application or document window. They are frequently referred to as dialogs. In Cocoa, panels are instances of `NSPanel` or of a subclass of `NSPanel`. Panels have some special behaviors suitable to their auxiliary function. They can become the key window, but never the main window. By default, they are removed from the screen when the application becomes inactive, and are redisplayed when the application becomes active again.

(Alert dialogs are an exception to this behavior.) And, because a panel is intended for repeated use, it isn't released when it's closed. You can also configure panels to become floating windows—such as utility windows—which are in a level above the other windows of an application.

Views

An `NSView` object (or simply, a view) occupies a rectangular area in a window. In Cocoa, views are instances of a subclass of `NSView`. They are the most pervasive type of object in the Application Kit; nearly every object you see in a Cocoa application is a view. Views are in the front line of both drawing and event handling, and hence are one of the more important types of objects to understand.

You can think of the drawing a view performs on the screen as a visual representation of the object itself. In a very real sense, a view draws itself. It also provides a surface that can be responsive to input from a mouse, keyboard, or other input device.

Further Reading: See *Cocoa Drawing Guide* for a comprehensive description of concepts and tasks related to drawing with `NSView` objects. Also, *View Programming Guide for Cocoa* describes various tasks related to the manipulation of views.

Varieties of Views

`NSView` is a class that defines the basic drawing, event-handling, and printing architecture of an application. `NSView` itself does not draw content or respond to user events, so you typically don't interact with a direct instance of `NSView`. Instead you use an instance of a custom `NSView` subclass. A custom view class inherits from `NSView` and overrides many of its methods, which are invoked automatically by the Application Kit.

If you look at the class hierarchy of the Application Kit ([Figure 1-10](#) (page 41)), you'll notice a proliferation of classes inheriting, directly or indirectly, from `NSView`. These classes fall into several categories:

- **Controls.** Controls are views that users manipulate (for example, by clicking or dragging) to indicate a choice to an application. Buttons, sliders, text fields, and steppers are examples of controls. Controls usually (but not always) work in concert with cell objects, which do not inherit from `NSView`. [“Controls and Menus”](#) (page 230) discusses controls at more length.
- **Container views.** Some views are used to enclose and present other views or more primitive data. They may allow the editing of data or a more efficient presentation of a user interface. Among these kinds of views are `NSTextView`, `NSImageView`, `NSBox`, `NSSplitView`, and `NSTableView` objects.
- **Compound views.** Some views are composed of other views. When you see a text view in a Cocoa application, it includes not just an `NSTextView` object but also an `NSClipView` object and an `NSScrollView` object (which itself includes `NSScroller` objects). Another example are table views (instances of `NSTableView`), which have constituent objects for table headers and table columns (the latter of which is not a view).
- **Wrapper views.** A few views act as a Cocoa “host” for a Mac OS X technology. Examples of these objects are instances of `NSOpenGLView` and `NSMovieView`.

There is some overlap between some objects in these categories. For example, an `NSTableView` object is a compound object, but it is also a control.

The View Hierarchy

As you may recall from the earlier discussion of windows ([“Windows”](#) (page 206)), each view object is associated with the window in which it is displayed. All of the views in a window are linked together in a **view hierarchy**. Each view has another view as its **superview** and may be the superview for any number of **subviews**. At the top of the hierarchy of views is the window's content view, which has no public superview (it does have a private one). The key visual characteristic of the view hierarchy is enclosure: a superview encloses its subviews, which are positioned relative to it. Figure A-11 illustrates this enclosure.

Figure A-11 View hierarchy



Arranging views in a hierarchy is beneficial for both drawing and event handling. It benefits drawing in three ways:

- It permits a complex view to be constructed out of other views. For example, a graphical keypad might be a container view with a separate subview for each key.
- It also permits each view to have its own coordinate system for convenient drawing. Views are positioned within the coordinates of their superviews, so when a view object is moved or its coordinate system is transformed, all of its subviews are moved or transformed with it. Because a view draws within its own coordinate system, its drawing instructions can remain constant no matter where it is or where its superview moves to on the screen.
- It is used to set the layering order of views rendered in a drawing pass (see [“Displaying a View”](#) (page 223)).

Note: Don't confuse the view *instance* hierarchy with the view *class* inheritance hierarchy. The view inheritance hierarchy is the arrangement of classes based on shared attributes, interface, and behavior. A view instance hierarchy is an arrangement of particular view instances based on enclosure.

The view hierarchy (referring now to the view instance hierarchy) plays an important role in event handling because it is a major part of the responder chain. See [“Responders and the Responder Chain”](#) (page 228) for more about the responder chain.

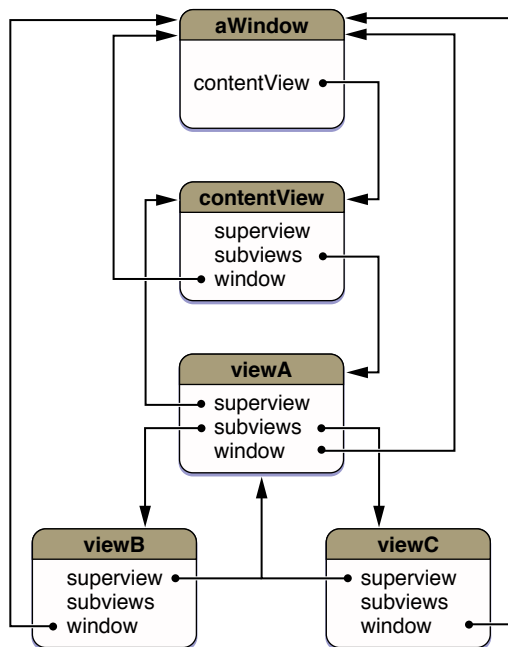
The view hierarchy is dynamic: As an application runs, you can rearrange views, add them, and remove them. You can move a view from one window to another, and you can move a view around a particular hierarchy.

`NSView` has three relationship properties that help to locate a view in the view hierarchy:

- `window`—the window (`NSWindow` object) in which the view appears
- `superview`—the view that's immediately above the view in the hierarchy
- `subviews`—the list of views that contained by the view (the list could have zero or more views)

Reflecting these properties, Figure Figure A-12 diagrams the relationships of a window object and its view hierarchy.

Figure A-12 Relationships among objects in a view hierarchy



View Geometry and Coordinates

View geometry is largely defined by two rectangles associated with each view: its frame and its bounds. Although these rectangles circumscribe the same area, they have different purposes. Together they help to define a view's position and size, and the coordinate system in which it draws and responds to events.

Note: Dimensions and locations described with these rectangles are expressed in floating-point values.

The Frame

The frame rectangle defines the area of a view, the tablet on which it can draw. If you think of a view as a rectangular area on a window, the frame specifies the dimensions of this rectangle and its location in the window. A view can draw only within its frame; by default, the Application Kit enforces clipping of the content the view draws to the view's frame.

As shown in Figure A-13, the frame rectangle of a view usually lies within the frame rectangle of its superview. But it doesn't have to. A view's frame rectangle can extend outside its superview's frame, but its drawing is clipped to its chain of containing ancestor views. The only part of the view that's visible on the screen is that part within the view's frame rectangle and within the frame rectangles of all its ancestor views.

Figure A-13 Hierarchical views

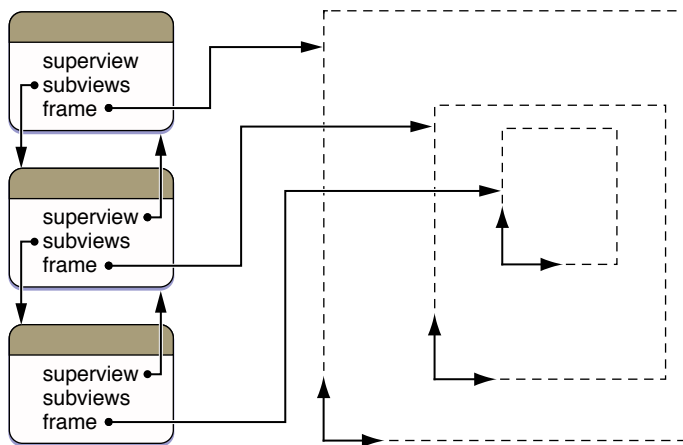
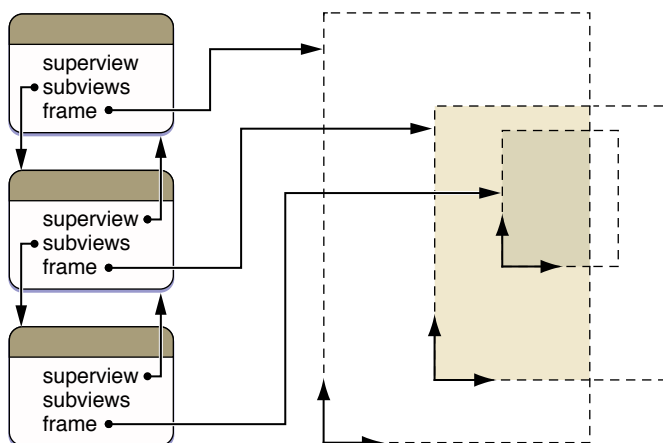


