Creating iPhone Apps with Cocoa Touch

mini

the missing manual®

O'REILLY®

Craig Hockenberry

Creating iPhone Apps with Cocoa Touch: The Mini Missing Manual
by Craig Hockenberry

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles: safari.oreilly.com. For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

May 2010: First Edition.

ISBN: 9781449388409

Table of Contents

Introduction

Since the iTunes App Store's launch in July 2008, developers have submitted over 100,000 iPhone applications to the store, resulting in over 3 billion downloads and climbing. Who wouldn't want a piece of that action?

Before the app store was launched, iPhone app development was limited to the engineers at Apple headquarters in Cupertino, California. Now that Apple has released the developer tools to anyone who wants to download them, thousands of developers have discovered how easy and fun it is to write software for the iPhone.

If you've done any programming in C or a related language like C++, Java, PHP, or Perl, you can learn to write iPhone apps in a snap. Objective-C will feel wonderfully familiar, since its entire infrastructure is based on standard C.

In *iPhone App Development: The Mini Missing Manual*, you'll create your first iPhone app right from the get-go, and get up to speed fast on all of your tools—Cocoa Touch, Interface Builder, Xcode, and Objective-C.

Tip: There's more to producing your own app than just writing the code. This eBook is ideal if the Cocoa Touch programming interface is all you want help with. For the full story on creating and selling a successful app–designing, programming, troubleshooting, submitting, and marketing–check out *iPhone App Development: The Missing Manual*. It covers everything in this Mini Manual, plus the entire lifecycle of an iPhone app.

Building Your First iPhone App

You have an idea that will lead to fame and fortune on the iTunes App Store. You decide to write an iPhone app. The first and most important task is for you to become comfortable with the tools used to build your products. A Chinese proverb says, "the journey is the reward," and this chapter is all about the journey. In the upcoming pages, you'll experience the entire application development process, start to finish. You'll learn how to set up the software you need, and try your hand at building an app.

But what app? If you do a quick search of the App Store, you'll find no shortage of flashlights. For many aspiring developers, this simple application is a rite of passage, so now's your chance to join this illustrious crowd. Once you see how easy it is to create your own app, you'll wonder why people pay 99¢ for them on iTunes!

Getting the Tools

You can't build anything, including an iPhone app, without tools. Luckily, you can find everything you need on your Mac, or download it for free. Specifically, you need to download and install Xcode development software and the iPhone Software Development Kit (SDK) on your Mac. (And if you don't have a Mac, see the box on page 2.)

Both the Mac and iPhone benefit from a rich set of technologies that have stood the test of time. The iPhone SDK is built upon the infrastructure created by NeXT in the 1980s. This company, founded by Steve Jobs, created a revolutionary object-oriented operating system called NeXTSTEP. This influential system has evolved into the OS X operating system in use today. As you learn more about the iPhone, you'll see that it has much in common with the Mac.

Note: You see the NeXT legacy whenever you encounter an object with the prefix "NS". Those initials stand for NeXTSTEP.

UP TO SPEED

Get a Mac

If you're going to create iPhone applications, you're going to do it on a Macintosh. Apple's development tools don't run on Windows or any other operating system. Just as you can't run Microsoft Visual Studio on a Mac, you need a Mac to run the tools used to build your iPhone app. They rely on features of the underlying system software.

If you don't have a Mac, here are some hints to help you make the right purchase:

- **Buy a used machine.** If you're on a shoestring budget, check out eBay or craigslist. Someone else's old hardware will be perfectly fine for iPhone development. The apps you're going to create are small and don't need a lot of processor power to build and test. The only caveat when buying older hardware is to make sure the Mac has an Intel processor. The development tools don't work with older PowerPC processors.

—Continued

- **Add a Mac mini.** Buying a new Mac mini is a great option if you already have a display, keyboard, and other peripherals. You can save quite a bit of money by just buying a new CPU and repurposing the devices you already own. If you're a software developer, you probably have this stuff already sitting around in a closet. And if you're developing for multiple platforms, it's handy to put the Mac mini behind a KVM switch so you can quickly shift between machines.
- **Go ahead and splurge.** Apple makes some very sexy hardware. In particular, the new laptops are hard to resist. If you're looking for excuses to justify the purchase, here's some help:

Macs now use an Intel processor, which means you can run Windows or any other x86-based operating system on your new machine. You can boot into any operating system using Apple's free Boot Camp utility. Or you may find it easier to install third-party software like VMware Fusion and to run other operating systems on a virtual machine within Mac OS X.

Virtual machines are particularly handy when you need to see how your iPhone product website appears in Internet Explorer. Just launch the virtual machine, open the browser in Windows, and load the test URL.

Finally, think of all the money you're saving on development tools. If you're used to spending thousands of dollars on Visual Studio and MSDN, it will come as a pleasant surprise to know that all of Apple's developer tools are free. Spend your dollars on the hardware instead of the tools, and you'll come out ahead.

Installing Xcode

Once you and your Mac are ready to go, it's time to load your hard drive with lots of new software. Apple supplies the Xcode development tools free of charge, but doesn't install them on every Mac, since most consumers will never use them.

Luckily, you can find the Xcode tools right on your Snow Leopard installation disk. To run Xcode, Apple recommends you have an Intel-based Mac running Leopard or Snow Leopard.

Note: You can install the iPhone SDK and other development tools on Leopard, but the Snow Leopard tools reflect significant improvements over the previous version. Working in the newest version of Mac OS X assures you the latest and greatest features.

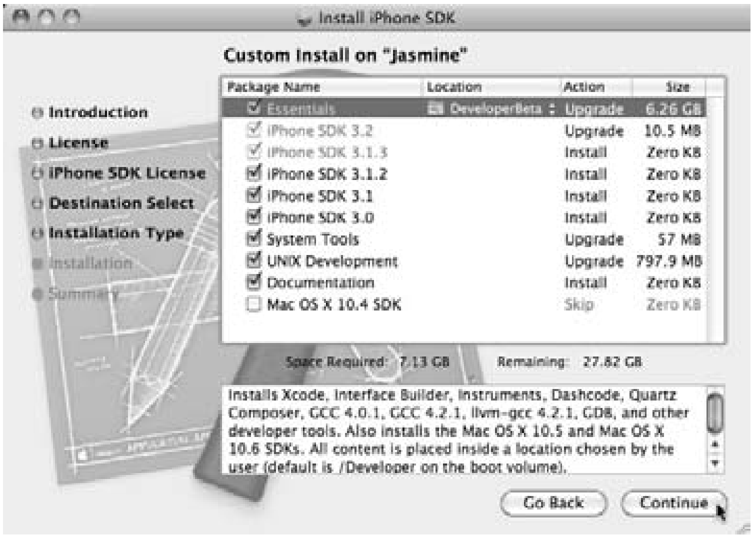
The following steps explain how to get the software onto your hard drive where you can use it:

1. **Pop the installation DVD into your Mac and double-click its icon. In the Optional Installs folder, double-click the Xcode.mpkg file.**

   When you double-click that file, the Xcode installation process begins.

2. **On the introductory screen, click Continue. When the license agreement screen appears, click Continue and then click Agree.**

   The license agreement is the same legalese you agree to whenever you install software. Read it if you're into such things. When you're done, the next screen lets you choose what you want to install, as shown in Figure 1-1.



Custom Install on "Jasmine"

Package Name	Location	Action	Size
✓ Essentials	DeveloperBeta	Upgrade	6.26 GB
✓ iPhone SDK 3.2		Upgrade	10.5 MB
✓ iPhone SDK 3.1.3		Install	Zero KB
✓ iPhone SDK 3.1.2		Install	Zero KB
✓ iPhone SDK 3.1		Install	Zero KB
✓ iPhone SDK 3.0		Install	Zero KB
✓ System Tools		Upgrade	57 MB
✓ UNIX Development		Upgrade	797.9 MB
✓ Documentation		Install	Zero KB
☐ Mac OS X 10.4 SDK		Skip	Zero KB

Space Required: 7.13 GB Remaining: 27.82 GB

Installs Xcode, Interface Builder, Instruments, Dashcode, Quartz Composer, GCC 4.0.1, GCC 4.2.1, llvm-gcc 4.2.1, GDB, and other developer tools. Also installs the Mac OS X 10.5 and Mac OS X 10.6 SDKs. All content is placed inside a location chosen by the user (default is /Developer on the boot volume).

Figure 1-1: *The Xcode installer starts out with the first four checkboxes selected for you. Leave them that way. You can click each package name to see what's being installed: Besides the integrated development environment (IDE) that you'd expect, you'll also find tools for monitoring performance and plenty of documentation.*

3. **On the Custom Install screen, make sure the first four checkboxes are turned on, and then click Continue.**

   The installer copies all of the Xcode from the DVD onto your hard drive. This process takes a few minutes.

4. **The final screen prompts you to select an install location. Make sure that it's on the same disk where all your other applications are stored. Click Install to start the process.**

   Depending on your Mac's speed, this process can take a few hours. Get away from the computer and get some fresh air for once.

After everything is safely on your hard drive, you see this message: "The installation was successful."

5. **Click Close to quit. You can safely eject the DVD at this point.**

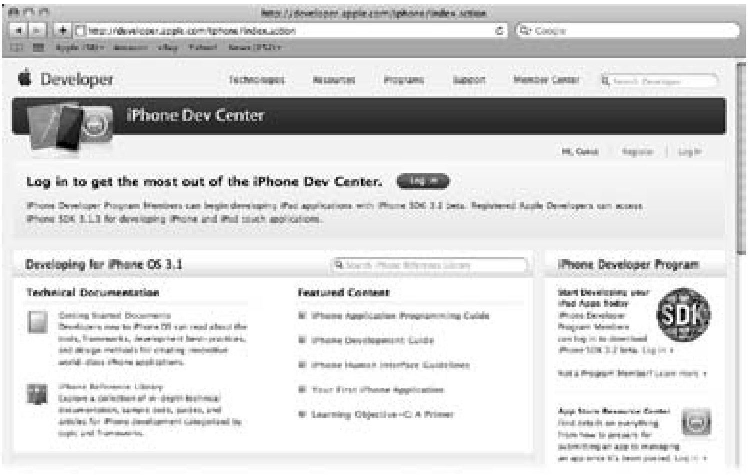
After the installation is complete, go to the Hard Drive→ Developer→ Applications folder on your hard drive, and check out your new tools. This folder contains the applications and utilities you use to develop both Mac and iPhone applications: The ones you'll use the most are Xcode and Interface Builder. The parent Developer folder also has all of the accompanying developer frameworks, libraries, and documentation.

The Xcode installation doesn't include one thing—the iPhone SDK that's required to develop apps for your phone. For that, go on to the next section.

Tip: Now that you have your tools, maintain them. Apple regularly updates Xcode, so the version on your Snow Leopard DVD will eventually become outdated. When major changes occur, Apple will send an email reminding you to upgrade by visiting the iPhone Dev Center, as described in the next section.

Getting the iPhone SDK

You have to join the iPhone Developer Program before Apple lets you get your hands on the iPhone SDK. Your free membership gives you access to the tools, documentation, and developer forums via the iPhone Dev Center (Figure 1-2).


Developer

Technologies Resources Programs Support Member Center

iPhone Dev Center

Hi, Guest | Register | Log In

Log in to get the most out of the iPhone Dev Center.

iPhone Developer Program Members can begin developing iPad applications with iPhone SDK 3.2 beta. Registered Apple Developers can access iPhone SDK 3.1.3 for developing iPhone and iPod touch applications.

