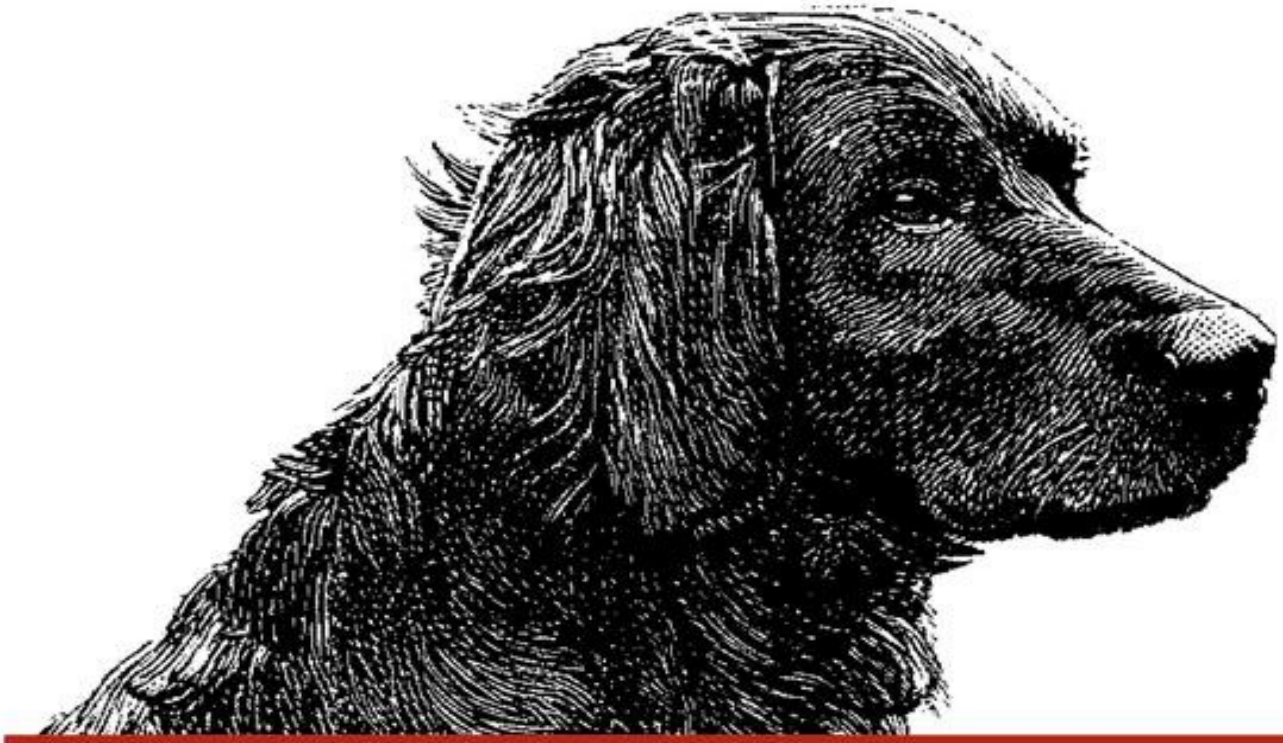
Developer
Connection
Recommended Title



COCOA

IN A NUTSHELL

A Desktop Quick Reference

O'REILLY®

Michael Beam
& James Duncan Davidson

Cocoa in a Nutshell

Michael Beam

James Duncan Davidson

Editor

Chuck Toporek

Preface

It’s practically impossible to know Cocoa inside and out. There was once a discussion between two programmers about Cocoa’s large APIs: one was a veteran Perl programmer, the other a Cocoa programmer. The Perl programmer grumbled about the intimidating and verbose Cocoa APIs, saying there was simply too much to remember. Bemused, the Cocoa programmer retorted: “You don’t *remember* Cocoa; you look it up!”

The point the Cocoa programmer was trying to impress upon the Perl programmer was that understanding object-oriented programming (OOP) concepts and the architecture of the frameworks is more important than remembering the wordy and numerous method and class names in the Cocoa frameworks.