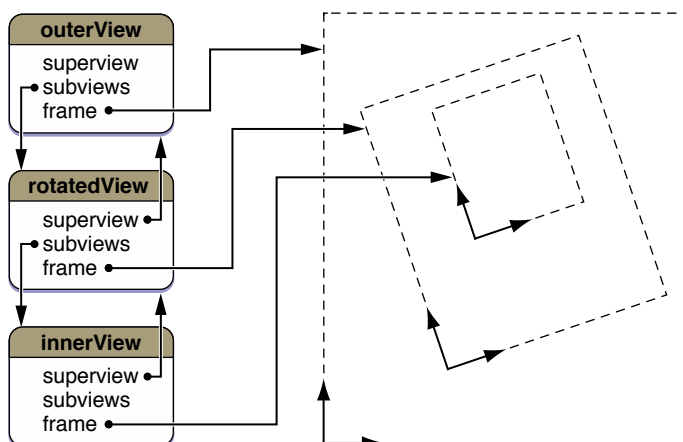
Figure A-14 also shows three hierarchically related views. In this case, however, the middle view lies partially outside its superview's frame rectangle. Although the lowest view lies entirely inside its superview, it also lies partially outside an ancestor view, so only the colored portion of it is visible.

Figure A-14 Views clipped by the superview


Some views may contain more material than a window has room enough to display—a view containing the contents of a long document, for example. Such a view may be made the subview of another, smaller view that exposes only part of it. This smaller view is known as a clip view (an instance of `NSClipView`). With the aid of enclosing scroll view (`NSScrollView`) and the scrollers (`NSScroller`) that it manages, users can control the visible portion of the document within its clip view. As the subview moves, different portions of it are scrolled into view.

You move the location of a view within its subview by resetting its frame's origin. You change the size of a view by altering the size of the frame rectangle. Because these values are interpreted according to the superview's coordinate system, the size and location of a view on a screen can also change if the superview's coordinates change.

You can rotate a view around its frame origin. Rotation doesn't affect the shape or size of the view; it remains a rectangle even though it has been turned at an angle and the sides of its frame are no longer parallel to its superview's x-axis and y-axis. The frame's origin stays at the same point regardless of the angle of the frame's rotation. Subviews of a rotated view remain aligned with it and hence are also turned in relation to the superview of the rotated view. Figure A-15 illustrates the same three views shown in Figure A-13 (page 219), but here the view in the center of the hierarchy has been rotated.

Figure A-15 Rotated view and its subview


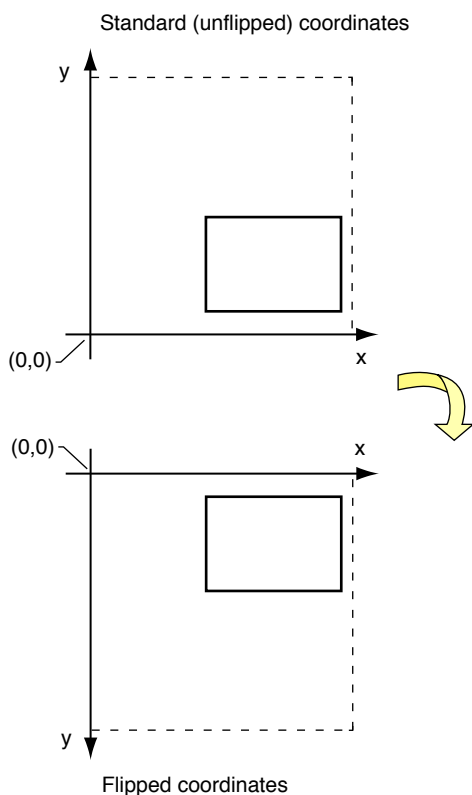
The Bounds

While a view's frame provides the means to size and position the view within its containing superview, it is of little use to the view for drawing. A view performs all drawing and event handling in its own local coordinate system, which is defined by its bounds rectangle.

A view's bounds rectangle defines the way that its local coordinate system is mapped onto the view's area. It describes exactly the same physical area as the frame rectangle, though this area is stated in the view's local coordinate system. By default, a view's bounds rectangle is identical in size to its frame, and has an origin of (0.0, 0.0). Under this arrangement, the view positions and draws its content using positive-valued coordinates.

However, If the view is flipped, this situation changes. A view can flip its coordinate system so that the drawing origin is in the upper-left corner of the view and the positive y-axis extends downward. Figure A-16 shows what a flipped coordinate system looks like. Flipped views are especially useful for applications that draw text in languages such as English, where text starts in the upper-left corner of a document and flows right and then down.

Figure A-16 Flipping a view



Views typically use the bounds rectangle to ensure that they don't fruitlessly attempt to draw something that will never be rendered on the screen. Because drawing is clipped if it falls outside of any ancestor view, the bounds rectangle by itself is a reliable guide only for views that aren't scrolled and stay entirely within the frame rectangles of all their ancestors. The `NSView` class provides other programmatic ways to determine where to draw, but the bounds rectangle remains part of any drawing calculation.

Drawing Coordinates

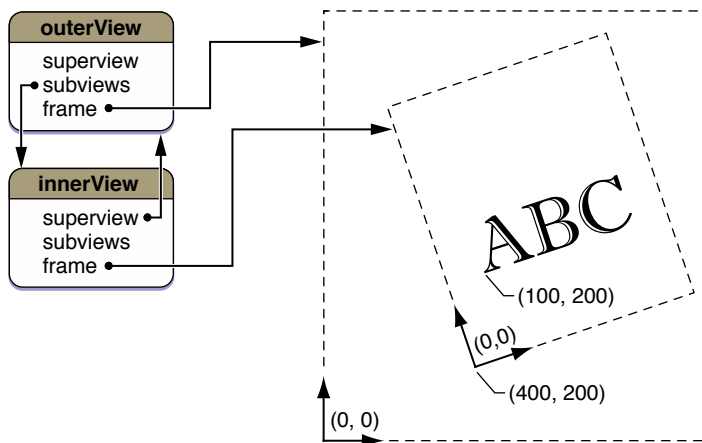
The bounds rectangle provides a view with its drawing coordinates. Before a view draws itself, its coordinate system is made the current coordinate system for the application (see “How Views Get Drawn” (page 223) for details). Recall that the default coordinate system for a view is the same as its superview's, except for these differences:

- The point that locates the view's frame in its superview is made the origin (0.0, 0.0) of the bounds rectangle, and hence is the origin of the drawing coordinates.
- If a view's frame is rotated, its drawing coordinate system is rotated with it; the bound's x-axis and y-axis remain parallel with the frame's axes.

If a transformation is performed on the bounds, however, these details can change.

Figure A-17 illustrates the relationship between a view's default coordinate system and the coordinate system of its superview. The `innerView` object in this diagram is located at (400.0, 200.0) in its superview coordinate system. For any drawing that `innerView` does, this same point is treated as the coordinate origin (0.0, 0.0). When `innerView` draws text beginning at (100.0, 200.0), as it does in this example, the point is measured from its own origin, not its superview's. Even if `innerView` is rotated, as shown in the diagram, the bound's axes rotate with the frame; the origin of the view's drawing coordinates remain unchanged.

Figure A-17 Relation between coordinate systems of view and superview



A view can modify its default local coordinate system in several ways:

- It can translate its origin to a point other than that defined by the frame's origin.
- It can scale the size of units along its bounds axes to a value different than its superview's.
- It can rotate the axes of its bounds rectangle around the bounds origin so that they're no longer aligned with the frame's axes.

These modifications alter the coordinate system that the view draws in, and may affect the appearance of what's drawn, but they don't change the area where the drawing appears. In other words, they don't affect the view's frame rectangle.

How Views Get Drawn

Views are the primary objects responsible for drawing window content in a Cocoa application. Other Application Kit objects can draw (such as `NSCell`, `NSBezierPath`, and `NSAttributedString` objects), but they require an `NSView` "host" object to provide the drawing surface and coordinate the drawing. The following sections give a high-level overview of how the Application Kit coordinates the drawing of a view.

Displaying a View

A view typically doesn't draw itself whenever it wishes. It must be explicitly marked as invalid (or "dirty") and thus in need of display. Redrawing of the view can then either take place immediately or, if the `NSWindow` auto-display feature is turned on, it can be deferred until the conclusion of the current cycle of the main event loop. Windows have auto-display enabled by default. (Auto-display is described in ["The Main Event Loop"](#) (page 202) and in ["Windows and Drawing"](#) (page 213).)