Developing for iPhone OS 3.1

Technical Documentation

Getting Started Documents
Developers new to iPhone OS can read about the tools, frameworks, development best-practices, and design methods for creating innovative world-class iPhone applications.

iPhone Reference Library
Explore a collection of in-depth technical documentation, sample code, guides, and articles for iPhone development categorized by topic and frameworks.

Featured Content

- iPhone Application Programming Guide
- iPhone Development Guide
- iPhone Human Interface Guidelines
- Your First iPhone Application
- Learning Objective-C: A Primer

iPhone Developer Program

Start Developing your iPad Apps Today
iPhone Developer Program Members can log in to download iPhone SDK 3.2 beta. Log in ›

Not a Program Member? Learn more ›

App Store Resource Center
Find details on everything from how to prepare for submitting an app to managing an app once it's been posted. Log in ›

Figure 1-2: *The iPhone Dev Center is your first and best resource as an iPhone developer. You'll use this site to download and update your iPhone SDK, find sample code and documentation, connect with other iPhone developers, and to prepare your product for sale on iTunes.*

6. **To sign up for an ADC membership, point your web browser to *http://developer.apple.com/iphone*. Click the Register link in the upper-right corner.**

   You access the iPhone Dev Center using an Apple ID. If you have an iTunes account or have made a purchase from the Apple Store, you already have one set up. Go ahead and use it when you create your developer account and skip to step 8.

Note: If you've been using your Apple ID for personal stuff like iTunes and a MobileMe family photo gallery, you may want to create a new Apple ID for your developer account. Having a separate Apple ID used solely for business purposes can help you avoid accounting and reporting issues.

7. **If you're setting up a new Apple ID, type your name, contact information, and security questions for password retrieval.**
8. **Turn on the checkbox to accept the licensing agreement and click Continue.**

   In a few minutes, Apple will send you an email to verify the account.

9. **Click the Email Verification link, and enter the code contained in the message to complete the account setup.**

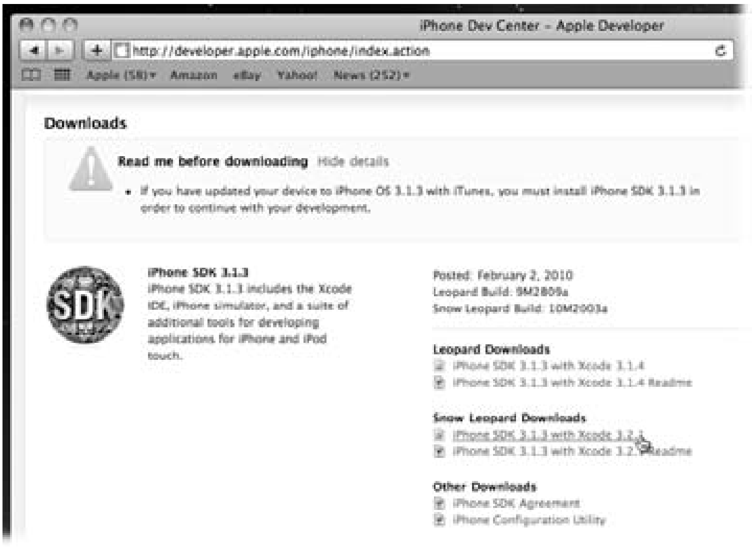
Once you set up your account and log in, you see a lot of new content available from the iPhone Dev Center. You have access to great resources like the Getting Started Videos, Coding How-To's, and Sample Code. Right now, turn your attention to the download for the iPhone SDK.

1. **Click the Downloads link, and you see a selection of links at the bottom of the page, as shown in Figure 1-3.**

   As new versions of the iPhone SDK are released, these links will be updated. Pick the most recent release that matches your version of Mac OS X. At the time of this writing, it's "iPhone SDK 3.1.3 with Xcode 3.2.1".

The iPhone SDK is a large download: Its size can range from several hundred megabytes to over 2 GB. Be patient as it downloads from your web browser, is verified, and mounted. It's going to take a while.

Once it's finished, you have a .dmg disk image in your Downloads folder and a new iPhone SDK disk on your desktop, as shown in Figure 1-4.


iPhone Dev Center – Apple Developer
http://developer.apple.com/iphone/index.action
Apple (58)▾
Amazon
eBay
Yahoo!
News (252)▾
SDK

Downloads

Read me before downloading Hide details

- If you have updated your device to iPhone OS 3.1.3 with iTunes, you must install iPhone SDK 3.1.3 in order to continue with your development.

iPhone SDK 3.1.3
iPhone SDK 3.1.3 includes the Xcode IDE, iPhone simulator, and a suite of additional tools for developing applications for iPhone and iPod touch.

Posted: February 2, 2010
Leopard Build: 9M2809a
Snow Leopard Build: 10M2003a

Leopard Downloads
iPhone SDK 3.1.3 with Xcode 3.1.4
iPhone SDK 3.1.3 with Xcode 3.1.4 Readme

Snow Leopard Downloads
iPhone SDK 3.1.3 with Xcode 3.2.1
iPhone SDK 3.1.3 with Xcode 3.2.1 Readme

Other Downloads
iPhone SDK Agreement
iPhone Configuration Utility

Figure 1-3: *You find the links to download the iPhone SDK toward the bottom of the iPhone Dev Center page. The links in this picture are for version 3.1.3, but these will change as Apple updates the SDK. You can click the Read Me links to see what's new in the release.*

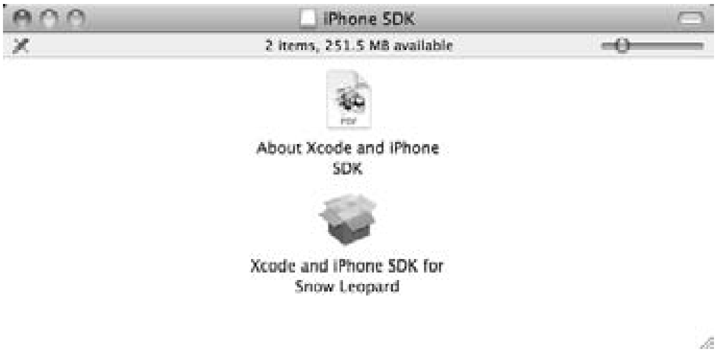

iPhone SDK
2 items, 251.5 MB available
About Xcode and iPhone SDK
Xcode and iPhone SDK for Snow Leopard

Figure 1-4: *After a successful download, this disk image appears on your desktop. Its name, which will vary with each new release, will begin with "iphone_sdk" followed by the version number and the ".dmg"extension. Launch the installer by clicking the box icon. The PDF file contains information about the release that you can read while the installation takes place.*

Once you have the iPhone SDK disk image, you can begin the installation:

1. **Double-click the "iPhone SDK" file to start the installation process. It's the brown and gold box icon.**
2. **Click Continue on the welcome and license agreement screens. Click Agree to accept the license.**
3. **On the Install screen, click Continue to install the standard packages, and then click Install to start the installation process.**

4. **If required, enter your password so system files can be modified. It's also a good idea to quit iTunes at this point to avoid a dialog box that pauses the install.**

   Depending on your Mac's speed and the size of the download, the installation process can take anywhere from half an hour to several hours. When the installation is finished, you'll see a green checkmark and can click Close to finish. You can eject the iPhone SDK disk, but make sure to keep the .dmg file around as a backup.

Note: As with Xcode, Apple updates the iPhone SDK regularly. You'll need to return to this iPhone Dev Center periodically to install the latest version of the SDK. Apple typically releases a new version in conjunction with a new iPhone firmware release.

What Lies Ahead for the SDK?

The iPhone SDK is constantly evolving as bugs are fixed and new features are added. You'll want to update your development environment to keep up with the latest changes. Apple updates the iPhone SDK in two different ways. The first, and simplest, is a maintenance release. These releases just fix bugs in the firmware and don't introduce any new features. In most cases, you won't need to make any changes to your application.

Apple provides maintenance releases of the SDK to developers on the same day that it makes the firmware available to customers. These releases have a three-part version number like 2.2.1 and 3.1.3. As soon as you install the new firmware on your device, you need to update the iPhone SDK so you can install and debug your applications from Xcode. If you don't, you'll see warnings that the tools don't work with the device's firmware version.

When Apple makes more substantial firmware changes that will affect developer software, either by adding new features or changing existing ones, it posts a beta version of the iPhone SDK on the iPhone Dev Center. Only developers who have paid to join the iPhone Developer Program have access to these advance releases. These betas are for major releases, such as 3.0 or 4.0, or revisions like 3.1. Apple typically starts the beta release cycle three or four months prior to a general public release. Once the cycle starts, it puts out a new SDK (called Beta 1, Beta 2, and so on) every couple of weeks. These beta releases usually also include a new version of Xcode with improvements and support for the new iPhone OS, along with new firmware.

With early access to the new SDK, you can build and run your application with the new iPhone firmware. If you've been careful to use only documented features and APIs, you shouldn't have many issues to deal with: Apple's engineers are very good at maintaining compatibility with published interfaces. You may see deprecation warnings as you compile, but those are usually simple to fix. It's more likely that you'll spend the beta test period learning about new features and testing them out in your application.

There are a couple caveats to keep in mind when installing a beta version of the iPhone SDK. First, you can't use the beta tools to submit an application to the App Store. Luckily, you can install multiple versions of Xcode on your hard drive. To install the tools in a separate location, follow these steps:

1. **Quit the iPhone Simulator if it's running.**

   If you skip this step, the installation process will hang indefinitely, and you'll need to quit the Installer and start over.

2. **Double-click the "iPhone SDK" icon in the disk image to start the installation process. Agree to the licenses and choose a destination hard drive.**

   You see a list of packages to install. In the second column, Developer is set as the Location. You need to change the location for the beta release.

3. **Click Developer and then select Other from the pop-up menu. See Figure 1-5.**

   A dialog box opens for you to select a folder.

4. **Navigate to the root of your hard drive by selecting its name from the list of DEVICES. Then click the New Folder button and type *DeveloperBeta*. Click Create to create the folder.**
5. **Select Choose to use the** *DeveloperBeta* **folder for the installation.**

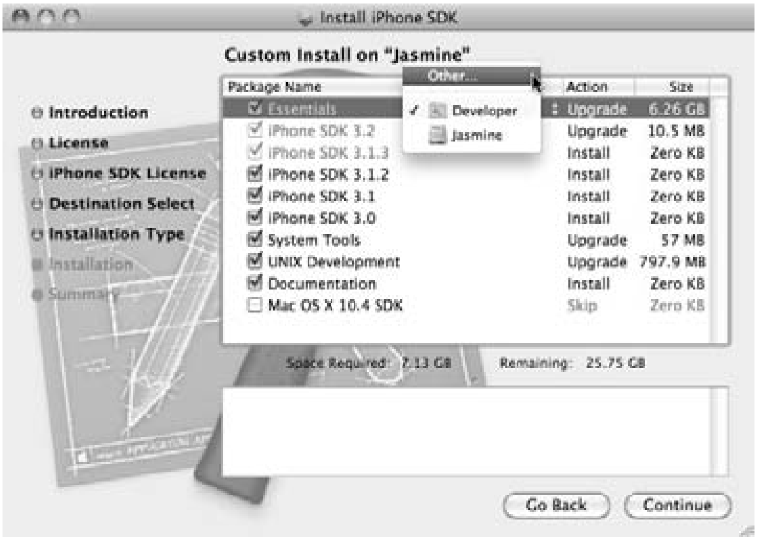
   After you return to the main installation window, you'll see *DeveloperBeta* as the Location.