This book is a compact reference that will hopefully grow worn beside your keyboard. Split into two parts, *Cocoa in a Nutshell* first provides an overview of the frameworks that focuses on both common programming tasks and how the parts of the framework interact with one another. The second part of the book is an API quick reference that frees you from having to remember method and class names so you can spend more time hacking code. This book covers the Cocoa frameworks—Foundation and Application Kit (AppKit)—as of Mac OS X 10.2 (Jaguar).

What Is Cocoa?

Cocoa is a complete set of classes and application programming interfaces (APIs) for building Mac OS X applications and tools. With over 240 classes, Cocoa is divided into two essential frameworks: the Foundation framework and the Application Kit.

The Foundation framework provides a fundamental set of tools for representing fundamental data types, accessing operating system services, threading, messaging, and more. The Application Kit provides the functionality to build graphical user interfaces (GUI) for Cocoa applications. It provides access to the standard Aqua interface components ranging from menus, buttons, and text fields —the building blocks of larger interfaces—to complete, prepackaged interfaces for print dialogs, file operation dialogs, and alert dialogs. The Application Kit also provides higher-level functionality to implement multiple document applications, text handling, and graphics.

Classes are not the only constituents of the Cocoa frameworks. Some programming tasks, such as sounding a system beep, are best accomplished with a simple C function. Cocoa includes a number of functions for accomplishing tasks such as manipulating byte orders and drawing simple graphics. Additionally, Cocoa defines a number of custom data types and constants to provide a higher degree of abstraction to many method parameters.

The Cocoa Development Environment

Project Builder and Interface Builder are the two most important applications used in Cocoa development. *Project Builder* is the interactive development environment (IDE) for Mac OS X used to manage and edit source files, libraries, frameworks, and resources. Additionally, it provides an interface to the Objective-C compiler, *gcc*, and the GNU debugger, *gdb*.

Interface Builder is used to create GUIs for Cocoa applications by allowing developers to manipulate UI components (such as windows and buttons) graphically using drag and drop. It provides assistance for laying out components by providing visual cues that conform to Apple's Aqua Human Interface Guidelines. From an inspector panel, the behavior and appearance of these components can be tweaked in almost every way the component supports. Interface Builder provides an intuitive way to connect objects by letting the user drag wires between objects. This way, you set up the initial network of objects in the interface. In addition, you can interface without having to compile a single bit of code.

Interface components are not the only objects that can be manipulated with Interface Builder. You can subclass any Cocoa class and create instances of the subclasses. More importantly, you can give these classes instance variables, known as *outlets* , and methods, called *actions* , and hook them up to user interface components. Interface Builder can then create source files for these subclasses, complete header files, and an implementation file including stubs for the action methods. There is much more to Interface Builder and Project Builder than we can cover in this book, but as you can begin to imagine, the tight integration of these two applications create a compelling application development environment.

Cocoa Design Patterns

Cocoa uses many design patterns. *Design patterns* are descriptions of common object-oriented programming practices. Effective application development requires that you know how and where to use patterns in Cocoa. *Cocoa in a Nutshell* discusses these patterns in the context in which they are used. Here is a brief list of the design patterns you will encounter in the book:

Delegation

: In this pattern, one object, the delegate, acts on behalf of another object. Delegation is used to alter the behavior of an object that takes a delegate. The developer's job is to implement any number of methods that may be invoked in the delegate. Delegation minimizes the need to subclass objects to extend their functionality.

Singleton

: This pattern ensures that only one object instance of a class exists in the system. A singleton method is an object constructor that creates an instance of the class and maintains a reference to that object. Subsequent invocations of the singleton constructor return the existing object, rather than create a new one.

Notification

: Notifications allow decoupling of message senders from multiple message receivers. Cocoa implements this pattern in the notification system used throughout the frameworks. It is discussed in Chapter 2.

Model-View-Control

: The Model-View-Controller (MVC) pattern is used extensively in the Application Kit to separate an application into logically distinct units: a model, which knows how to work with application data, the view, which is responsible for presenting the data to the user, and the controller, which handles interaction between the model and the view. Chapter 3 discusses MVC in more detail.