You can initiate immediate redrawing of a view or a portion of a view with one of the `NSView` display methods (so-called because each of these methods has "display" embedded in its name). These methods differ in various respects but they all result in the Application Kit doing the following things:

1. Locking focus on the invalidated view (described in ["Locking Focus"](#) (page 224))
2. Invoking the view's `drawRect:` method (described in ["What Happens in drawRect:"](#) (page 226))
3. At the conclusion of a drawing pass, flushing the window associated with the view (if the window's backing store is double-buffered)

Rather than displaying views immediately, the recommended course for most situations is to use the auto-display mechanism and during an event cycle mark views (or parts of views) as needing display. The `NSWindow` associated with the views collects the marked views in a list ordered by position in the view hierarchy, topmost view first. At the end of the event cycle, it recursively processes this list in one drawing pass, locking focus on each view in turn and asking the view to draw itself entirely or the portion of the view that was marked as needing display. When all views have been drawn, the window (if buffered) is flushed.

The Application Kit may request drawing of additional views and view areas beyond those that your application explicitly marks as needing redisplay. It may determine that additional view drawing is needed to fully update the corresponding window area. That's because an important aspect of the drawing of views is view opacity. A view does not have to draw every bit of its surface, but if it does it should declare itself to be opaque (by implementing `isOpaque` to return YES). When you mark a view for display, the Application Kit checks the opacity of the view, and if it is not opaque (that is, partially transparent) the Application Kit goes up the view hierarchy until it finds a superview that is opaque. It calculates the portion of the opaque ancestor covered by the original view. Then it draws forward through the hierarchy from this view to the view originally marked as dirty. If you want the Application Kit to not look for the first opaque ancestor before drawing a view, there are several "display ignoring opacity" methods you can use (which are listed in Table A-1 and Table A-3).

You can also mark portions of views and their subviews as needing display and then have them redrawn at once, instead of waiting for the auto-display mechanism to trigger. The `NSView` display methods that offer this feature all begin with `displayIfNeeded...` Even though display is immediate, these methods are more efficient than sending `display` or `drawRect:` messages to isolated views.

Table A-1 shows the `NSView` display methods for displaying individual views or regions of views immediately.

Table A-1 `NSView` display methods—immediate display

Display area and opacity	Display method
Entire view	<code>display</code>
Partial view	<code>displayRect:</code>
Entire view, ignoring opacity	None
Partial view, ignoring opacity	<code>displayRectIgnoringOpacity:</code>

Table A-2 lists the methods that mark views or regions of views for redrawing using the auto-display feature.

Table A-2 `NSView` display methods—deferred display

Display area	Display method
Entire view	<code>setNeedsDisplay:</code>
Partial view	<code>setNeedsDisplayInRect:</code>

Table A-3 lists the methods that force an immediate display of views (or portions of views) invalidated with the methods in Table A-2.

Table A-3 `NSView` display methods—Immediate display of marked views

Display area and opacity	Display method
Entire view	<code>displayIfNeeded</code>
Partial view	<code>displayIfNeededInRect:</code>
Entire view, ignoring opacity	<code>displayIfNeededIgnoringOpacity</code>
Partial view, ignoring opacity	<code>displayIfNeededInRectIgnoringOpacity:</code>

To reiterate, displaying views immediately is less efficient than using the auto-display feature, and immediately displaying only the marked parts of views falls somewhere in between. In addition, it is generally more efficient to mark areas of views as dirty rather than entire views. The display methods are a convenience for application development. A view can lock focus, draw itself, and unlock focus. But this is recommended only in certain circumstances, such as when animating content on a timer callback. In general, you should not try to circumvent the Application Kit's display mechanism.

Locking Focus

When the Application Kit—or your code—locks focus on a view, `NSView` completes the following set-up steps:

- It transforms the view's coordinate system from its superview's coordinate system and makes it the application's current coordinate system.
- It constructs a clipping path to define the rectangular area outside of which the view cannot draw.
- It activates other parameters of the current graphics state to establish the view's drawing environment.

You lock focus on a view by sending it a `lockFocus` (or related `lockFocus...`) message. After drawing is completed in a focused view, you unlock focus by sending it `unlockFocus`. The Application Kit automatically locks and unlocks focus when sending a view a `drawRect:` message.

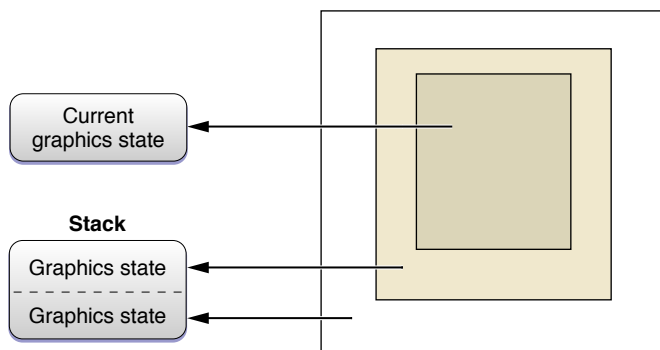
A focused view is one that is currently drawing. Every window has a window graphics context, which defines a window server destination for drawing. This graphics context contains one or more graphics states, each of which isolates drawing operations for a particular view. The current graphics state contains the effective coordinate system and other drawing parameters for the most recently focused view.

A graphics state includes the current transformation matrix, which is a mathematical construct used to map points from one coordinate space to another through rotation, scaling, and translation operations. A graphics state also defines the current clipping area and other parameters that drawing operations may consult when they render an image. These parameters include the following:

- The current color for fill and stroke operations
- Alpha value (transparency)
- Line attributes, including width, join, cap, dash, and miter limit
- Anti-aliasing value
- Current color space
- Text attributes: font, font size, character spacing
- Blend mode

It's possible to lock focus on a view when another view already has focus. In fact, this is what the Application Kit does during a drawing pass. Because each view keeps track of its own coordinate system as a modification of its superview's, a view is generally brought into focus after its superview. In a group of invalidated views, the Application Kit locks focus on the topmost view and works its way down the view hierarchy, making a nested series of `lockFocus` calls. As focus is locked on each successive view, the current graphics state is saved on a stack maintained by the graphics context (see Figure A-18). As focus is unlocked on a view, the graphics state at the top of the stack is "popped" and restored as the current graphics state.

Figure A-18 Nested focused views and graphics state stack



What Happens in drawRect:

After the Application Kit locks focus on a view, it sends it a `drawRect:` message. A view's class implements this method to draw the view. A rectangle is passed in to `drawRect:` that defines the area to be drawn in the view's coordinate system. This rectangle might or might not correspond to the bounds of the view. It could be the union of all rectangular regions of the view that have been marked as invalid, or it could be a superset of that.

The sending of drawing instructions and data to the window server has a cost that should be avoided when possible, especially for drawing that will not end up being visible. A major Application Kit optimization is restricting the area to draw in, especially when the drawing is complex. The view can choose to draw itself entirely—the least efficient of alternatives—or it can draw the area defined by the passed-in rectangle. A potentially even more efficient procedure for a view would be to obtain the list of invalidated regions (through the `NSView` method `getRectsBeingDrawn:count:`) and selectively draw each of these regions in turn.

In its implementation of `drawRect:` the view class invokes various functions and methods to specify the drawing instructions and data to be passed to the window server. The Application Kit provides the following high-level drawing functions and methods:

- Drawing functions (declared in `NSGraphics.h`) to draw, fill, erase, and perform other operations with rectangles
- Methods to construct lines and shapes with bezier paths (`NSBezierPath` class)
- Methods to create and apply affine transforms, involving translation, scaling, and rotation operations (`NSAffineTransform` class)
- Color and color-space methods (`NSColor` and `NSColorSpace`)
- Methods for creating and compositing images (`NSImage` and various image-representation classes)
- Methods for drawing text (`NSString` and `NSAttributedString`)

The Application Kit implements these methods and functions using Core Graphics (Quartz) functions and types. A view can also use these Core Graphics functions natively when it draws itself. These Quartz client library functions map directly to rendering operators of the window server, resulting in a raster (bitmap) image that, for in-window drawing, becomes part of the window's backing store.

Threads and Drawing

The drawing of views does not have to occur on the main thread; each thread of an application has the ability to lock focus on a view and draw. However, there are the following qualifications:

- Mutations of the properties of an `NSView` object (for example, its frame rectangle) should happen only on the main thread.
- When an `NSView` display method is invoked, the Application Kit grabs a lock for drawing in the receiving view's window; you cannot perform any custom drawing until the display method returns. This means that only one thread can draw into a given window at a time.

Views and Printing

Views are the foundation of the Cocoa printing architecture. They provide the content to be printed, just as they provide content to be displayed on a screen. The general procedure is the same: The Application Kit locks focus on the view, its `drawRect:` method is invoked, the view draws the printable content, and focus is unlocked. You can tell a view to print itself by invoking its `print:` method.

Views and Events

Views are the objects in an application that respond directly to most user events such as mouse clicks or key presses. They are almost always the object providing the surface on which a user event occurs. Consequently, these front-line objects are given the first shot at handling event messages.

As discussed in [“Windows and Event Handling”](#) (page 214), a window forwards a user event (as an event message) to the view in its view hierarchy that should receive the event. For mouse events, it sends the message to the view under which the event occurred. If the event is a key event, the window sends the message to its first responder, which is usually the view with key focus. In order to receive the key event, the view must declare that it accepts first-responder status (that is, it overrides the `NSResponder` method `acceptsFirstResponder` to return YES).