6. **To use the beta, launch Xcode and other tools from the new Hard Drive→DeveloperBeta→Applications folder.**

Now for the second caveat: the beta release is Apple Confidential Information and is covered by a Non-Disclosure Agreement (NDA). These big legal words mean that you can't talk about it in public. You can discuss the new SDK only on the Apple Developer Forums (*http://devforums.apple.com*). You can connect with other developers who are doing the same thing you are: learning about a new release by asking questions and sharing discoveries. Apple engineers also contribute to the discussion.

The NDA also means that you won't find any books or other media to help you understand the changes. The only information about the beta release comes from Apple itself and is posted on the iPhone Dev Center. Typically there's a "What's New" document, release notes, and a list of API differences. Read each of these documents fully: it's a great way to pass the time when you're waiting for several gigabytes of SDK to download!

Another source of information is Apple's annual developer conference, WWDC. Beta releases often coincide with this weeklong conference so everyone can discuss new features in detail. The conference takes place during the summer in San Francisco: it's a great opportunity to meet your fellow developers and learn lots of new things.



Custom Install on "Jasmine"

Package Name	Action	Size
✓ Essentials	Upgrade	6.26 GB
✓ iPhone SDK 3.2	Upgrade	10.5 MB
✓ iPhone SDK 3.1.3	Install	Zero KB
✓ iPhone SDK 3.1.2	Install	Zero KB
✓ iPhone SDK 3.1	Install	Zero KB
✓ iPhone SDK 3.0	Install	Zero KB
✓ System Tools	Upgrade	57 MB
✓ UNIX Development	Upgrade	797.9 MB
✓ Documentation	Install	Zero KB
☐ Mac OS X 10.4 SDK	Skip	Zero KB

Space Required: 7.13 GB Remaining: 25.75 GB

Figure 1-5: *You can choose a custom installation location for the iPhone SDK. Since you can't use beta releases of the iPhone SDK to build your application for the App Store, you'll need to keep two versions of the tools on your hard drive. During the install process, click the Developer folder icon and select Other to choose the location for the beta version.*

Exploring Your New Tools

Your Mac is now set up to create iPhone applications, so you're ready to start making your first one. The best part is that you're not going to write any code. How can you develop without writing code? It's possible with the timesaving power of Xcode templates and Interface Builder.

If you're an experienced developer, this way of working can present a challenge. If you're used to working in Visual Studio, Eclipse, or some other environment, your first encounter with Xcode can be a bit daunting. Besides working on a new operating system, you're also going to be dealing with new project layouts, keyboard shortcuts, and preferences. Don't worry, all of the tools you're used to having are still there, it's just a matter of time before you become comfortable using Xcode's version of them.

In this section, you'll go through all of the phases of creating an iPhone app, from creating a project file with Xcode to running it in the iPhone Simulator. You'll also take a peek at the Interface Builder application that lets you modify the user interface.

Every Flashlight Needs a Parts List

The first phase of creating an iPhone app is setting up a Project file. This file keeps track of the information Xcode uses to build your application. It's where you manage your source code, user interfaces, frameworks, and libraries. Think of it as a parts list for your application.

1. **In your Hard Drive→Developer→Applications folder, double-click the Xcode icon to start the application. (It's at the bottom of the list.)**

   The tricky part is that Xcode isn't in your normal Applications folder. The installer puts it in the *Developer*→Applications folder. To make it easier to return to Xcode later, store the icon in your Dock.

2. **In the Dock, Control-click the icon and choose Options→Keep in Dock.**

   From then on, you can launch Xcode by simply clicking the Dock icon.

   Once Xcode is running, you'll see its Welcome window, as shown in Figure 1-6.

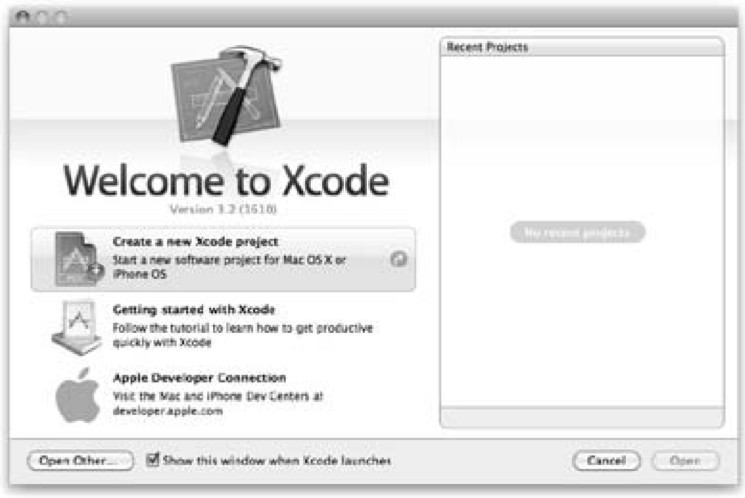
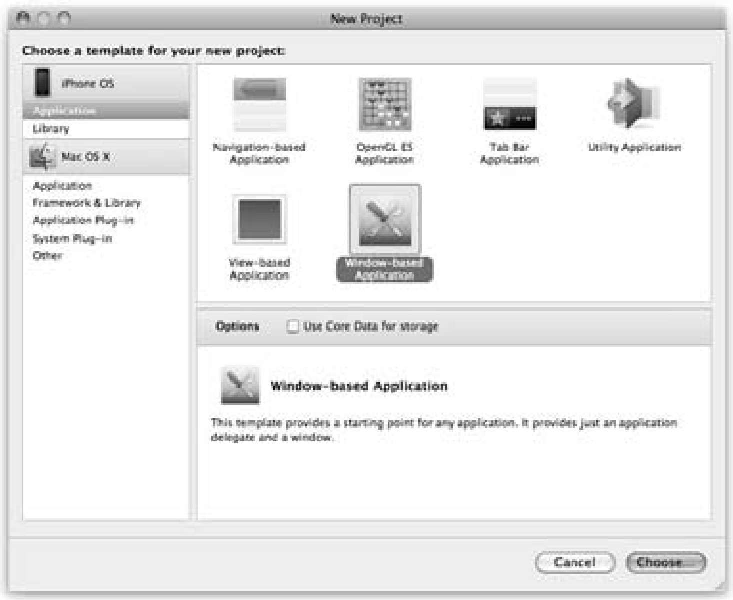

Welcome to Xcode
Version 3.2 (1610)
Create a new Xcode project
Start a new software project for Mac OS X or iPhone OS
Getting started with Xcode
Follow the tutorial to learn how to get productive quickly with Xcode
Apple Developer Connection
Visit the Mac and iPhone Dev Centers at developer.apple.com
Recent Projects
No recent projects
Open Other...
Show this window when Xcode launches
Cancel
Open

Figure 1-6: *The Xcode launch window. As you create new projects with Xcode, you see them listed on the right. Click the "Create a new Xcode project" button to start your first iPhone application. The "Getting started with Xcode" button opens the documentation viewer and displays a helpful overview of Xcode. The last button is a convenient link to the Dev Centers for the Mac and iPhone.*

Tip: If you close the Welcome window by accident, you can reopen it by choosing Help→Welcome to Xcode.

3. **Click the big "Create a new Xcode project" button.**

   The New Project window opens (Figure 1-7). In Xcode, a *template* is a predefined set of source code files, libraries, frameworks, and user interface elements that you use to create different styles of applications.



Window-based Application

This template provides a starting point for any application. It provides just an application delegate and a window.

Figure 1-7: *The Xcode New Project window lists all of the templates you can use to get a quick start. When you're starting out with a new application, select the template that best describes the style of user interface you want. When selected, each template displays a short description. Some templates even include options, like the one shown here to "Use Core Data for storage".*

4. **Since you're creating an iPhone application, under the iPhone OS group in the upper-left corner, choose Application and take a look at the available templates.**

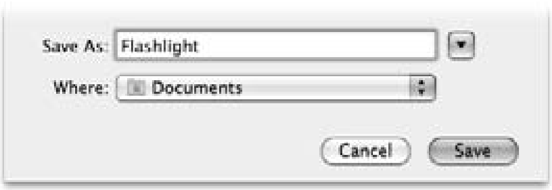
   Your choices come in the following categories:

   - **Navigation-based Application.** These applications have a "drill-down" style interface, like the iPhone Mail application.
   - **OpenGL ES Application.** Games that draw objects in a 3-D space use this template.
   - **Tab Bar Application.** This style of application uses a tab bar at the bottom of the screen to switch views. Apple's iPod application is a great example of this user interface style.
   - **Utility Application.** These applications generally present a simple interface, with a front view containing information and a back view for configuring the information. The built-in Weather app uses this metaphor.
   - **View- and Window-based Applications.** Turn to these templates when your application combines elements of the previous four styles. Think of them as bare-bones templates that you can customize to your own needs.

   For your Flashlight app, you're going to use the Window-based Application template. Since the application only uses a single window, this basic template is all you need. A nice side effect of using this customizable template is that it creates fewer files for the project. In effect, you have a shorter, simpler parts list.

5. **Click Window-based Application and then click Choose. Leave the Use Core Data checkbox unchecked since a flashlight doesn't need a database.**

   A save file dialog box appears so you can indicate a name and location for your project's folder (Figure 1-8).



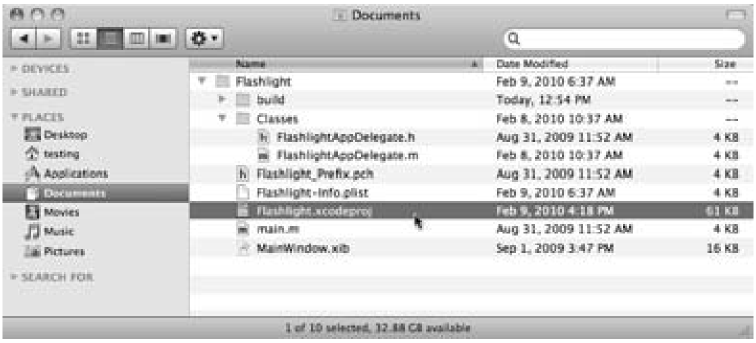
Save As: Flashlight

Where: Documents

Figure 1-8: *For the Flashlight project, tell Xcode to create a folder named Flashlight inside your Documents folder.*

6. **Type *Flashlight* for the project name, and choose the Documents folder from the bottom pop-up menu.**

   As shown in Figure 1-9, the folder you've just created in the Finder contains everything you need to build your application, including the main project file Flashlight.xcodeproj.



Name	Date Modified	Size
Flashlight	Feb 9, 2010 6:37 AM	--
build	Today, 12:54 PM	--
Classes	Feb 8, 2010 10:37 AM	--
FlashlightAppDelegate.h	Aug 31, 2009 11:52 AM	4 KB
FlashlightAppDelegate.m	Feb 8, 2010 10:37 AM	4 KB
Flashlight_Prefix.pch	Aug 31, 2009 11:52 AM	4 KB
Flashlight-Info.plist	Feb 9, 2010 6:37 AM	4 KB
Flashlight.xcodeproj	Feb 9, 2010 4:18 PM	61 KB
main.m	Aug 31, 2009 11:52 AM	4 KB
MainWindow.xib	Sep 1, 2009 3:47 PM	16 KB

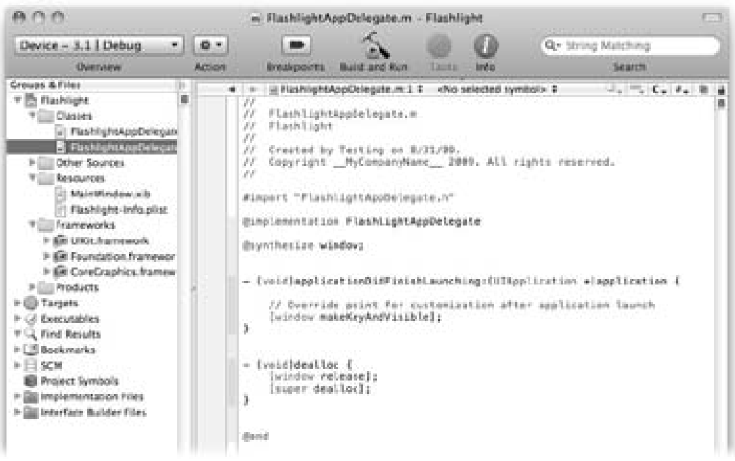
1 of 10 selected, 32.88 GB available

Figure 1-9: *When you create a project with Xcode, it creates a folder of files used to build your application. The most important one is the .xcodeproj file–you can double-click this file to open the project in Xcode. Also, since you're creating a window-based app, Xcode starts you out with a file called MainWindow.xib.*

Xcode creates this project folder behind the scenes. You may never in your Xcode career interact directly with files or folders. Instead, you can rely on Xcode to manage everything for you. But you still need to know where the folder is so you can back up your work.