Target/action

: The target/action pattern decouples user-interface components, such as buttons and menu items, with the objects (the targets) that implement their actions. In this pattern, an activated control sends an action message to its target. Chapter 3 discusses this topic further.

Responder chain

The responder chain pattern is used in the event handling system to give multiple objects a chance to respond to an event. This topic is discussed in Chapter 3.

Key-value coding

Key-value coding provides an interface for accessing an object's properties indirectly by name. Chapter 2 covers key-value coding more thoroughly.

Benefits

These days, application developers expect a lot from their tools, and users expect a lot from any application they use. Any application or application toolkit that neglects these needs is destined for failure. Cocoa comes through grandly by providing the features needed in applications now and in the future, including:

Framework-based development

Cocoa development is based on its frameworks: the Foundation framework and the Application Kit. With framework-based programming, the system takes a central role in the life of an application by calling out to code that you provide. This role allows the frameworks to take care of an application's behind-the-scene details and lets you focus on providing the functionality that makes your application unique.

"For free" features

Cocoa provides a lot of standard application functionality "for free" as part of the frameworks. These features not only include the large number of user-interface components, but larger application subsystems such as the text-handling system and the document-based application architecture. Because Apple has gone to great lengths to provide these features as a part of Cocoa, developers can spend less time doing the repetitive work that is common between all applications, and more time adding unique value to their application.

The development environment

As discussed earlier, Project Builder and Interface Builder provide a development environment that is highly integrated with the Cocoa frameworks. Interface Builder is used to quickly build user interfaces, which means less tedious work for the

developer.

Cocoa's most important benefit is that it lets you develop applications dramatically faster than with other application frameworks.

Languages

Cocoa's native language is Objective-C. The Foundation and Application Kit frameworks are implemented in Objective-C, and using Objective-C provides access to all features of the frameworks. Chapter 1 covers Objective-C in depth.

Objective-C is not, however, the only language through which you can access the Cocoa frameworks. Through the Java Bridge, Apple provides a way to access the Cocoa frameworks using the Java language. The Java Bridge does not provide a complete solution since many of Cocoa's advanced features, such as the distributed objects system, are not available with Java. This book will not discuss Cocoa application development with Java.

Another option for working with Cocoa is AppleScript. AppleScript has traditionally been associated with simple scripting tasks, but with Mac OS X, Apple enabled AppleScript access to the Cocoa frameworks via AppleScript Studio. AppleScript Studio provides hooks into the Cocoa API so scripters can take their existing knowledge of AppleScript, write an application in Project Builder, and use Interface Builder to give their applications an Aqua interface—all without having to learn Objective-C. This exposes Cocoa to a completely new base of Macintosh developers, who know enough AppleScript to build simple task-driven applications for solving common problems. For more information about AppleScript Studio, see http://www.apple.com/applescript/studio.

How This Book Is Organized

This book is split into two parts: the overview of Cocoa familiarizes developers with Cocoa's structure, and the API quick reference contains method name listings and brief descriptions for all Foundation and Application Kit framework classes.

Part I is divided into the following eight chapters:

Chapter 1, Objective-C

This chapter introduces the use of Objective-C language. Many object-oriented concepts you may be familiar with from other languages are discussed in the context of Objective-C, which lets you leverage your previous knowledge.

Chapter 2, Foundation

This chapter discusses the Foundation framework classes that all programs require for common programming tasks such as data handling, process control, run loop management, and interapplication communication.

Chapter 3, The Application Kit

This chapter introduces the Application Kit and details larger abstractions of the Application Kit, such as how events are handled with responder chains, the document-based application architecture, and other design patterns that are important in Cocoa development.

Chapter 4, Drawing and Imaging

This chapter discusses Cocoa's two-dimensional (2D) graphics capabilities available in the Application Kit.

Chapter 5, Text Handling

This chapter details the architecture of Cocoa's advanced text-handling system, which provides a rich level of text-handling functionality for all Cocoa developers.

Chapter 6, Networking

This chapter summarizes networking technologies, such as Rendezvous and URL services, that are accessible from a Cocoa application.