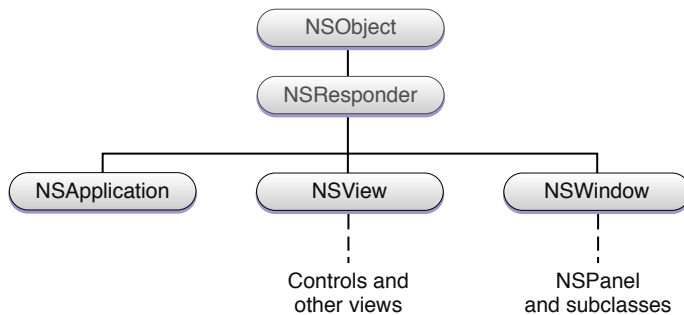
Views are responder objects, which means they inherit the programmatic interface of the `NSResponder` class. Event messages invoke methods declared by this class; examples of such messages are `mouseDown:`, `mouseMoved:`, and `keyUp:`. To handle an event, a view's class must implement the appropriate `NSResponder` method; in its implementation it can inspect the passed-in `NSEvent` objects (which encapsulates information about the event) and proceed from there. If a view does not handle an event, the next responder up the responder chain—which is usually a superview—is given an opportunity to handle the event. [“Responders and the Responder Chain”](#) (page 228) describes responders and how events travel up the responder chain.

Views also play a major role with action messages. These messages often originate as event messages sent to `NSControl` objects; in handling them, a control object sends an action message to a target object. If no target is specified, the application searches up the responder chain for an object that can respond to the action message. For more on `NSControl` objects and the `NSCell` objects they use, see [“Controls and Menus”](#) (page 230).

Responders and the Responder Chain

Objects of the core application classes—`NSApplication`, `NSWindow`, and `NSView`—are responders. They are instances of classes that inherit, directly or indirectly, from `NSResponder` (see Figure A-19). This abstract class defines the interface and expected behavior of objects that can respond to events. The subclasses of `NSResponder` implement this behavior in whole or in part.

Figure A-19 `NSResponder` and its direct subclasses

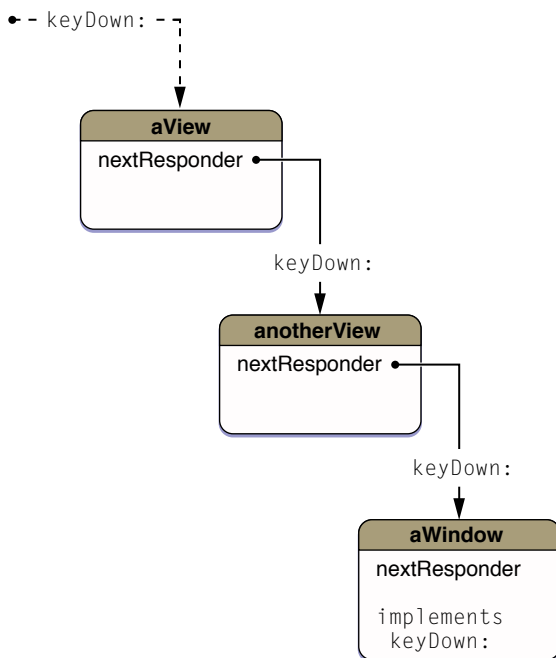


Note: Two other Application Kit classes directly inherit from `NSResponder`: `NSWindowController` and `NSDrawer`. Although `NSWindowController` is part of the Cocoa document architecture (described in “Other Cocoa Architectures on Mac OS X” (page 241)), neither of these classes is central to the core Cocoa application architecture.

The `NSResponder` class defines an interface for three major patterns or mechanisms of the core application architecture:

- It declares a number of methods for handling **event messages** (that is, messages originating in user events such as mouse clicks and key presses).
- It declares dozens of methods for handling **action messages** that are tied to the standard key bindings (for example, those that move the insertion point within text). An action message is dispatched to a target object; if no target is specified, the application searches for a suitable responder.
- It defines a set of methods for designating and managing the responders in an application. These responders form what is known as a **responder chain**—a series of responders along which an event or action message is passed until it finds an object capable of handling it.

The responder chain is a central mechanism in the Application Kit's architecture for event handling. It is a linked series of responder objects along which an event or action message is passed. As depicted in Figure A-20, if a responder object cannot handle an event or action—that is, it either does not respond to the message or the event is one it doesn't recognize—it resends the message to the next responder in the chain. The message travels up the chain, toward higher-level objects, until it is handled. (If it isn't handled, it is discarded.)

Figure A-20 The responder chain

When the Application Kit constructs the objects in a application, it sets up the responder chain for each window. The essential objects in a responder chain are the `NSWindow` object and its view hierarchy. Lower views in the hierarchy are given the chance to handle an event or action message before higher level objects. The `NSWindow` keeps a reference to the **first responder**, which is usually the currently selected view in the window, and gives it first shot at responding to the message. For event messages, the responder chain usually ends at the `NSWindow` object managing the window in which the event occurred, although other next-responders can be added after the `NSWindow` object.

For action messages, the responder chain is more complex. Two factors determine the chain of responders for action messages:

- If the application currently has both a main window and a key window, the responders chains of both windows are involved, with the key window's responder chain first given the chance at handling the action. At the end of each window's chain, the `NSWindow` delegate is given a chance to respond; at the end of the combined responder chains are `NSApp` and its delegate.
- The type of application—simple, document-based, or an application that uses a window controller—determines the type and location of responder objects in the chain.

The `NSResponder` class also includes declarations of methods for error presentation and recovery, message dispatch, application help, and other features.

Further Reading: To learn more about responders and the responder chain, see *Cocoa Event-Handling Guide*

Controls and Menus

Many of the objects you see in an application let you manipulate them to signal intent. These objects include buttons, checkboxes, sliders, table views, file-system browsers, and menus (including the application and pop-up menus). In Cocoa, similar architectures underlie the implementations of the two general kinds of choice-enabling objects: controls and menus. In these architectures, various types of objects, including `NSView` objects, work together to allow the indication of user choice or intent.

Control and Cell Architecture

A control is a user-interface object that responds to a user event such as a mouse click by sending a message to another object in an application. Common types of controls are buttons, sliders, and text fields (which typically send a message when users press the Return key). Other, less obvious controls are table views, data browsers, and color wells.

A control is an instance of a subclass of the abstract `NSControl`. It generally manages one or more cells—instances of a subclass of `NSCell`, another abstract class. If you look at the Application Kit class hierarchy ([Figure 1-10](#) (page 41)), you notice that `NSControl`, a subclass of `NSView`, is the root of a fairly large branch of control classes: `NSButton`, `NSStepper`, `NSTextField`, and so on. At an entirely different location in the hierarchy (under `NSObject`), `NSCell` is the start of a branch of cell classes, most of which correspond to a control class: `NSButtonCell`, `NSStepperCell`, `NSTextFieldCell`, and so on.

Note: Two control classes dispense with cells entirely: `NSScroller` and `NSColorWell`. Instances of these classes, like any control, enable the user to indicate a choice, but they behave like any non-control view in that they draw themselves and respond to user events without using cells.

Controls That Manage Multiple Cells

Most controls in the Application Kit manage a single cell. In these cases, the control has its cell do almost all of the actual work; it forwards many of the messages sent to it to its cell. But some controls manage multiple cells. The Application Kit takes two general approaches to managing multiple cells within controls:

- A single cell instance is used as a template for drawing. Whenever a control must draw a cell, it uses this instance to replicate each cell representation, varying only the content. The `NSTableView` and `NSOutlineView` classes take this approach when drawing the cells of table-view and outline-view columns.
- An individual cell instance represents each cell region drawn on a control. The `NSMatrix` class takes this approach when it requests its cell instances to draw themselves. `NSBrowser` and `NSForm` objects function in a similar way: their cells are also individual instances.

An `NSMatrix` object (or matrix) can manage most cell types. It arranges them in grids of arbitrary dimensions. When you construct a matrix of cells in Interface Builder, the cells are all copies of a given cell prototype. However, you can programmatically set the cells to be instances of different `NSCell` subclasses. `NSBrowser` and `NSForm` controls are more limited in the types of cells they can manage.

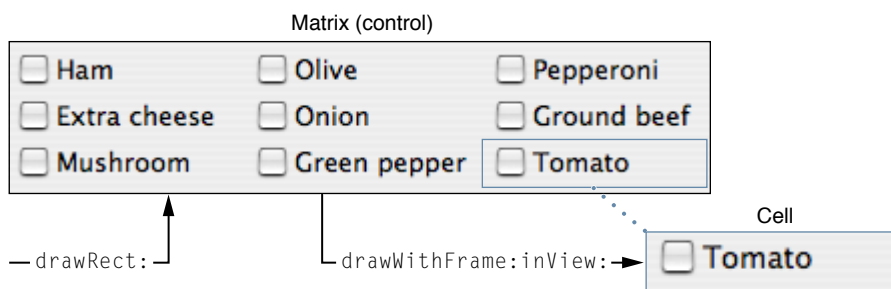
A single-cell control may seem to be a redundant design, but the value here is with the cell, not the control. See [“Rationale for the Control-Cell Architecture”](#) (page 232) for an explanation.