Tip: As you get more advanced with Xcode, you may want to put your projects in folders of their own within your Home folder. Many developers create a Projects folder that contains nothing but their Xcode folders. Just as the Pictures, Movies, and Music folders make it easier to manage your media, a Projects folder makes it easier to manage your software.

After Xcode finishes creating the new project, it displays the files in a project window, as shown in Figure 1-10.



```
//
//  FlashlightAppDelegate.m
//  Flashlight
//
//  Created by Testing on 8/31/09.
//  Copyright __MyCompanyName__ 2009. All rights reserved.
//

#import "FlashlightAppDelegate.h"

@implementation FlashlightAppDelegate

@synthesize window;


- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // Override point for customization after application launch
    [window makeKeyAndVisible];
}


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


@end
```

Figure 1-10: *The Xcode project window. On the left are the project's groups and files, and on the right is the source code editor. Although the yellow group icons look like folders, they're not the same as the blue ones you see in the Finder. You can rename the Classes group and not affect the folders on disk. Likewise, you won't find a Resources folder in the Finder, but the group is a great way to organize files that aren't source code.*

The Groups & Files panel on the left side of the project window lists the individual files that make up your application. To go back to the parts list metaphor, each group contains parts of a similar type. Here are a few of the most important groups:

- **Classes.** The files in this group contain your project's actual source code.
- **Resources.** User interface files, graphics, and application configuration files all fall under the Resources group.
- **Frameworks.** These files contain tools that the iPhone SDK uses.

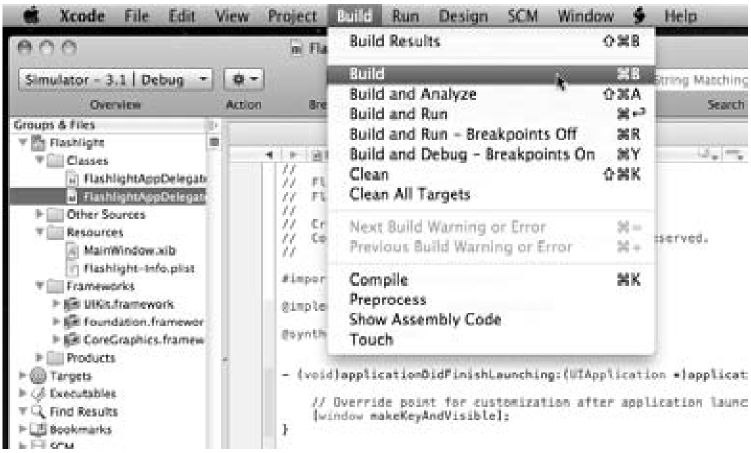
The editor part of the window (the big white area in the lower right) shows the code that runs when the application finishes launching. In this simple example, the code makes a window object visible and able to respond to taps.

You'll learn more about these important groups and the source code in the next chapter. Remember: Your goal in this chapter is to build an app without writing any code!

Some Assembly Required

You've gathered all the parts, and now it's time to assemble them. Unlike toys on Christmas Eve, Xcode projects are easy to put together. Thanks to the template, all you have to do is initiate the Build command, and Xcode takes all of the source code, resources, and frameworks and combines them into an executable file that can run on the iPhone.

Make sure that the pop-up menu in the upper-left corner of your project window is set to Simulator and Debug (the exact wording on the menu will change depending on which version of the iPhone SDK you're using). Then choose Build→Build, as shown in Figure 1-11.



```
- (void)applicationDidFinishLaunching:(UIApplication *)applicat
    // Override point for customization after application launc
    [window makeKeyAndVisible];
}
```

Figure 1-11: *Building your iPhone application. Note that the Overview menu is set to the Simulator and Debug. The keyboard combination ⌘-B is a handy shortcut to build the project after you've made a change to the source code.*

After a short wait, you should see "Build succeeded" in the status bar, at lower left.

Note: Xcode's status bar, which runs along the bottom of the window, is an important information source. As you perform various tasks, this area keeps you posted on their progress. It's the first place you should look when you're wondering what's going on.

Once you've built your app, you can run it on your iPhone. Or better yet, run it on your Mac. That's right. If you're like most developers, you'll run your iPhone apps on the Mac about 90 percent of the time. Apps launch faster on the Mac than on the iPhone, and they're easier to debug on the Mac when problems occur.

Note: Don't discount that last 10 percent. Running your app on an actual device with an actual multitouch screen is extremely important, so you can catch design and device issues.

Taking It for a Run on Your Mac

When you develop an iPhone app, you'll probably run it on your Mac to test and debug it before it ever gets near an iPhone.

So how do you get an iPhone onto your Mac? It's easy: Make sure Simulator is selected in the Overview menu, and choose Run→Run. Since you have Simulator selected, Xcode uses a simulation of the iPhone to run your app.

Keep an eye down on the status bar. You'll see "Installing Flashlight in Simulator" displayed, and eventually, "Flashlight launched". Soon after, a giant iPhone appears on your desktop, and it's running your Flashlight application (Figure 1-12). Congratulations!

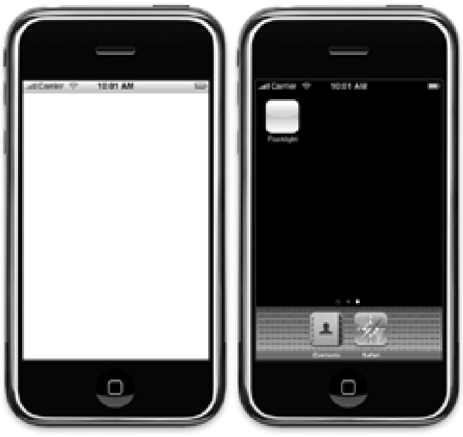


Figure 1-12: *In the iPhone Simulator, the image on the left shows the application running, and the one on the right shows the application's icon on the home screen. It won't fit in your pocket, but the simulator acts just like a real, live iPhone.*

Tip: You can do everything you've done in this section and the previous one with a single keystroke. Pressing ⌘-Enter builds and runs your application (in the simulator) in one step. When you get into the thick of iPhone app development, you'll be pressing these keys in your sleep.

So what does this giant iPhone do? It's called the iPhone Simulator, and it behaves like the device in your pocket, except:

- It's hundreds of times faster.
- It has as much memory as your Mac.
- The network is much more reliable.
- It has a larger display.
- It doesn't sync with iTunes over a USB cable.
- Touching the simulator screen has no effect.

In reality, this big phone sitting on your desktop has very different hardware specifications than the one you're used to. But once you get used to keeping your fingers off the screen, you'll end up loving the simulator. It makes your life as a developer so much easier because it does one important thing: it lets you test your code without having an iPhone plugged into your Mac. The box on page 25 shows you how to get the most out of this important tool.

When you click the Home button, you see a screen with the applications installed by Xcode. Since you only have one at this point, all you'll see is the Flashlight app. Drag your mouse to swipe between the pages of applications. Many of the applications you're used to seeing on your iPhone have gone missing, but you'll still be able to use Photos, Contacts, Settings, and Safari while testing. Clicking an app's icon launches it in the Simulator just as it would on a real device.

Tip: The Safari icon in the simulator's tray is very helpful for testing how websites will look on the iPhone. When you start promoting your application, you'll want to use the simulator to check your product pages on a mobile device.

POWER USERS' CLINIC

Simulating Reality

The iPhone Simulator acts very much like the device in your pocket. Sometimes, however, it's not obvious how to make the virtual device behave like the physical one.

When you hold down the Option key, two dots appear. These dots show the position of multitouch events when you click the mouse button. Use this feature to simulate the pinching and spreading gestures that zoom the iPhone screen.

You should also check out the Simulator's Hardware menu. Two commands on this menu let you rotate the device left and right—very handy if your app detects device orientation. Choose Hardware→Rotate Left or Rotate Right.

If your application uses shake gestures, there's a menu item to simulate one. Shaking is a standard part of the iPhone copy and paste mechanism (it's used for undo).

The Hardware menu also lets you simulate the status bar that the iPhone displays when you're on a phone call (choose Hardware→Toggle In-Call Status Bar). That way, you can verify that your application resizes its user interface (UI) correctly when the display window is 20 pixels smaller. (The iPhone displays the same "double height" status bar when using Internet tethering.)

—Continued

As you get more advanced in iPhone development, you'll have occasion to use the Hardware→Lock command. You can make your app detect when the iPhone sleeps and wakes up, and Lock is the way to simulate that.

Another advanced simulator feature is memory warnings. The iPhone has a limited amount of memory available to applications. Hardware→Simulate Memory Warning lets you test how your application behaves when the phone runs out of memory. Many developers run into problems when they go from an unlimited amount of memory on their Mac to 128 MB on a device. Simulate Memory Warning helps you avoid that fate.

It's also important to realize that the simulator is sharing many of the resources on your Mac. Since both are built on top of OS X technologies, things like the network are common to both platforms: take advantage of the similarity. For example, if you want to see how your app behaves when it loses the cell network, just go into your Mac's System Preferences, and turn off the network interface.

Revision Decision

So now that you have a running application, you notice that the white light makes it look like every other flashlight app in iTunes. You need a better color for the light—a way to stand out in the crowd. How are you going to do that without breaking the "no code" rule? The answer is simple: Interface Builder.

Xcode works hand-in-hand with Interface Builder to create your app's user interface (UI). You create objects like windows and buttons with this tool and then drop them into your code. When you created the project from a template, Xcode created a file containing these objects automatically.

To get an idea of how easy this method is, open your Flashlight app's UI. From the main project window, open the Resources group by clicking the disclosure triangle, and double-click the MainWindow.xib file. Interface Builder opens (you'll see it bouncing in the Dock). Once it's open, you're ready to start working on your UI (Figure 1-13).

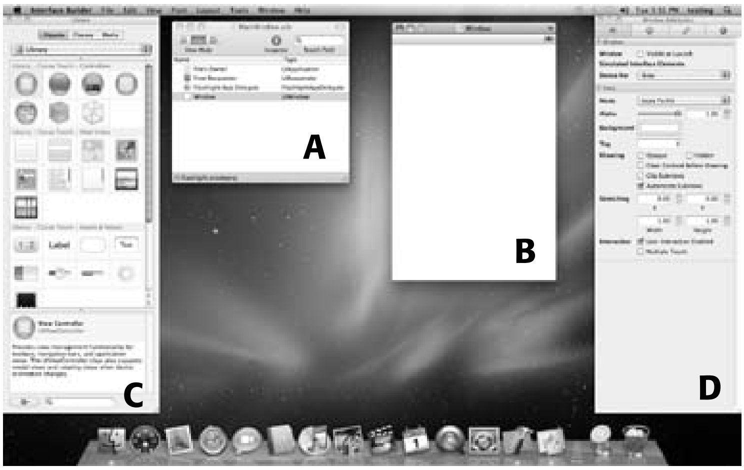


Figure 1-13: *Interface Builder and its many windows take up much of your display. In the middle are the .xib document (A) and the window displayed in the iPhone app (B). To the left is the library of user interface elements where buttons, windows, and other user interface components can be accessed (C). A property inspector for the objects in the interface is displayed on the right (D).*

The main document window is MainWindow.xib. To change the View Mode, use the three buttons in the upper-left corner. The buttons work much like they do in the Finder: the leftmost displays icons, the center a list, and the right one shows columns. The list view (which you can see in Figure 1-13), is more compact and easier to read (especially with longer names).