Chapter 7, Interapplication Communication

This chapter discusses interapplication communication techniques, including distributed objects, pipes, and distributed notifications.

Chapter 8, Other Frameworks

This chapter provides information about the many Objective-C frameworks that can be used in conjunction with Cocoa. These frameworks include those that are part of Mac OS X, such as AddressBook and DiscRecording, as well as frameworks supplied by third-party developers.

Part II contains Foundation and AppKit framework references and, as such, makes up the bulk of the book. First, there's an explanation of the organization of chapters in Part II and how class information is referenced. The rest of the section is divided into eight chapters and a method index. Each chapter focuses on a different part of the Cocoa API.

Chapter 9, Foundation Types and Constants

This chapter lists the data types and constants defined by the Foundation framework.

Chapter 10, Foundation Functions

This chapter lists the functions defined by the Foundation framework.

Chapter 11, Application Kit Types and Constants

This chapter lists the data types and constants defined by the Application Kit.

Chapter 12, Application Kit Functions

This chapter lists the functions defined by the Application Kit.

Chapter 13, Foundation Classes

This chapter contains the API quick-reference Foundation framework classes.

Chapter 14, Foundation Protocols

This smaller chapter covers the handful of protocols declared as part of the Foundation framework.

Chapter 15, Application Kit Classes

This chapter provides the API quick reference for Application Kit classes.

Chapter 16, Application Kit Protocols

This chapter provides reference to the protocols defined and used in the AppKit.

Chapter 17, Method Index

This index contains an alphabetical listing of every method in the Foundation framework and Application Kit. Each method name in the index has a list of classes that implement that method.

Unlike the rest of the book's sections, there is but one short appendix in Part III. Regardless of your experience level as a Mac developer, this section contains valuable resources for Cocoa programmers, including details on how you can partner with Apple to market your application.

Appendix A

This appendix lists vital resources for Cocoa developers, including Apple developer documentation, web sites, mailing lists, books, and details on how to partner with Apple to gain exposure for your applications.

Conventions Used in This Book

This book uses the following typographical conventions:

Italic

Used to indicate new terms, URLs, filenames, file extensions, directories, commands, options, and program names, and to highlight comments in examples. For example, a filesystem path will appear as */Applications/Utilities*.

`Constant width`

Used to show the contents of files or output from commands.

`Constant-width bold`

Used in examples and tables to show commands or other text that the user should type literally.

`Constant-width italic`

Used in examples and tables to show text that should be replaced with user-supplied values, and also to highlight comments in code.

Menus/navigation

Menus and their options are referred to in the text as File → Open, Edit → Copy, etc. Arrows will also signify a navigation path in window options—for example, System Preferences → Screen Effects → Activation means that you would launch System Preferences, click on the icon for the Screen Effects preferences panel, and select the Activation pane within that panel.

Pathnames

Pathnames show the location of a file or application in the filesystem. Directories (or folders for Mac and Windows users) are separated by a forward slash. For example, if you see something like, "...launch the Terminal application (*/Applications/Utilities*)" in the text, you'll know that the Terminal application can be found in the Utilities subfolder of the Applications folder.

`%`, `#`

The percent sign (`%`) shows the user prompt for the default *tcsh* shell; the hash mark (`#`) is the prompt for the root user.

Menu symbols

When looking at the menus for any application, you will see symbols associated with keyboard shortcuts for a particular command. For example, to open a document in Microsoft Word, go to the File menu and select Open (File → Open), or issue the keyboard shortcut,

⌘

-O.

Figure P-1 shows the symbols used in various menus to denote a shortcut.

Figure 1. Keyboard accelerators for issuing commands

You'll rarely see the Control symbol used as a menu command option; it's more often used in association with mouse clicks or for working with the *tcsh* shell.

Tip ✎

Indicates a tip, suggestion, or general note.

Warning ☠

Indicates a warning or caution.

How the Quick Reference Was Generated

You'd have to be a madman to write this book's quick reference by hand. Madmen we are not, so following the example of David Flanagan, author of O'Reilly's *Java in a Nutshell*, Mike wrote a program that would take care of most of the tedious work.