How Controls Manage Cells

Controls are full-fledged `NSView` objects. They can be marked as needing display, they are responsible for drawing themselves, they can respond to user events, and they are in the responder chain. At runtime, a control initially behaves like any view. The Application Kit asks it to redraw itself by invoking the control's `drawRect:` method. If an event message is dispatched to the responder chain and the control implements the appropriate `NSResponder` method, the control may have an opportunity to handle the event. However, instead of drawing and handling events themselves, controls pass on these responsibilities to their cells. If the control has multiple cells, it also coordinates the behavior of its cells.

As illustrated in Figure A-21, a control within its `drawRect:` method typically sends `drawWithFrame:inView:` to a cell, asking it to draw itself within the passed-in rectangle. Because focus is already locked on the control, the cell can draw itself using the surface of its "host" view. A control in an implementation of a `mouseDown:` event method sends a `trackMouse:inRect:ofView:untilMouseUp:` to the cell in which the event occurred. Typically, the cell tracks the mouse events within its border until either the mouse exits or the cell receives a `mouseUp:` event; then it responds appropriately. (Other event types are handled in a similar manner.)

Figure A-21 A control coordinating the drawing of a cell



When a cell draws itself, it must represent two aspects. The first is the general appearance of the cell, which is consistent among cells of the same class. These characteristics include cell shape, background color, style, and visual indications of cell state (for example, a checkbox for cell selection or gray text if the cell isn't enabled). The second aspect is the cell's content, which individuates the cell. For content, a cell usually has a title (a string), an image, or in some cases both title and image. The title can be in a specified font. (Some cells, like `NSSliderCell`, have custom content instead of image or title.)

As content (or part of its content) a cell has an object value and possibly a represented object. The object value must be an object that can be formatted as a string and thus displayed as the cell title—for example, an `NSNumber` object encapsulating a `float` value. A represented object, although not

displayed, is associated with the cell; for example, a button with a title of "Red" could have an `NSColor` represented object. For more on object values and represented objects, see [“Represented Objects”](#) (page 234).

What also individuates most cells is the information they encapsulate for action messages. The target-action mechanism (discussed in [“The Target-Action Mechanism”](#) (page 184)) allows a control to send a message to a designated object when the user activates the control (by clicking a button or pressing Return in a text field, for example). The action is a selector that identifies the method to invoke; the target is the designated object. (The target can be `nil`, which tells the application to search the responder chain for an object that can handle the message; see [“Responders and the Responder Chain”](#) (page 228).) The abstract `NSActionCell` class defines the interface for storing and retrieving action selectors and references to target objects. Most `NSCell` subclasses inherit from `NSActionCell`.

Rationale for the Control-Cell Architecture

The control-cell architecture in the Application Kit has deep historical roots, going back to the early days of NeXTSTEP. But the necessity for it might seem puzzling at first glance. Why have controls manage cells at all? Why can't controls do the required work themselves?

There's no reason why a control couldn't, and if you are designing a custom control, a cell-less control is certainly a valid approach. But the control-cell architecture brings some advantages with it:

- The architecture extends the usefulness of controls. A control such as a table view or a matrix can efficiently manage many different types of cells without having to have specific knowledge of each type. A cell such as an `NSButtonCell` object might be designed to work with `NSButton` controls, but it can also work with matrix and table-view objects.

A cell is an abstraction that simplifies the management of different kinds of graphical objects on a view. It allows for a kind of plug-in design where a control can host different kinds of graphical objects, each with its own identity, including target and action information.

- The control-cell architecture permits a tighter coupling between a control and its cells than between, say, a view and a collection of subviews. Subviews are largely autonomous within their superview; a control can better act as a coordinator of its cells. For example, in a matrix of radio buttons, the control ensures that only one of the buttons is on at any time.
- The `NSTableView` model for drawing cell regions—reusing a handful of cell instances as "rubber stamps" for drawing—is efficient, especially for controls that must manage a potentially unbounded number of subregions.
- Even in cases where it's not feasible to use a few cell instances as templates for drawing, cells often offer a performance advantage over subviews. Views are relatively heavyweight objects both in terms of memory and computation. For example, tracking and propagating invalid view regions through the view hierarchy can exact a significant cost.

There are also tradeoffs between the use of cells versus subviews for drawing subregions. In the absence of the view invalidation mechanism, the control must take responsibility for figuring out what each cell needs to draw on itself. But because views are general purpose objects, a specialized control can often do the required calculations more efficiently.

Note: See *Control and Cell Programming Topics for Cocoa* for more information about controls, cells, and the architecture enabling them to work together.

Menu Characteristics and Architecture

In addition to controls and cells, users can signal their intentions to an application (as well as to the operating system itself) using menus. A menu is a list of tersely worded choices—or menu items—with potentially nested menus (called submenus). Users typically choose a menu item by clicking it (although other methods are supported); as a result, a command is issued that the active application or operating system acts upon.

Cocoa supports various kinds of menus. Primary among these are the application-specific menus, which in Cocoa are collectively known as the main menu. At runtime, the application-specific menus include *the* application menu (the one bearing the application name) and all the menus to the right of it up through the Help menu. The application-specific menus share the menu bar with the Apple menu and menu extras—the service-specific menus to the right of the application-specific menus. The Application Kit automatically creates and manages certain application-specific menus, such as the Services menu, the Font menu, the Windows menu, and the Help menu. Other types of menus that Cocoa applications can manage include pop-up menus, contextual menus, and dock menus.

Menus and especially menu items have a number of interesting characteristics. Both menus and menu items have a title; the titles of menus are the strings that appear in the menu bar. Menu items can additionally have an image that appears to the left of the title, or they can have an image instead of a title. The title can be an attributed string, allowing different fonts and even text attachments (which permits an image to appear anywhere in the content area of a menu item). Menu items can also have an assigned key called the key equivalent which, when pressed together with modifier keys (except for Shift), causes the same response as a mouse click. Menu items are enabled or disabled, or they can indicate an on or off state; a check-mark appears to the left of the title if the item is in an on state.

Menus and menu items in Cocoa are instances of the classes `NSMenu` and `NSMenuItem`, respectively. In a Cocoa application, menus (in a generic sense) are based on a simple design that gives `NSMenu` and `NSMenuItem` objects complementary roles. An `NSMenu` object manages and draws a collection of menu items one below another. It contains an array of `NSMenuItem` objects representing this collection. An `NSMenuItem` object encapsulates all the data that characterizes a menu item but does no drawing or event handling itself. An `NSMenu` object uses the data in each `NSMenuItem` to draw the menu item within the boundaries of the menu, to track the location of the menu item within the menu, and to send an action message to a target object when users choose the menu item. When drawing, an `NSMenu` object uses an `NSMenuItem` object's title and image; for tracking, it uses the item's index; for sending action messages, it uses the action selector and target object stored in an `NSMenuItem` object.

Pop-up menus make use of this basic menu architecture. However, since they appear within an application's user interface, they have an additional design. Before a user clicks it, a pop-up menu appears as a button-like object. This object is an instance of `NSPopUpButton`, which manages an `NSPopUpButtonCell` instance—in other words, the control-cell architecture is used for this initial presentation. The `NSPopUpButtonCell` object contains an `NSMenu` object along with its encapsulated `NSMenuItem` objects. When a user clicks a pop-up button, this embedded menu is asked to display itself.

The menu items in a menu can be validated for the current context and, if an item is not relevant to that context, it can be disabled. `NSMenu` includes an auto-enabling feature that performs this validation automatically. Before a menu is displayed, it searches the responder chain for an object that can

respond to a menu item's action message; if it cannot find any such object, it disables the item. Applications can further refine menu validation by implementing the `NSMenuValidation` informal protocol.

Contextual menus are implemented in way that's similar to pop-up menus. You can attach an `NSMenu` object (along with its menu items) to a view using the `NSResponder` method `setMenu:`. This menu lists commands specific to the view (which can be validated according to context). When users Control-click or right-click in the view, the menu is displayed.

Further Reading: *Application Menu and Pop-up List Programming Topics for Cocoa* discusses Cocoa menus and menu items in more detail.

Represented Objects

A cell and menu item can have a represented object, an object that is arbitrarily associated with it. The target of an action message can ask a clicked menu item or cell (`sender`) for its represented object; the target can then display the represented object, load it, or perform whatever operation is required with the fetched object. A cell or menu item allows clients to access a represented object, and it archives and restores it, but otherwise does not use it.

To understand how you might use represented objects, consider a couple of examples. A matrix contains cells for setting the background color of a text view. These cells have names such as "Light Blue", "Light Gray", "Pink," and so on; the represented object for each menu item is an `NSColor` object encapsulating the RGB components for that color. Your application might use that represented object as in Listing A-1.