The Library window, at far left, contains a list of all the interface objects you can use in your design. You'll learn about these objects in detail in Chapter 3.

To start modifying the Flashlight's UI, double-click the Window item in the list in MainWindow.xib. The app's window opens to the right of the main document window and gives you the opportunity to modify the color. Start thinking about your favorite color!

At the far right is the Property Inspector. You'll use this window often as you refine your UI. Currently, it's showing Window Attributes because you're working with a window. The inspector is split into four main sections, which you choose from the tabs at the top. Each section lets you adjust various aspects of each object:

- **Attributes.** The object's particular settings. Behind the scenes, these items set properties and attributes for the object to save you from writing code (although you can still write the code manually if necessary).
- **Connections.** The connections define how your source code accesses the UI objects. For the Flashlight app, you'll see that "window" is connected to "Flashlight App Delegate".
- **Size.** This panel lets you define the selected object's geometry. For example, the window's width and height is 320 x 480 (the size of the iPhone's screen).
- **Identity.** This view shows what kind of object is defined. The app's window has a class of UIWindow.

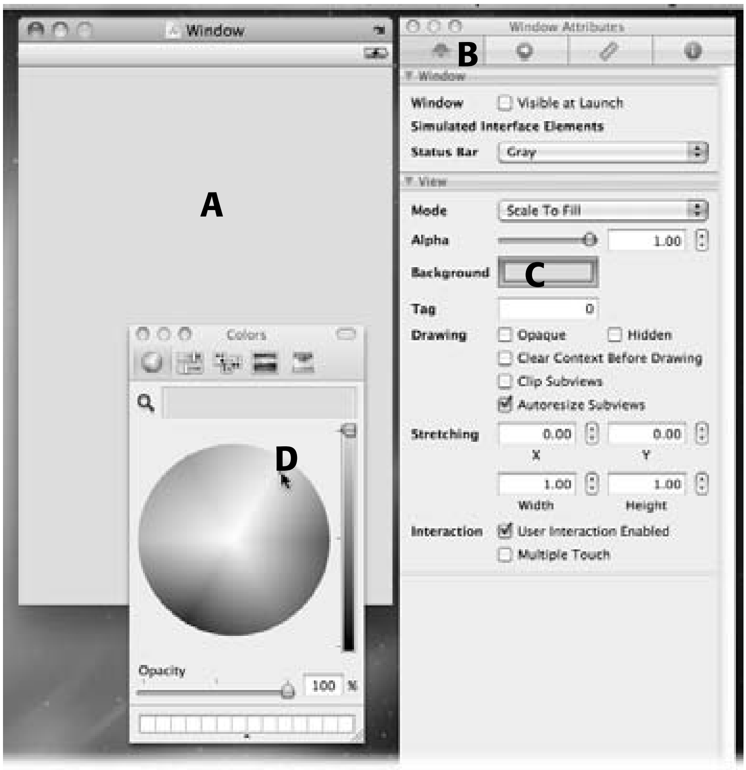
Tip: To manage Interface Builder's many windows, you'll find that keyboard shortcuts make short work of finding the information you need. You can access the main document window by pressing ⌘-0, and you can switch to each section of the inspector with ⌘-1 through ⌘-4. Use Shift-⌘-L to bring up the Library window.

If you don't fully understand what's going on here, don't worry. You'll learn everything there is to know about classes, objects, and instances in the next chapter.

Note: If you've used other development environments, leave your preconceptions about how Interface Builder works at the door. Instead of automatically generating code to display the UI from resources, the .xib file contains an XML representation of the actual object instances and hierarchy. When you load a file, Interface Builder creates an object graph in memory and connects it to the instance variables that you've chosen. There's more detail about how this works in Chapter 3.

Now that you've had a little tour of Interface Builder, you can modify your flashlight's UI. Make sure that Window is selected in the MainWindow.xib document and that the Property Inspector is on the first panel (for modifying attributes).

Have you picked your favorite color yet? Time to change the background color of the window by clicking the Background color picker. Then click a nice color for the light in the color wheel (Figure 1-14). A soothing shade of yellow, for example.



▼ Window

Window	☐ Visible at Launch
Simulated Interface Elements	
Status Bar	Gray

▼ View

Mode	Scale To Fill
Alpha	1.00
Background	
Tag	0
Drawing	☐ Opaque ☐ Hidden ☐ Clear Context Before Drawing ☐ Clip Subviews ☑ Autoresize Subviews
Stretching	0.00 X, 0.00 Y, 1.00 Width, 1.00 Height
Interaction	☑ User Interaction Enabled ☐ Multiple Touch

Figure 1-14: *Changing your application window's background color involves clicking a color wheel. After selecting the Window in the MainWindow.xib document (A), you view the attributes (B), which include a background color. Clicking in the well (C) brings up a color wheel (D) where you can select a new color with your mouse. As you select new colors, you get immediate feedback.*

The window preview updates immediately as you select colors. As your UI develops, you'll really appreciate this quick feedback: you don't have to build your app to see how a change will look, and that saves a lot of time.

Choose File→Save to update your MainWindow.xib file. Get in the habit of saving after tweaking your UI, and remember to check for the "modified" dot in the document's close box if your changes aren't working right. The last thing you want is to waste time debugging a user interface bug because you forgot to save the file.

Now switch back to the Xcode project window for the Flashlight, and select Build→Build and Run.

Note: If you see a window pop up with a warning to Stop Executable, just click OK. Xcode is just reminding you that you already have an application running in the simulator.

You've just made your first iPhone application and customized it to fit your own tastes. Well done!

Now that the journey is complete, read on to look at some details for things you saw along the way.

The Power of Brackets

You've now seen how easy it is to build an iPhone app: It's not rocket science. Or is it?

Apple has done an excellent job of creating the tools you use for building iPhone applications, but if you're going to become a master craftsperson, you need to learn more about these new tools. The first section of this chapter will get you started in this exploration: You're going to take a look the basics of the Objective-C language used to create apps.

Also, a thriving community of developers is working on the iPhone. You'll learn where to look for help with your coding questions, how to keep up with the ever evolving technologies used in the iPhone, and where to download free sample code to use in your own app development.

Note: There's a lot to learn in Objective-C. This chapter covers the topics that you're most likely to encounter in iPhone development. At the end of the chapter, you'll see some recommendations on where to look for additional information.

Once you've gotten up to speed with the essential parts of the language, you'll take a quick look at the documentation viewer that's built into Xcode (page 80). Knowing how to find information quickly and effectively will ease your entrée into this new platform.

Objective-C: The Nuts and Bolts for Your iPhone App

In this section, you'll explore the language used for programming iPhone applications: Objective-C. This chapter won't teach you how to program, but rather will show you the differences between this language and others you might have used. You'll also encounter features that are exclusive to Objective-C.

The Land of Square Brackets

While you were creating your flashlight, you might have noticed this line of code:

```
[window makeKeyAndVisible];
```

If you've programmed in Java, C, or JavaScript, those square brackets probably made you scratch your head. So what are those funny brackets for? To find the answer, take a look at the language's name—the *C* on the end. You see, this whole new language is really based on an old one: the C language developed by Dennis Ritchie at Bell Telephone Laboratories in 1972. As with Java and JavaScript, the creators of Objective-C built something new on top of something familiar to many programmers. Once you learn a few tricks, this language will quickly become familiar to you, too.

UNDER THE HOOD

C Plus Stuff

Objective-C is a superset of the C language syntax. Everything you see in this chapter is built using the GNU C/C++ compiler.

When a source file uses .m rather than .c, additional processing on the file occurs (similar, in concept, to using a preprocessor). This step hooks your code into a small runtime that supports the features of Objective-C. Classes and objects are simply C structures; the runtime lets them communicate with each other via introspection and message passing.

Because the entire infrastructure is based on standard C, it's easy to mix in open source libraries and legacy code with your Objective-C application. You don't have to give up your prized C/C++ source code when starting with this new language.

That said, many developers find working in this object-oriented environment quite addictive. It's fairly common to see Objective-C used as a wrapper that makes an open source library easier to work with. A good example is with regular expressions: Cocoa Touch has no native support for them. Thanks to the work of John Engelhart, the RegexKit Framework *http://regexkit.sourceforge.net* makes Perl Compatible Regular Expressions (PCRE) a joy to use with your string objects.

After you've had a chance to get familiar with the syntax and feel of this new language, you'll explore how standard C idioms are used to implement it in the box on page 78.

Take a look at the following Objective-C code. You'll immediately feel comfortable with the familiar expressions and control structures you've used in other languages:

```
int i;
for (i = 0; i < 10; i++) {
    // check for even and odd values
    if ((i % 2) == 0)
        [label setTitle:@"Even"];
    else
        [label setTitle:@"Odd"];
}
```

In this contrived example, it's easy to see that you're checking for even and odd values in a loop that executes 10 times.

The first part of the language's name gives you a hint as to what those brackets are used for: *Objective* tells you that you're dealing with objects. And those confounded brackets let you communicate with those objects. In the first example, you were communicating with a *window* object. The second example interacts with a *label* object.

The Object of It All

So now you're probably thinking, "Great, but what's an object?" Objective-C is what computer scientists call an *object-oriented* language. You're going to be doing object-oriented programming as you develop your iPhone application.

Tip: If you're new to object-oriented programming, take a moment to read the Wikipedia page on this topic. The overview and history will help you come up to speed on the design motivations and terminology used in this style of programming: *http://en.wikipedia.org/wiki/Object-oriented_programming*.

Objects are chunks of memory that contain code and data for your application. One of the main tenets of object-oriented design is that your application should know as little as possible about the inner workings of an object. Terms like *encapsulation* and *data abstraction* describe this behavior.

An object should be like a black box. In fact, it may be helpful for you to use those square brackets as a visual reminder: [] looks a lot like a box.

As soon as you turn on your iPhone, thousands of objects are created in memory. Thousands more are added as soon as you launch your own application. You'll often hear *instance* used to describe each of these objects. With so many instances floating around, you'll need variables to keep track of the important ones.

Telling Your Objects to Do Things

The variables that reference objects let you change an object's behavior or state. You do this by sending a *message* to the object. In the Cocoa documentation, the object that's getting the message is often referred to as the *receiver.*

Note: You'll also see the term *target* used to represent the object that's getting the message. The message itself is sometimes called an *action,* since messages often cause things to happen. You'll see these terms often when dealing with user interfaces (in Interface Builder).

For the *window* object shown at the beginning of this section, you sent a message to *–makeKeyAndVisible*. In essence, the message is telling the window to make itself visible onscreen. Your "label" variable was sent the message *–setTitle:* along with a variable; it's being instructed to change its title to a new value. Don't worry about the action that takes place—your main focus right now is to understand how actions are initiated using messages.