The idea is to attack the problem in two stages. In the first stage, the code enumerates each header file of each Framework that is to be ripped (Foundation and AppKit) and runs each line of each header through a parser. This parser would look for key elements that identify parts of the header, such as `@interface`, `+` for class methods, `-` for instance methods, and so forth. Every discovered element was assembled into a cross-linked hierarchy of framework names, class names, or method names. When all headers had been processed, the hierarchy was output into a property list file, which, at the end of the day, weighed in at just over 41,500 lines of text!

Stage two involved reading the contents of this file and running it through several formatting routines that output the XML-formatted text required by the O'Reilly production team.

Each class has a little class hierarchy figure. These figures were autogenerated by drawing into a view (using `NSBezierPath`) and saving the PDF representation of the view contents to a file. The input data for the program that did all of the drawing was the same property list used to create the API quick reference entries.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O’Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/cocoaian

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O’Reilly Network, see the O’Reilly web site at:

http://www.oreilly.com

Acknowledgments

The authors would like to acknowledge the many people who helped make this book possible.

From Mike

Writing this book has been quite an experience, and it was made possible only by the efforts and support of the people I worked with. My editor, Chuck Toporek, put in a lot of time on this book and kept this first-time author on course and in the right frame of mind with his kind words of encouragement and level-headed advice. He has become a good friend over the past year that we've worked together on this project.

I am grateful to Duncan for his efforts in helping me shape up the book and for contributing the material on Objective-C. Duncan is quite a person to work with, and I look forward to working with him on this book in the future. Any success of this book is due in no small part to both Chuck and Duncan. These two make a great team, and I am fortunate to have the opportunity to work with them.

Thanks to the tech reviewers: Scott Anguish, Sherm Pendley, and the engineers and technical writers at Apple who were kind enough to take time out of their busy lives to review the book. Special thanks go to Malcolm Crawford for going above and beyond the call of duty by providing in-depth comments and suggestions and working closely with us to give the book its final polish. His upbeat attitude and British charm helped us all bring this book to completion.

Derrick Story at the O'Reilly Network took an amazing chance with me by letting me write about Cocoa for http://www.macdevcenter.com, which gave me the opportunity to get my foot in the door when I was least expecting it. Why he did this baffles me to this day, but I am grateful for it and for his encouragement over the past two years.

Ryan Dionne introduced me to Macs when we were freshman at UT Austin, and he quickly changed my attitude about them (I was a switcher before switching was fashionable). Shortly after that, John Keto of the University of Texas, my teacher and employer, was tricked, by some of the grad students I worked with, into believing that I was some sort of Linux and C guru; let's just say that I quickly

became one! I suppose that if either of these things hadn't happened, you wouldn't be reading this acknowledgment. Life's funny sometimes.

All remaining thanks, and all that I am, go to my family and my friends: Mom and Dad, for the love, encouragement, and support during the whole process; my sisters Kristin and Jennifer; and my future parents-in-law, Bill and Lauren, for their love and support; Ryan, Paige, and Tommy for putting up with me and my antisocial behaviors during the past year, and for always having an eye on me and knowing when I needed to get some lunch. As always, my love and appreciation to my fiancée, Heather, (until July 2003!) for being incredibly patient, supportive, and caring during the past year.

From Duncan

I'd like to thank Mike and Chuck for letting me contribute Chapter 1 to the book. They were both very patient and attentive to all of the feedback I contributed to the rest of the book, even when they must have become annoyed by all my suggestions. Chuck, you're a great editor and you've helped me develop as an author, a skill that I never thought I'd have. Mike, I'm honored to have helped you with this book, and I look forward to working with you on it again in the future.

mmalcolm Crawford provided an invaluable service by checking the Objective-C chapter, as well as the rest of the book, in detail, and he really helped shape it into the what you see today. His dinner table discussions, and plenty of red ink stemming from many years of experience, have illuminated several areas of Cocoa and Objective-C for me. This book would not be the book it is without his valuable help.

Finally, thanks to my family and friends who put up with me disappearing during the crunch time leading up to the production of this book. You guys know who you are.