Listing A-1 Using a represented object

```
- (void)changeColor:(id)sender {
    NSColor *repObj = [sender representedObject];
    [textView setBackgroundColor:repObj]; // textView is an outlet
}
```

Another example would be a pop-up menu in an Info window that allows you to change the displayed pane of settings. Assigned as a represented object to each menu item is an `NSView` object containing the required set of text fields, controls, and so on.

A represented object is not the same thing as the object value of a control or cell. A represented object is arbitrarily associated while an object value is the value behind what the cell or control displays. For example, an object value for a text field that displays "05/23/2006" would be an `NSDate` or `NSDateFormatter` object representing that displayed value. A cell or control's formatter must be able to "understand" an object value. (A formatter is an `NSFormatter` object).

Represented objects are not strictly limited to cells and menu items. For example, `NSRulerMarker` objects can have represented objects. And you can design your own custom views to have represented objects.

Nib Files and Other Application Resources

So far the discussion in this chapter has focused on the fundamental application architecture of Cocoa, describing how at runtime the core objects of an application work together to facilitate event handling and drawing. But now the focus shifts somewhat to take in the full picture of a running Cocoa application. Rarely does an application create all of its constituent objects "from scratch" when it launches. Many, if not most, of these objects are stored in the application bundle as archives of object graphs. These object graphs can represent the model objects that encapsulate the application's data as it existed just before the user quit the application. Or they can be encoded representations of the windows, views, and other objects that make up the application's user interface. An application loads and unarchives object archives at runtime to re-create the original objects.

An application includes not just objects and code but the other resources in its bundle, such as images and localized strings. This section also summarizes the role that instances of the `NSBundle` class plays in locating and loading all types of application resources, both localized and non-localized.

Object Archives

Objects in a program exist in a network of relationships with other objects. An object can own particular objects or collections of objects, it can be dependent on other objects, and it can hold references to objects in the program in order to send messages to those objects. This web of interrelated objects is known as an object graph. Object graphs can become quite complex.

An archive is a means for storing an object graph. It usually takes the form of a file but can be a stream transmitted between processes. An archive preserves the identity of each object in the graph and all the relationships that object has with all other objects in the graph. It encodes the type of each object along with the object's data. When an object graph is unarchived, each decoded object is typically of the same class as the object that was originally encoded into the stream. An object's instance data is also decoded and is used to reconstitute the object. The relationships among objects in the graph are also restored. As a result, an unarchived graph should almost always duplicate the original object graph.

Note: Many Cocoa applications use archives as a persistent store for their model objects. However, the Core Data framework (a Cocoa framework) uses a more sophisticated mechanism for object persistence. See [“Other Cocoa Architectures on Mac OS X”](#) (page 241) (in this document) and *Core Data Programming Guide* for more information.

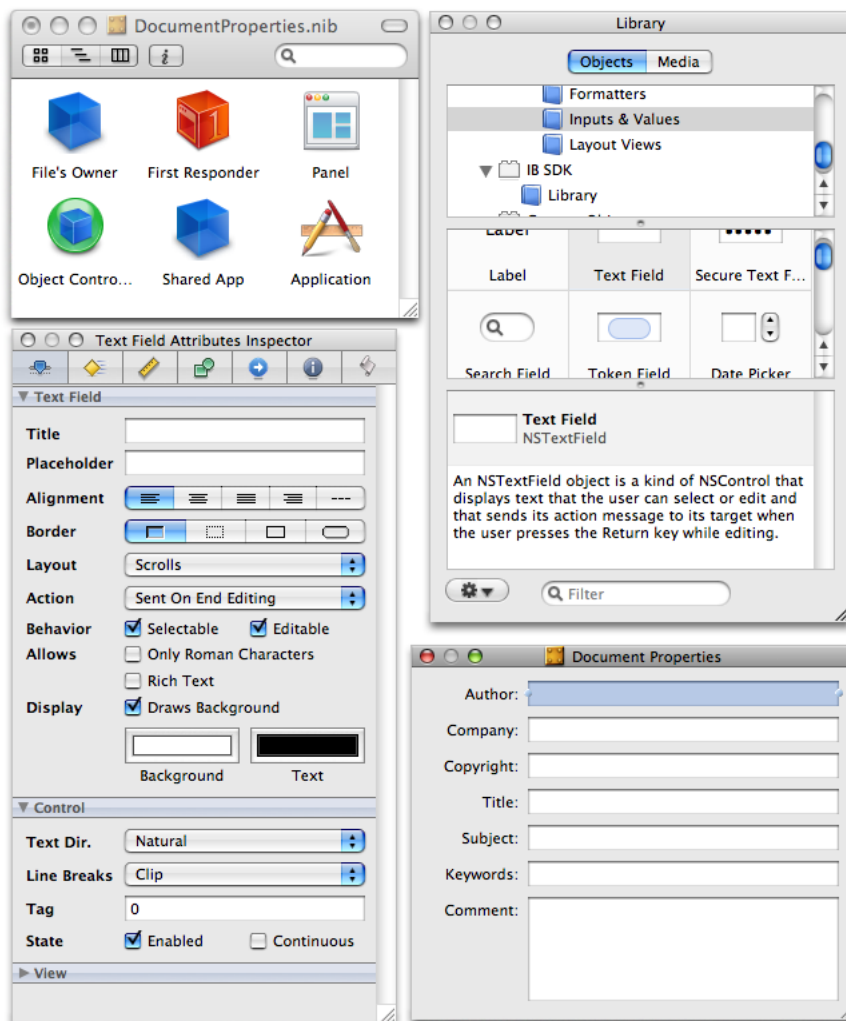
Archiving is an operation that requests each object in an object graph to encode itself into a stream; unarchiving does the reverse, requesting that each object decode itself. Both operations are initiated with a message sent to the root object of the graph. An object that wants to be saved to an archive must encode itself when asked to; to be restorable, it must be able to decode itself. The two types of archiving in Cocoa, sequential and keyed, reflect different styles for encoding and decoding objects. Sequential archiving requires objects to encode and decode their instance data in the same order. Keyed archiving (a more modern approach) permits each piece of stored instance data to be stored and retrieved using a key (that is, an identifying string). A class that wants to archive its instances must conform to the `NSCoding` protocol. An object that does sequential archiving uses the encoding and decoding methods of the `NSCoder` class. For keyed archiving (and unarchiving), it must use the methods of the `NSKeyedArchiver` and `NSKeyedUnarchiver` classes.

Further Reading: To learn more about archiving and unarchiving objects, see *Archives and Serializations Programming Guide for Cocoa*.

Nib Files

Nearly all Cocoa developers use the Interface Builder application to construct the user interface of their applications. (There's no requirement to use Interface Builder, but it makes the job of a developer so much easier.) Figure A-22, which you might recall from an earlier chapter, shows a typical arrangement of Interface Builder windows. To create a user interface, you drag objects such as text views, buttons, and table views from from palettes and drop them on windows; you then position and size the objects and set their other attributes. You can also make various kinds of connections—outlets, target-action, and bindings—between these objects. Interface Builder also lets you make the initial definitions of custom model and controller classes for the purpose of making connections to and from proxy instances of those classes. And, using Interface Builder, you can specify a placeholder object assigned to a custom `NSView` subclass. When you finish work on a user interface in Interface Builder, you can save it as a **nib file** in the project as a localized resource. When the project is built, the nib file is copied to the same localization (`.lproj`) folder in the application bundle.

Figure A-22 Interface Builder



Nib files are archives whose object graphs describe an entire or partial user interface. These graphs represent the complex relationships within a user interface, including a window's view hierarchy and the various connections between objects. The descriptions of these object graphs use XML as a representation language. (You should never attempt to directly edit the XML in a nib file.)

The root object of a graph appears in the Instances pane of the nib file window (the lower-left window in [Figure A-22](#) (page 237)). In this example, the Panel instance is a root object (since it contains a view hierarchy). A nib file could have other root objects as well as `NSController` objects (used for bindings) and proxy instances of custom classes. In addition, each nib file window of a Cocoa application has two special kinds of instances:

- **File's Owner.** An object that owns the nib file and manages the objects within it. The File's Owner must be external to the nib file. You use the File's Owner object as the conduit for connections between objects in the nib file and objects outside of it.

- **First Responder.** An object representing the first responder in the responder chain (see [“Responders and the Responder Chain”](#) (page 228)). In target-action connections you can specify the First Responder object as target; when a control or menu sends an action message, the application searches the responder chain (starting with the first responder) until it finds an object that can handle the message.

The standard objects on Interface Builder palettes are allocated and initialized when a user interface is built and later are archived when the objects are saved to a nib file. When the nib file is unarchived, these objects are restored. If you create a custom subclass based on the class for a standard palette object, Interface Builder encodes the superclass when it archives the object but has the custom class swapped in when the object is unarchived. In either case, the initializer for the unarchived object is not called. However, if you have a custom class of `NSView` (represented by the Custom View palette object), the initializer of the class is called when the Custom View object is unarchived. In any case, the application sends each custom class associated with a nib file an `awakeFromNib` message when all objects in a nib file have been unarchived; this message gives the class an opportunity to establish connections or perform any other set-up tasks.