As you get used to this new programming syntax, the square brackets remind you that you're communicating with a black box. The first word specifies the object that's receiving the messages, and the remaining words supply any other parameters that are needed. Here are some examples:

```
[myBlackBox messageWithNoParameters];
[myBlackBox messageWithFirstParameter:one
andSecondParameter:two];
```

Note: If you're coming from a background in another object-oriented language, such as C++ or Java, sending messages is functionally equivalent to this:

```
myBlackBox->functionWithNoParameters();
myBlackBox->functionWithParameters(one, two);
```

or this:

```
myBlackBox.functionWithNoParameters();
myBlackBox.functionWithParameters(one, two);
```

The big difference is that the messages you send to objects can be generated at runtime as well as when your app is compiled. Similarly, object methods can be modified while your code is running. Objective-C is a dynamic language, so you can change the behavior of your objects at any time.

Each of these messages is handled by a *method* defined by the object. A method uses a unique signature that lets Objective-C route your message and any parameters to the right code for execution. You'll find that many method signatures are quite readable in your code: It's a little more typing, but you'll end up with code that's easier to understand.

Some methods return values. Earlier, you saw the *setTitle*: method change a label's title. There's a corresponding method called *title* that lets you query the object for the current value. For example:

```
NSString *labelTitle = [label title];
```

You're now thinking outside the black box. But what the heck is that *NSString* thing?

It's a variable definition. Just like any variable definition in C, the *labelTitle* variable is a pointer (because of the asterisk) to a class called *NSString*. That's great, but what's a class? Read on.

Masses of Classes

Every object in Objective-C belongs to a *class,* because that's where behavior and internal data are specified. A class is where the methods (code) are defined and implemented. It's also where you define what instance variables (data) are managed by the class.

NSString is a class that provides a rich set of methods for storing and manipulating text. The `NS` in the name means it's a part of a system foundation framework in Cocoa Touch.

Note: Frameworks are collections of classes and other resources you can add to your application. You'll learn all about the Cocoa Touch framework that supplies all the interesting classes, like *NSString*, in the next chapter.

Also, since Objective-C lacks namespaces, frameworks typically use a two-character prefix to prevent class name collisions. Foundation classes, which were originally written for NeXTSTEP, use *NS.* The user interface classes in Cocoa Touch use *UI.*

Some developers use their own prefix for class names to avoid these types of conflicts. For example, the Iconfactory uses *IF* for all classes that are shared among projects.

Therefore, *labelTitle* points to an instance of a string object that's defined by the system.

Because you have this object pointer in a variable, you can send messages to it. For example, you can send the *uppercaseString* message to your *labelTitle*, and it will return a new string object that contains all the lowercase letters converted to uppercase letters:

```
NSString *moreAwesomeLabelTitle = [labelTitle uppercas-
eString];
```

And that *moreAwesomeLabelTitle* variable, in turn, can be used to update the title on the original label:

```
[label setTitle:moreAwesomeLabelTitle];
```

Tip: Nesting objects and their messages can be an effective way to shorten your code and reduce the number of temporary variables. You can collapse these three lines of code:

```
NSString *labelTitle = [label title];
NSString *moreAwesomeLabelTitle = [labelTitle
 uppercaseString];
[label setTitle:moreAwesomeLabelTitle];
```

into a single line of code by using this technique:

```
[label setTitle:[[label title] uppercaseString]];
```

Just make sure to balance the brackets correctly or the compiler will complain!

You might have wondered earlier what the @*"Even"* and @*"Odd"* parameters were for in the *setTitle:* method on the label object. Good catch!

Since strings are used a lot in programming, the creators of Objective-C defined a shortcut for these text objects. Without the shortcut, @*"Even"* would need to be instantiated using:

```
[NSString stringWithUTF8String:"Even"]
```

That's a lot more typing. And the best programmers are the lazy ones. Welcome to the club!

Classes in Detail

At this point, it should be pretty obvious that classes are really important in Objective-C. They act as building blocks for your application, and they let you avoid writing a lot of boring code. It's not hard to envision how an *uppercaseString* method would be implemented, but it's even better to not think about it at all and to just let the system handle the work.

Classes are defined in header files that are included in your source code. System-level classes, like *NSString*, get incorporated by default. Take a look at the top of the FlashlightAppDelegate.h file that was created automatically for you in the last chapter:

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

That one line of code pulls in the class definitions for the entire system. The preprocessor begins by loading the classes for developing Cocoa Touch user interfaces (*UIKit*). These classes, in turn, load foundation classes for managing data and the core graphics classes for drawing.

Note: For you experienced C programmers, note that there's a subtle difference between *#include* and *#import*: The preprocessor will keep track of imports and not include them more than once. This technique makes it much easier to avoid multiple definitions!

What do these header files for classes look like? Here's a part of the class definition for *NSString* taken from the NSString.h header file:

```
@interface NSString : NSObject <NSCopying, NSMutableCopy-
ing, NSCoding>
- (NSUInteger)length;
- (unichar)characterAtIndex:(NSUInteger)index;
@end
```

Even if you have experience working with other programming languages, this code won't make any sense. Time to deconstruct the syntax!

First there's *@interface*:

```
@interface NSString : NSObject <NSCopying, NSMutableCopy-
ing, NSCoding>
```

This code tells Objective-C that you're about to define data and methods for a class. This, in effect, is a template for every object that gets instantiated from the class. The interface is also where the class name is defined as *NSString*.

The interface definition continues until the `@end` and the, well, end.

One class to rule them all

After definition, there's a colon and some other names beginning with `NS`. That's your hint that you're dealing with system class definitions. The most important one is *NSObject*:

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

All classes, and therefore objects, in Objective-C "inherit" from another class. *Inheritance* lets the characteristics of a parent class pass on to a child class. The child class—more commonly referred to as a *subclass*—can then modify the methods and instance variables that were defined by the parent. (The parent is often called a *superclass*.)

Tip: There's one exception to this inheritance rule, and that's *NSObject*. It's the root class in the hierarchy, and it doesn't inherit from any other class.

Much as with your parents, inheritance gets you a lot of behavior for free (even if you don't want or need it!). Even if there's no money involved with this inheritance, you'll find it invaluable because it saves you a lot of time and effort. (For details on this "free code," see the box on page 43.)

An example of inheritance occurs in custom views. It's likely that at some point in your development, you'll want to develop a custom view for displaying some application-specific data. You'll start this work by subclassing a system class like *UIView*.

As you implement your child class of *UIView*, you'll find that a lot of code has already been done for you. Things like handling multitouch events, drawing, animation, and view management are available from the parent class. You only have to implement the new behavior for the view.

POWER USERS' CLINIC

Tell Us about Yourself

The concept of free code extends all the way through the object hierarchy. As you're working on a *UIView* subclass, you'll have access to the code from *UIView*, *UIResponder* (where touch events are handled), and *NSObject* (the root class).

To see the power of the hierarchy, look at the *–description* method in *NSObject*. Since every object is a subclass of this root class, they all know how to describe themselves. The default implementation of the method returns the object's class name and a memory address. So even if you don't know anything about an object, you can do this:

```
[mysteryObject description]
```

The resulting string that describes the *mysteryObject* is helpful when you're debugging or working with data from a source outside of your direct control.

For example, the debugger that you use with Xcode (gdb) includes a *print-object* command. When you type this into the debugger:

```
(gdb) po myObjectVariable
```

a description message is sent to the object, and the result is then printed on the console.

This behavior can also be handy when you're logging information. Cocoa's logging function, *NSLog*, takes a *printf*-style string and outputs the results to the current console device:

```
NSLog(@"THIS OBJECT IS AWESOME: %@", chockLockObject);
```

The formatting specification is supplied with an *NSString* (remember the @ shortcut mentioned earlier). There's also a new formatting specification—%@—that prints an *NSString* just as %s prints a C-style string.

The really cool part is that the logging function is also smart enough to know that *chockLockObject* isn't an instance of *NSString*, so it uses the object's description method to create the string that's displayed.

Now that you've learned a bit about the class hierarchy in Objective-C, it should be clear that the first word after the colon is the superclass (parent) of the class being defined. *NSString*'s superclass is *NSObject*.

Follow the protocol

After the superclass name, you'll see some other class names in angled brackets (< >):

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

These are *protocols* that are adopted by the class. Protocols define a group of methods that aren't associated with any particular class.

Using protocols is like signing a contract. If you specify that your class adopts a protocol by including the class names in the angle brackets, you promise to implement the methods that are defined in the protocol. For example, the *NSString* protocol promises to fulfill the contract of providing copies of itself. These object copies can be read-only (with *NSCopying*) and writable (with *NSMutableCopying*). The *NSCoding* protocol tells you that the class also implements methods for encoding and decoding objects.

Note: The encoding and decoding methods in *NSCoding* are used to archive objects on disk or to distribute them across a network. It's Cocoa's mechanism for serializing objects.

Protocols aren't required when you're defining a new class, but you'll find them used quite a lot as you develop your app. The most common use of protocols is with delegates; you'll see that design pattern in the next chapter.

Note: Since Objective-C supports only single inheritance, protocols are often used to achieve the same goals as multiple inheritance in other languages.

The Methods Behind the Madness

That's a lot of reading just to understand a single line of code! Thankfully, it gets a little easier with the next two lines, both of which begin with a minus sign (–).

```
    - (NSUInteger)length;
    - (unichar)characterAtIndex:(NSUInteger)index;
```

This is how you define methods in a class interface. You can represent any string in Cocoa with just these two methods: The first returns the length of the string; the other returns a Unicode character at an index position.

Tip: When writing about methods in online forums, mailing lists, or blogs, many developers use an abbreviated form for the signatures: They leave out type information and parameter names. Apple's developer documentation also uses this format.

It's common to see something like "I'm having problems with *–messageWithFirstParameter:andSecondParameter:*" in a post asking for help. Another reason to keep those method signatures readable! The short forms for the *NSString* methods shown above are *–length* and *–characterAtIndex:*. This book uses this format throughout.

The definitions also include types for any parameters and the result. The *–length* method returns an unsigned integer (*NSUInteger*). The *–characterAtIndex:* method takes one parameter with an unsigned integer index and returns a unichar. These methods use primitive types, but you can also specify objects using a pointer to a class name (you'll see that shortly with the *–uppercaseString* definition).

Warning: The types used in the method definition are checked at compile time, but not when messages are sent at runtime. When you see a compiler warning that "'NSString' may not respond to '–aNonexistentMethod'", make sure that you're sending the message to the right kind of object.

When the compiler warns about "Passing arguments" from a method, that's your clue that you have mismatched types in the message you're sending to an object. If you see a message about incompatible pointer types, make sure you're using *@"string"* instead of a standard C *"string"*. The "without a cast" warnings are usually an indicator that you've used a primitive type instead of an object (or vice versa).

If you ignore these warnings, you'll still be able to run your app, but it will crash as soon as these incompatible types are sent in a message.

Note that this is just a class definition, so there's no mention of where or how your string data is stored. Remember that you don't really need to know about the inner workings of an object. (You'll learn to love this encapsulation thing.)

But you may be asking yourself, "What happened to that *–uppercaseString* method I saw used on page 39?" That's a good question, and it leads into another powerful, and unique, feature of Objective-C: categories.

Categorically Speaking

One of the problems with important classes—like the ones for handling strings—is that they can grow quite large and unwieldy. Categories help avoid this by breaking the class definition into pieces. In the case of *NSString*, only the most basic functions are present in the main class definition. The real meat of the class is in this category:

```
@interface NSString (NSStringExtensionMethods)
```

There's the *@interface* you saw earlier (page 41), followed by the class name. It's only when you get to the parentheses that things get interesting.