Part I. Introducing Cocoa

This part of the book provides a series of chapters that provide a general overview of Cocoa, helping you to quickly come up to speed. The chapters in this part of the book include:

Chapter 1, Objective-C

Chapter 2, Foundation

Chapter 3, The Application Kit

Chapter 4, Drawing and Imaging

Chapter 5, Text Handling

Chapter 6, Networking

Chapter 7, Interapplication Communication

Chapter 8, Other Frameworks

Chapter 1. Objective-C

Objective-C is a highly dynamic, message-based *object-oriented* language. Consisting of a small number of additions to ANSI C, Objective-C is characterized by its deferral of many decisions until runtime, supporting its key features of dynamic dispatch, dynamic typing, and dynamic loading. These features support many of the *design patterns* Cocoa uses, including delegation, notification, and Model-View-Controller (MVC). Because it is an extension of C, existing C code and libraries, including those based on C++,[1] can work with Cocoa-based applications without losing any of the effort that went into their original development.

This chapter is an overview of Objective-C's most frequently used features. If you need more detail about these features or want to see the full language specification, read through Apple's document, *The Objective-C Programming Language*, which is installed as part of the Developer Tools in */Developer/Documentation/Cocoa/ObjectiveC*.

Objects

The base unit of activity in all object-oriented languages is the *object*— an entity that associates data with operations that can be performed on that data. Objective-C provides a distinct data type, `id`, defined as a pointer to an object's data that allows you to work with objects. An object may be declared in code as follows:

```
id anObject;
```

For all object-oriented constructs of Objective-C, including method return values, `id` replaces the default C `int` as the default return data type.

Dynamic Typing

The `id` type is completely nonrestrictive. It says very little about an object, indicating only that it is an entity in the system that can respond to messages and be queried for its behavior. This type of behavior, known as *dynamic typing* , allows the system to find the class to which the object belongs and resolve messages into method calls.

Static Typing

Objective-C also supports *static typing* , in which you declare a variable using a pointer to its class type instead of `id`, for example:

```
NSObject *object;
```

This declaration will turn on some degree of compile time checking to generate warnings when a type mismatch is made, as well as when you use methods not implemented by a class. Static typing can also clarify your intentions to other developers who have access to your source code. However, unlike other languages' use of the term, static typing in Objective-C is used only at compile time. At runtime, all objects are treated as type `id` to preserve dynamism in the system.

> **Tip** ✎
>
> There are no class-cast exceptions like those present in more strongly typed languages, such as Java. If a variable declared as a `Dog` turns out to be a `Cat`, but responds to the messages called on it at runtime, then the runtime won't complain.

---

[1] For more information on using C++ with Objective-C, see the Objective-C++ documentation contained in */Developer/Documentation/ReleaseNotes/Objective-C++.html*.

Messaging

Objects in Objective-C are largely autonomous, self-contained, opaque entities within the scope of a program. They are not passive containers for state behavior, nor data and a collection of functions that can be applied to that data. The Objective-C language reinforces this concept by allowing any *message*— a request to perform a particular action—to be passed to any object. The object is then expected to respond at runtime with appropriate behavior. In object-oriented terminology, this is called *dynamic binding* .

When an object receives a message at runtime, it can do one of three things:

- Perform the functionality requested, if it knows how.
- Forward the message to some other object that might know how to perform the action.
- Emit a warning (usually stopping program execution), stating that it doesn't know how to respond to the message.

A key feature here is that an object can forward messages that it doesn't know how to deal with to other objects. This feature is one of the significant differences between Objective-C and other object-oriented languages such as Java and C++.

Dynamic binding, as implemented in Objective-C, is different than the *late binding* provided by Java and C++. While the late binding provided by those languages does provide flexibility, it comes with strict compile-time constraints and is enforced at link time. In Objective-C, binding is performed as messages are resolved to methods and is free from constraints until that time.

Structure of a Message

Message expressions in Objective-C are enclosed in square brackets.[2]

The expression consists of the following parts: the object to which the message is sent (the receiver), the message name, and optionally any arguments. For example, the following message can be verbalized as "send a `play` message to the object identified by the `iPod` variable":

```
[iPod play];
```