Each Cocoa application has a main nib file that contains the application-specific menus and possibly one or more windows. `NSApp` is File's Owner of the main nib file. When an application launches, `NSApp` loads the main nib file, unarchiving it, and displays the menus and initial windows. Many applications have auxiliary nib files for such things as documents and panels; these nib files are loaded on demand (that is, when the user requests behavior provided by objects in the nib file).

Further Reading: You can find out more about nib files (including how to dynamically load them) by reading *Resource Programming Guide*.

Loading Application Resources

Nib files are an application resource, as are image files, sound files, help files, and other kinds of data. Application resources can be localized—that is, they can be adapted to multiple languages and locales. For nib files, localization primarily means translating the strings that appear in a user interface, but other modifications might also be required. Text fields, buttons, and other user-interface objects might have to be resized to accommodate the new strings. The formats of dates and numbers might also have to change.

Internationalization denotes the developmental infrastructure that supports localization. When internationalizing a software product, you must put resources for a language or locale in a particular location inside a bundle's `Resources` directory. This location is a folder whose name identifies a language and perhaps a locale, either as a well-known language string or an abbreviation conforming to the ISO 639-1, ISO 639-2, and (for locales) ISO 3166-1 specifications. The extension of localization folders is `.lproj`. Users compose their list of preferred localizations in the International pane of System Preferences, and an application selects resources from the `.lproj` folder that corresponds to the first matching localization. Xcode offers support for internationalizing application resources; it creates the bundle structure, including the `.lproj` folders, and automatically populates them.

Application resources can also be nonlocalized. Nonlocalized resources go just under the application bundle's `Resources` directory, outside of any `.lproj` directory.

At runtime, an application can locate an application resource within a bundle and load it into memory. It uses an instance of the `NSBundle` class for this purpose. Methods of this class, when given the name and extension of a resource, return the file-system path to that resource. Application Kit categories

on `NSBundle` allow an application to locate and load nib files, help files, and sound and image files. For example, the `NSBundle` class method `loadNibNamed:owner:` finds and loads a nib file that has the specified name and is owned by the referenced object. Loading resources dynamically this way is a programming practice that contributes to the overall efficiency of the application. An application should load resources into memory only when users request those resources.

An application can dynamically load more than file-based resources. It can programmatically load localized strings in certain contexts (for example, dialogs where the displayed message might vary). The localized string is fetched from a "strings" file (that is a file with an extension of `.strings`) stored in one of a bundle's `.lproj` directories. An application's main bundle can also contain subsidiary bundles, called loadable bundles. Loadable bundles can contain their own code (and resources), and an application, using `NSBundle` objects, can dynamically load this code into memory. Loadable bundles make it possible to extend application behavior flexibly through a kind of plug-in architecture. (The Automator application is an example of this architecture; it dynamically loads actions, which are loadable bundles.)

Further Reading: For more information about localization and internationalization, see *Internationalization Programming Topics*. To learn more about `NSBundle` and bundles, see *Resource Programming Guide* and *Code Loading Programming Topics for Cocoa*.

Other Cocoa Architectures on Mac OS X

Seen through the lens of Model-View-Controller, the core application architecture of Cocoa concerns itself with the view layer of applications. It deals primarily with the raw mechanics of event handling and drawing, providing an efficient model for how an application presents itself to users and allows users to interact with it. But in a real application, these things don't exist in a vacuum. The other kinds of MVC objects in an application—the controller and model objects—give significance and content to the mechanics of drawing and event handling.

Note: The MVC design pattern is described in [“The Model-View-Controller Design Pattern”](#) (page 161).

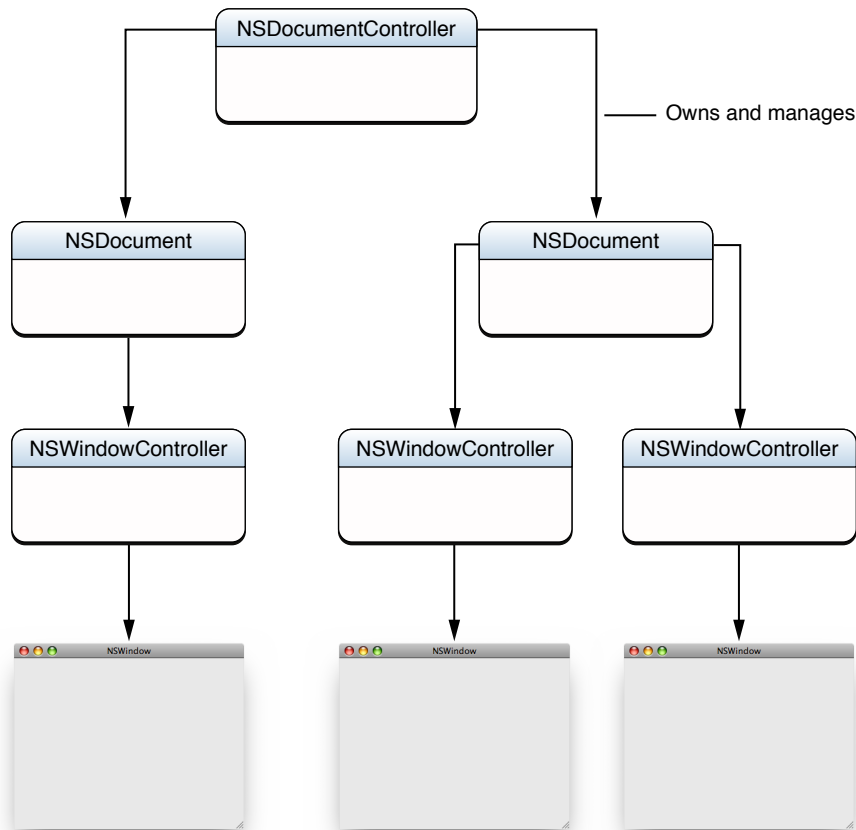
Cocoa makes several other architectures available to software developers. These architectures are all mostly concerned with the controller and (especially) the model objects of an application. They are based on several design patterns in addition to Model-View-Controller, particularly object modeling (described in [“Object Modeling”](#) (page 169)). Another thing they have in common is that they are intended to make the life of a Cocoa developer easier.

Document Architecture

Many applications let users create and edit documents: unique aggregations of data presented in windows with identical user interfaces. Word processors, photo-image editors, and web browsers are examples of document-based applications. These applications have similar features. They enable users to create new documents, save those documents to files, and then open those documents later. Document-based applications also validate menu items, monitor each document's edited status, manage document windows, and respond appropriately to application-wide events (such as termination). Often they can have different internal representations of document data.

Cocoa offers developers an architecture that reduces the effort required to implement a document-based application with features such as these. The essential component of this architecture includes three Application Kit classes: `NSDocument`, `NSDocumentController`, and `NSWindowController`. In a document-based Cocoa application, objects of these three classes have distinct spheres of responsibility and a cascading series of relationships with each other based on ownership and management (Figure B-1).

Figure B-1 Ownership relationships among the document classes



A Cocoa application that is based on the document architecture has a single `NSDocumentController` object. This object owns and manages one or more `NSDocument` objects. Each of these `NSDocument` objects, in turn, creates and manages one or more `NSWindowController` objects. And each `NSWindowController` object is associated with a window of the document. (A document can have multiple windows.) An `NSWindowController` object manages the presentation of the document.

Each of the three kinds of objects in the Cocoa document architecture has a specific responsibility which is dictated by its MVC role:

- **The `NSDocumentController` object manages the application's documents.** It initiates and coordinates the creation of new documents, the saving of edited documents, and the opening of saved documents. It plays the MVC role of a coordinating controller for the entire application. The `NSDocumentController` object is provided automatically for document-based applications. You generally do not need to subclass it.

`NSDocumentController` gets the information it needs to manage documents from the document-type metadata specified in the application's information property list (`Info.plist`). This metadata tells the application the extensions, HFS type code, MIME types, icon, and `NSDocument` subclass (if applicable) of each document the application can read and write.

- **An `NSDocument` object manages the model objects of a document.** In MVC terms, an `NSDocument` object is a model-controller, a controller whose management responsibility is shifted towards the model layer. `NSDocument` is an abstract class, and you must create a custom subclass of it that has

knowledge of the document's data, as encapsulated in model objects. The instance of that subclass must be able to read data from a file and from it re-create the document's model objects, and it must be able to save the document's current collection of model objects to a file. Occasionally it may have to maintain more than one internal representation of document data. An `NSDocument` object also owns and manages one or more `NSWindowController` objects.

- **An `NSWindowController` object manages the presentation of the document in a window.** In MVC terms, a `NSWindowController` document is a view-controller: Its concern is managing the window in which document data is displayed. For simple applications you might not need to create a custom subclass of `NSWindowController`; the default instance can handle rudimentary window-management chores while the `NSDocument` object can incorporate knowledge of the view layer. But in most cases you should add the knowledge of the document's view objects to an `NSWindowController` subclass. An instance of this subclass is typically the File's Owner of a document nib file. A multi-window document can have several different `NSWindowController` objects, one for each window.