The parenthesized name gives you a hint as to what's going on here. This class interface supplies methods that extend the previously defined *NSString*. It's adding functionality without cluttering up the basic definition of a string.

Tip: There's an art to naming in Objective-C. It's OK to be wordy, as with the *NSStringExtensionMethods* category name. Just make sure the name is descriptive enough that you can understand it without referring to documentation.

This string category contains the following method definition:

```
- (NSString *)uppercaseString;
```

That's the exact method you saw on page 39.

But categories get even cooler. You can use them to extend classes that already exist (and that you don't even have the source code to!). Imagine you need to make all your strings *awesome*. You don't really know what awesome means yet, but you do know that you'll need to do it often.

You start by defining your own category like this:

```
@interface NSString (AwesomeMethods)
- (NSString *)awesomeString;
@end
```

You're basing your category on the existing class *NSString*, and you assign it a category name of *AwesomeMethods* by including it between the parentheses.

The new class will have all the capabilities of a normal *NSString*, but it will also respond to one additional message named *–awesomeString*. But it gets even better: All other string objects in the system will understand the new message. If you had implemented *–awesomeString* in a subclass, you'd spend a lot of time refactoring your own code to use the new implementation, and have issues with code out of your control (such as strings returned by a Cocoa API). Categories are a powerful construct that lets you extend the system without breaking what is already there.

All you need now is some code to make those awesome strings.

Implementation: The Brains Behind the Beauty

For every *@interface,* there's an *@implementation*. The easiest way to think about it is that *@interface*s are what your class looks like from the outside, and the *@implementation* is what it looks like from the inside.

Note: So far, you've been looking at what goes in an interface header file. These files, like in C, use the .h extension by convention. An interface's implementation goes in a .m file. Just remember that *M*ethods go in a file that begins with the letter *M.*

Here's what the implementation could look like:

```
@implementation NSString (AwesomeMethods)

- (NSString *)awesomeString {
    NSString *awesome = [self uppercaseString];
    if (! [self hasSuffix:@"!"]) {
        // ADD SOME IMPACT!!!
        awesome = [awesome
stringByAppendingString:@"!!!"];
    }
    return awesome;
}

@end
```

By now, you're getting better at reading Objective-C code. You'll see that an awesome string is created using your old friend *–uppercaseString*. The *–hasSuffix:* method checks to make sure that there's at least one exclamation point at the end. If not, the *–stringByAppending String:* method adds the necessary impact. But what does that *self* mean?

Every method implementation is passed a hidden argument named *self* that references the object receiving the message. Since your method is a part of the *NSString* class, *self* refers to an instance of that class. (Yes, you're talking to yourself in this method, and that's OK.)

Note: C++, Java, and PHP all use *this* to provide this self-reference.

Once you've implemented this category method, your code can call it like this:

```
NSString *ordinaryString = @"typing power";
NSString *excitingString = [ordinaryString awesom-
eString];
```

The `excitingString` object will have the value "TYPING POWER!!!".

Categories also help you avoid repetition in your own code. Suppose you do a lot of checking of strings to see if they're awesome. You'd be writing code like this over and over again:

```
NSString *myString = @"something";
if ([myString isEqualToString:[myString awesomeString]])
{
    // sorry, but myString isn't awesome
}

NSString *myOtherString = @"SOMETHING ELSE!!!";
if ([myOtherString isEqualToString:[myOtherString
awesomeString]]) {
    // myOtherString, on the other hand, is truly awesome
}
```

That's a pain to type and a pain to read. So add this to your `NSString`'s category interface:

```
@interface NSString (AwesomeMethods)
- (NSString *)awesomeString;
- (BOOL)isAwesomeString;
@end
```

And add this to its implementation:

```
- (BOOL) isAwesomeString {
    return [self isEqualToString:[self awesomeString]];
}
```

This cleans up your checking code quite a bit:

```
NSString *myString = @"something";
if ([myString isAwesomeString]) {
    // sorry, but myString isn't awesome
}

NSString *myOtherString = @"SOMETHING ELSE!!!";
if ([myOtherString isAwesomeString]) {
    // myOtherString, on the other hand, is truly awesome
}
```

It's also important to note that at this point you haven't actually created a new class. You've only extended code provided in a system framework to suit your own needs. Cocoa Touch is awesome; you just made it more awesome.

Creating New Classes

As you can see, categories can accomplish a lot and provide a great way to extend code that you didn't write. But they have one major limitation: You can't add instance variables in your category definition. And with the *NSString* class, how could you? The class interface tells you nothing about how the string is stored.

It's time to learn how to create new classes and to extend the Cocoa hierarchy. And to see how that's done, you're going to create a new class that lets you control the number of exclamation points, as in the phrase "THIS IS GOING TO BE AWESOME!!!!!!!"

The first thing to do is create an *@interface* for this new class:

```
@interface AwesomeStringMaker : NSObject
{
    NSNumber *exclamationCount;
    NSString *originalString;
}
- (NSNumber *)exclamationCount;
- (void)setExclamationCount:(NSNumber *)
newExclamationCount;
- (NSString *)originalString;
- (void)setOriginalString:(NSString *)newOriginalString;

- (NSString *)awesomeString;
@end
```

This code looks similar to the class category on page 50. This time, the category name (in parentheses) is gone, and some new code is in curly braces: { }. A class's data is specified between these braces.

This example has two instance variables: a number that keeps track of the number of exclamation points and a string that you want to make awesome.

Note: You'll find that class data is referred to in many ways. Objective-C developers use the terms *instance variable, ivar,* and *property* interchangeably. If you're talking to a longtime C++ developer, you may hear *member variables.* Java diehards use *fields.*

Whichever term is used doesn't matter; it's still just a block of memory that's associated with each instance of your class.

A few new methods at the end of the interface let you read and write the data managed by this new class. These methods are called *accessors* because they let you access the class's internal information. By convention, the method that reads the instance variable just uses the name. The method that updates the instance variable is prefixed with *set.* Other languages call these methods *getters* and *setters.*

Only the implementation of this class has easy access to all instance data. Without the accessor methods, the data is essentially hidden. In most cases, that's a good thing, especially when you have private data that you don't want exposed to the caller.

Now for the *@implementation* of your *AwesomeStringMaker* class:

```
@implementation AwesomeStringMaker

- (NSNumber *)exclamationCount {
    return exclamationCount;
}

- (void)setExclamationCount:(NSNumber *)
newExclamationCount {
    if (exclamationCount != newExclamationCount) {
        [exclamationCount release];
        exclamationCount = [newExclamationCount retain];
    }
}

- (NSString *)originalString
{
    return originalString;
}

- (void)setOriginalString:(NSString *)newOriginalString {
    if (originalString != newOriginalString) {
        [originalString release];
        originalString = [newOriginalString copy];
    }
}

- (NSString *)awesomeString {
    NSString *awesome = [originalString uppercaseString];
    NSUInteger length = [awesome length];
    NSInteger padding = [exclamationCount unsignedInt-
Value];
    NSString *moreAwesome = [awesome
stringByPaddingToLength:(length + padding)
withString:@"!" startingAtIndex:0];
```

```
        return moreAwesome;
    }

    @end
```

And with this `AwesomeStringMaker` power, you can use the class like this:

```
AwesomeStringMaker *myAwesomeStringMaker =
[[AwesomeStringMaker alloc] init];
[myAwesomeStringMaker setExclamationCount:[NSNumber
numberWithFloat:8.0f]];
[myAwesomeStringMaker setOriginalString:@"typing power"];

NSString *myAwesomeString = [myAwesomeStringMaker awesom-
eString];

// myAwesomeString now contains "TYPING POWER!!!!!!!!"

[myAwesomeStringMaker release];
```

Tip: As you're learning Objective-C, be careful which objects get which message. If you try to send a *–setExclamationCount:* message to an instance of *NSString* instead of to *AwesomeStringMaker*, you'll get a "method not found" warning while compiling your code. At runtime, your application will crash with an exception in *objc_msgSend* because the *NSString* class doesn't know what to do with the message you've sent.

Managing Memory

If you take a look at your class's accessors for *exclamationCount* and *originalString*, you'll see some new methods being called on the instance variables. The *retain, copy,* and *release* messages are extremely important: You use them to manage the memory usage of objects.

Note: Memory is also one of the more complicated things to understand about Objective-C, so don't get frustrated if you don't understand it at first.

As you learned before, thousands of objects are created and deleted every second that you're using your iPhone. And each of those objects uses a chunk of memory, which is a precious resource on a mobile device. Your desktop computer may measure its memory capacity in gigabytes, but your phone only has a few hundred megabytes.

If you're not careful about your memory usage, the iPhone's operating system will shut down your application. If it didn't, the system would eventually grind to a halt, and you'd have to reboot your phone. That's kind of inconvenient when you're expecting an important call.

So how is memory managed with these *–retain, –release,* and *–copy* methods?

Every object in memory maintains a counter. This counter keeps track of how many other objects are using the object. When you want to use an object, you must tell the object, so it can update its counter, and you do so by using the *–retain* message. After you send that message, the object updates its retain count, and that object won't be deleted. If you're ever interested in how many other objects are keeping a reference to that object, you can send it the *–retainCount* message, and it'll return the current value of the counter. When objects are initially allocated, their retain counter is set to *1*.

Now that you know how you keep objects around in memory, you need a way to get rid of them when you're finished using them. That's where the *–release* message comes in. It does the exact opposite of *–retain*; instead of incrementing, it *de*crements the retain count. You're telling the object you don't need it anymore.

As your app runs in the Cocoa Touch environment, the retain count of objects is checked periodically. Objects with retain counts that have reached zero aren't being used by any other objects, so the system can delete them and reclaim their memory.

Now that you're familiar with the mechanics of retain and release, take a look at the accessor in more detail:

```
        if (exclamationCount != newExclamationCount) {
```

This line is a simple performance optimization. If the old object and new object are the same, there's no need to adjust the retain count.

Tip: This code is comparing pointer addresses, which is a quick (and valid) way to check that two objects are the same. In fact, *NSObject*'s *–isEqual* method uses this same method. (For many classes, such as *NSString*, checking for equality is more complicated than checking pointers.)

The first thing to do is tell the old instance variable that you no longer need it by sending the *–release* message:

```
            [exclamationCount release];
```

Now that you've released the instance variable, you have no guarantee that the variable is valid. If you try to send additional messages to *exclamationCount*, your app is likely to crash. So the next step is to retain the new object that is being passed into the accessor. The retain method returns itself, so that value is used to update the instance variable for *exclamationCount*. At this point, it's safe to send messages to this object again:

```
            exclamationCount = [newExclamationCount retain];
```

A variation on the *–retain* message is the *–copy* message. As with retain, you'll have a unique reference to an object that won't go away until you decide that it's OK. The difference is that you'll be holding onto a copy of the original object's data.

Here's a case where *–copy* comes in handy. With the *originalString* instance variable, you don't want to use *–retain,* because the original string could be modified by another object. Remember, if you're sharing a reference with many other objects and one of them changes the string, you'll see the change when you call –*awesomeString.*

You want to have your own unique copy of the string, so you use this instead:

```
            originalString = [newOriginalString copy];
```

If another object modifies the object pointed to by *newOriginalString*, you won't be affected.

Take a *nil* Pill

Sometimes you want to clear out an instance variable completely. You might want to save some memory and remove the object because you're no longer using it. Or you might be modeling some state within your application where the absence of an object is important. You can clear the variable with a special object called *nil.* If a variable referencing an object contains the value *nil,* there's no object.