Xcode provides a project template for Cocoa applications based on the document architecture. When you create a project using this template, you get an `NSDocument` subclass (named `MyDocument`) with stub implementations of the required method overrides. You also get a document nib file with File's Owner set to `MyDocument`.

Further Reading: For a complete description of the document architecture, see *Document-Based Applications Overview*.

Application Scriptability

AppleScript is a scripting language and an interprocess communications technology that many Macintosh power users are familiar with. When compiled and run, an AppleScript script controls an application by sending it commands and may receive data in return.

To be the target of AppleScript commands, an application must be made scriptable. A scriptable application offers its behavior and data in response to AppleScript-generated interprocess messages, called Apple events. Production-quality applications generally should be scriptable. You want to make what your application can do available to as many users as possible, including scripters and users of the Automator application.

Cocoa provides runtime support for scriptable applications. When an application's scriptability information is first needed, the Application Kit loads it and automatically registers Apple event handlers for the supported commands. When the application receives an Apple event for a registered command, the Application Kit instantiates a script command object, initializing it with information obtained from the application's scriptability information. This information enables it to find the scriptable objects in the application on which the command should operate. The Application Kit then executes the command and the scriptable objects perform the work requested. If those objects return a value, the Application Kit packages the value in an Apple event and returns it to the originating script.

For this runtime support to be effective, you must make the application scriptable. This task has several components:

- **You must supply the application's scriptability information.**

The scriptability information specifies the terminology that scripts can use to target the application and its objects. It also includes information about how the application implements support for its terminology. You provide scriptability in one of two formats. The preferred format is an XML-based scripting definition—also called an sdef format because the extension of the file is `.sdef`. The second, older format consists of a pair of property list files: a script suite file and a script terminology file.

Both formats contain the same scriptability information. They define the terminology to be used in scripts and describe the scriptable objects of the application by specifying the classes, commands, constants, and other information that AppleScript and Cocoa need to execute AppleScript commands.

- **You must implement any classes or methods needed to support scriptability.**

Cocoa includes supports for common commands and other terms (such as `get`, `set`, `window`, and `document`) in the Standard suite. (In AppleScript, terms associated with related operations are collected in suites). It also provides the Text suite for scriptable operations on objects of the Cocoa text system. If your application has scriptable operations that can't be performed by the commands defined in the Standard or Text suites, it must define additional command classes. In addition, the application must implement an object-specifier method for each scriptable class; this method describes an object of the class and locates it within a containment hierarchy of the application's objects.

- **You must properly design your application's model and (possibly) its controller objects.**

A Cocoa application that is scriptable should be designed according to the Model-View-Controller (MVC) design pattern, which separates the objects of the application into assigned roles. The model objects of the application are generally the objects that provide scriptability (although controller objects can sometimes be scriptable objects too). The scriptable classes of the application should be compliant with key-value coding (KVC). KVC is a mechanism that allows an object property to be fetched and set indirectly by a key. At runtime, command objects use KVC to find the scriptable objects on which to operate.

You implement KVC-compliance by incorporating the name of the property—which, by default, is also its key—in the instance variable that holds the property or the accessor methods of the class that get and set the property. As you design the classes for your scriptable (and generally model) objects, you also specify in the application's scriptability information the keys for these objects, the scriptable properties of these objects, and the element classes they contain.

For more on the design pattern on which KVC is based, see [“Object Modeling”](#) (page 169).

Important: There is no absolute requirement to base the design of your application's objects on MVC and KVC, but not doing so will make implementing application scriptability much more difficult.

Separately from its support for scriptability, Cocoa automatically handles certain Apple events an application receives from other processes on a system. [“Handling Apple Events”](#) (page 205) describes the Apple events and how Cocoa handles them.

Further Reading: *Cocoa Scripting Guide* explains Cocoa application scriptability in detail.

Core Data

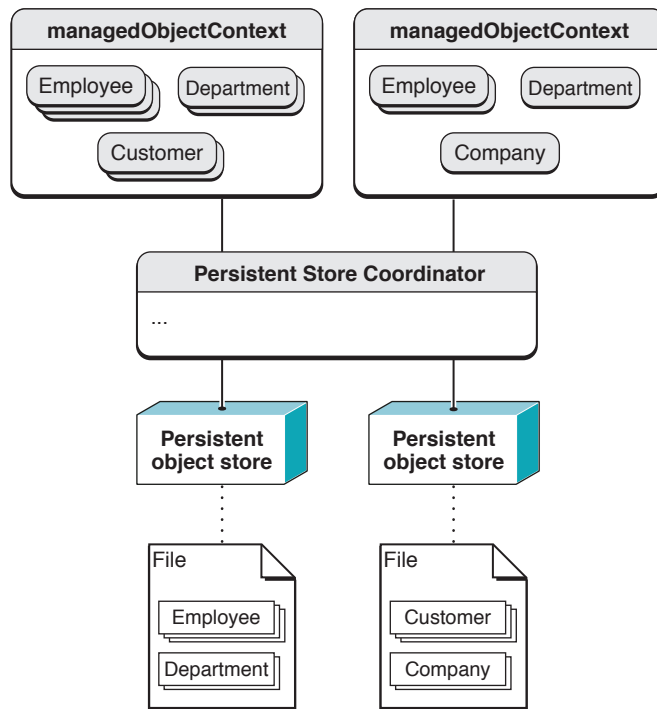
Core Data is a Cocoa framework that provides an infrastructure for managing object graphs, including support for persistent storage to a variety of file formats. Object-graph management includes features such as undo and redo, validation, and ensuring the integrity of object relationships. Object persistence means that Core Data saves model objects to a persistent store and fetches them when required. The persistent store of a Core Data application—that is, the ultimate form in which object data is archived—can range from XML files to SQL databases. Core Data is ideally suited for applications that act as front ends for relational databases, but any Cocoa application can take advantage of its capabilities.

The central concept of Core Data is the managed object. A managed object is simply a model object that is managed by Core Data, but it must be an instance of the `NSManagedObject` class or a subclass of that class. You describe the managed objects of your Core Data application using a schema called a managed object model. (The Xcode application includes a data modeling tool to assist you in creating these schemas.) A managed object model contains descriptions of an application's managed objects (also referred to as entities). Each description specifies the attributes of an entity, its relationships with other entities, and metadata such as the names of the entity and the representing class.

In a running Core Data application, an object known as a managed object context is responsible for a graph of managed objects. All managed objects in the graph must be registered with a managed object context. The context allows an application to add objects to the graph and remove them from it. It also tracks changes made to those objects, and thus can provide undo and redo support. When you're ready to save changes made to managed objects, the managed object context ensures that those objects are in a valid state. When a Core Data application wishes to retrieve data from its external data store, it sends a fetch request—an object that specifies a set of criteria—to a managed object context. The context returns the objects from the store that match the request after automatically registering them.

A managed object context also functions as a gateway to an underlying collection of Core Data objects called the persistence stack. The persistence stack mediates between the objects in your application and external data stores. The stack consists of two different types of objects, persistent stores and persistent store coordinators. Persistent stores are at the bottom of the stack. They map between data in an external store—for example, an XML file—and corresponding objects in a managed object context. They don't interact directly with managed object contexts, however. Above a persistence store in the stack is a persistent store coordinator, which presents a facade to one or more managed object contexts so that multiple persistence stores below it appear as a single aggregate store. Figure B-2 shows the relationships between objects in the Core Data architecture.

Figure B-2 Managed object contexts and the persistence stack



Core Data includes the `NSPersistentDocument` class, a subclass of `NSDocument` that helps to integrate Core Data and the document architecture. A persistent-document object creates its own persistence stack and managed object context, mapping the document to an external data store. An `NSPersistentDocument` object provides default implementations of the `NSDocument` methods for reading and writing document data.

Document Revision History

This table describes the changes to *Cocoa Fundamentals Guide*.

Date	Notes
2008-07-08	Updated for iPhone OS.
2007-10-31	Updated to describe new and enhanced features, including garbage collection, properties, fast enumeration, and development tools. Also added section on multithreading.
2006-12-20	Added a "Cocoa API Conventions" section to "Adding Behavior to Your Cocoa Program."
2006-10-03	Clarified the statement about reinitialization of objects and made minor corrections.
2006-08-07	Corrected several minor errors.
2006-05-23	Added description of represented objects and updated examples and descriptions of singleton objects and object copying.
2006-03-08	A new document that gives developers an orientation to the basic concepts, terminology, architectures, and design patterns of Cocoa.
	The now-retired documents consolidated as chapters in this document are "What Is Cocoa?," "Cocoa Objects," "Adding Behavior to a Cocoa Program," "Cocoa Design Patterns," and "Communicating With Objects." Two new chapters have been added: "The Core Application Architecture on Mac OS X" (page 201) and "Other Cocoa Architectures on Mac OS X" (page 241). In addition, the discussion of the Model-View-Controller design pattern in "Cocoa Design Patterns" (page 141) has been greatly expanded.

REVISION HISTORY

Document Revision History