Nil objects have an interesting behavior that affects how you use them while managing memory. When you send a message to a *nil* object, the message is ignored and a result of *nil* is returned. This technique is commonly called a *nil targeted message* and it looks like this:

```
    NSString *missingString = nil;
    NSString *excitingString = [missingString awesomeString];
    // excitingString has a value of nil, too
```

When you're writing accessors with *–retain, –copy,* and *–release,* this feature lets memory get cleaned up. To see how this works, replace `newExclamationCount` with *nil,* and you have this code:

```
        if (exclamationCount != nil) {
            [exclamationCount release];
            exclamationCount = [nil retain];
        }
```

If the current count isn't already *nil,* the object for `exclamationCount` is released. Then the *nil* result of the retain message is used to set the new count.

Nil targeted messages can also help prevent crashes in code that has failed some kind of initialization. (Objects that fail to initialize properly are assigned *nil* to show that they don't exist.) You can also use *nil* objects to your advantage for those cases where you want to manage a state. An example would be a *middleName* instance variable in a view: If the value of the variable is *nil,* you know that you don't need to display a person's middle name.

Some code will also take advantage of *nil*'s value: zero. In the case of the *exclamationCount*, when it's *nil,* the message for *unsignedIntValue* is ignored and *nil* is returned:

```
    NSInteger padding = [nil unsignedIntValue];
    // padding has a value of zero
```

With a padding of 0, no exclamation points will be used, which seems like the right behavior when there's no object for the instance variable. On the other hand, if your code needs an object but one has not been set, you'll want to use something like this:

```
    if (myObject) {
        [myObject doSomething];
    }
    else {
        NSLog(@"Can't doSomething because myObject is nil!");
    }
```

Again, this code relies on the fact that a *nil* object has a value of zero. If there's a value for the *myObject*, its nonzero value will let the *–doSomething* message be sent.

Tip: *Nil* objects can also be a great source of head scratching. If you forget to initialize a variable, you'll wonder why an object instance isn't doing what you expect when you send it messages. The "aha" moment will come when you go into the debugger, display the object, and see "Cannot access memory at address 0x0". Your object's value is zero.

Autorelease with Ease

Autorelease has nothing to do with letting go of your car. All of this retaining, copying, and releasing is a lot of cumbersome programming when you're dealing with temporary objects. Often, you'll want to keep an object around just long enough to do some work. For example, many objects never exist outside of the scope of a method implementation.

The brilliant and lazy programmers who came up with Objective-C have a work-around called the *–autorelease* method. To see how it works, first look at how the *myAwesomeStringMaker* object is maintained:

```
AwesomeStringMaker *myAwesomeStringMaker =
[[AwesomeStringMaker alloc] init];

// do some stuff with myAwesomeStringMaker

[myAwesomeStringMaker release];
```

This code can be simplified by using the *–autorelease* message:

```
AwesomeStringMaker *myAwesomeStringMaker =
[[[AwesomeStringMaker alloc] init] autorelease];

// do some stuff with myAwesomeStringMaker
```

After the object has been allocated and initialized, it's sent the *–autorelease* message. An object that's been autoreleased is guaranteed to remain in memory for the current method. After that, it will be deleted without your direct intervention.

This phenomenon makes things much easier for you. In this example, *myAwesomeStringMaker* is only needed for a short time, so autorelease does the object cleanup automatically. If you're doing a lot of things with *myAwesomeStringMaker* between allocating the object and the end of the method, it's easy to forget to send the message to *–release.*

Because other programmers are just as smart and lazy as you are, many methods return autoreleased objects. The assumption is that you're probably not going to need that object for a long time, so it's best to let the system do the cleanup.

By convention, only methods that have *init* or *copy* in their name return objects that aren't autoreleased. If you use one of these methods, you've got to pay attention to what's happening with your memory.

WORD TO THE WISE

Memorize the Rules

Longtime Cocoa programmers will cite the following rules, originally written by Don Yacktman, whenever you ask a question about how to manage memory:

- If you allocated, copied, or retained an object, then you're responsible for releasing the object with either *–release* or *–autorelease* when you no longer need the object. If you did not allocate, copy, or retain an object, then you should not release it.
- When you receive an object (as the result of a method call), it will normally remain valid until the end of your method, and the object can be safely returned as a result of your method. If you need the object to live longer than this—for example, if you plan to store it in an instance variable—then you must either *–retain* or *–copy* the object.
- Use *–autorelease* rather than *–release* when you want to return an object but also wish to relinquish ownership of the same. Use *–release* wherever you can, for performance reasons.
- Use *–retain* and *–release* (or *–autorelease*) when you want to prevent an object from being destroyed as a side effect of the operations you're performing.

Properties and Dots

As you get more advanced with your Objective-C coding, you'll find that you start to have a lot more accessors in your classes. They're another case of boring code that the lazy programmer wants to avoid, especially all of the retain and release stuff that's repeated over and over. You can utilize another shortcut called *declared properties*. This language feature provides a simple way for you to specify, implement, and use a class's accessor methods.

Both the class interface and implementation change when you're using properties:

```
@interface AwesomeString : NSString
{
    NSNumber *exclamationCount;
    NSString *originalString;
}
@property (nonatomic, retain) NSNumber *exclamationCount;
@property (nonatomic, copy) NSString *originalString;

- (NSString *)awesomeString;
@end
```

Each property is configured using the attributes in parentheses. The *nonatomic* attribute makes the accessor faster; you only need atomic properties when working across multiple threads. The *retain* attribute causes the instance variable to use the retain/release pattern during assignment. Likewise, the *copy* attribute uses the copy/release pattern.

You just got rid of two lines of code, but wait, there's more!

In the implementation, you can replace the *exclamationCount* and *setExclamationCount* methods with one line of code. And you can replace the *originalString* and *setOriginalString* with a second line of code:

```
@implementation AwesomeString

@synthesize exclamationCount;
@synthesize originalString;

...
```

The synthesize shortcut causes the compiler to generate the code necessary to access your instance variables. The code in the *@implementation* is generated according to the *@property* definition in the *@interface*. You may get tired of typing all those @ symbols, but those *@synthesize* statements saved you from writing 19 lines of code. With a larger class, the savings in lines of code will be even more impressive.

Once you've defined properties for your class, you may want to access them using *dot notation*. This notation lets you access the instance variables without using the square brackets. Finally, some relief for the C++ and Java coders amongst us! For many developers this code will be much more readable:

```
AwesomeStringMaker *myAwesomeStringMaker =
[[[AwesomeStringMaker alloc] init] autorelease];

// use dot notation to set a property's value…
myAwesomeStringMaker.exclamationCount =
[NSNumber numberWithFloat:8.0];
myAwesomeStringMaker.originalString = @"typing power";
NSString *myAwesomeString = [myAwesomeStringMaker
awesomeString];

// or use dot notation to read a property
if ([myAwesomeStringMaker.exclamationCount integerValue]
< 4) {
    // that's not awesome enough
    myAwesomeStringMaker.exclamationCount = [NSNumber
numberWithFloat:8.0];
}
```

A property on the left side of the equal sign will use the setter. The same property on the right side will use the getter.

Methods of Class

Earlier, you might have been a little confused by this code:

```
[NSString stringWithUTF8String:"Even"]
```

With all the talk of sending messages to object instances, now you're not using one. Where the heck is that message going?

To begin answering that question, take a look at the definition in the `NSString` interface:

```
+ (id)stringWithUTF8String:(const char *)bytes;
```

Yep, it's another wacky character courtesy of Objective-C. This time it's a plus sign (+).

On page 45, you saw that methods were defined with a minus sign at the beginning. But those are just *instance* methods. *Class* methods are another kind.

With instance methods, you need a variable that references the object instance before you can send the message. Sometimes this is a limitation in your class design: You'd like to make a method available without an object instance. Class methods are Objective-C's solution to this problem. With these methods, you send messages directly to the class without needing an actual object.

These class methods are often used to create new object instances. The *+stringWithUTF8String:* method is just such a case; sending the message to the class lets it return a new object for you to use in your code.

Note: You may be familiar with using *factory* objects. That's exactly what sending a message to the class does. The *+stringWithUTF8String:* message acted as a factory for a new instance of *NSString*.

So what could make your *AwesomeStringMaker* class more awesome? A class method that returns the *mostAwesomeString*:

```
 @interface AwesomeStringMaker : NSObject
…

+ (NSString *)mostAwesomeString;
@end
```

Your implementation would look like this:

```
@implementation AwesomeStringMaker
…

+ (NSString *)mostAwesomeString {
    return @"CHOCKLOCK!!!!!!!!";
}
@end
```

Now you don't even have to think when you need to make your app a lot more awesome:

```
[label setTitle:[AwesomeStringMaker mostAwesomeString]];
```

Initializing Objects

Your `AwesomeStringMaker` class has a bug. Hard to imagine, isn't it?

```
AwesomeStringMaker *myAwesomeStringMaker =
[[[AwesomeStringMaker alloc] init] autorelease];
myAwesomeStringMaker.originalString = @"typing power";
NSString *myAwesomeString = [myAwesomeStringMaker
awesomeString];
```

The value for *myAwesomeString* is "TYPING POWER". It's missing some awesome exclamation points!!!!! That's because when a new instance of an object is created, all instance data is cleared out. Pointers are set to *nil* values and numbers are set to zero. This means your *exclamationCount* instance variable is zero. And you can't be awesome with zero exclamation points.

This situation illustrates a larger problem: Your objects usually need to be set up before you can send them messages. With Objective-C, there's a method for just that:

```
- (id)init {
    if (self = [super init]) {
        // WHY USE A LITTLE WHEN YOU CAN USE A LOT
        exclamationCount = [[NSNumber numberWithInt:8]
retain];
        originalString = [@"" copy];
    }
    return self;
}
```

That code may look convoluted, but what it does is simple and elegant: After every object is allocated in memory, the new instance is sent the –*init* message. You can choose to ignore the message, but it's more likely that you won't.

Note: You may be wondering why every object gets sent the *–init* message. It's because the default implementation is in *NSObject*, and every object is a descendent of this root class. If you don't implement the method, it will be handled by superclasses, including *NSObject*.

When your object gets the message, it assigns the result of sending *–init* to the parent object (represented by the implicit variable *super*) to the variable *self.* So what's all that about?

Just like your class needed to do some initialization, your parent class may need to do some. You don't know exactly what that setup is, but now is the only opportunity for any classes between you and `NSObject` to get their houses in order.

Once you've given everyone else a chance to get set up, you check the result of the assignment to *self.* It's possible that one of the superclasses wasn't able to initialize itself, so it returned *nil.* If that's the case, you'll skip your initialization and return *nil,* too.

Assuming that everything went fine with your superclasses, you'll get your chance to set the *exclamationCount* to a whopping 8 characters.

Deallocation Location

There's still a pretty serious bug in your code. You went to all the work of retaining, copying, and releasing your accessors. But what happens when the whole object is released? Unless you called each accessor and set a *nil* value, the memory used by instance variables will never be freed.

As you implement classes, it's your responsibility to clean up after yourself as objects of that class are released. The way you do this is with a *–dealloc* method. Just as *–init* let you get things set up, the *–dealloc* method lets you tear things down. Here's what that method should look like for your *AwesomeStringMaker* class:

```
- (void)dealloc {
    [exclamationCount release];
    [originalString release];

    [super dealloc];
}
```

First, both of your instance variables are released so that memory can be reclaimed. Then the *–dealloc* message is forwarded to the superclass so it can do its own cleanup.

Manual override

You may have noticed that *–init* and *–dealloc* were never defined in your *@interface*. That's because it's not necessary; as a descendent of *NSObject*, the method has already been declared. Your own implementations of these methods merely "override" those defined in the superclasses.

A class's documentation will give you guidance about which methods can be overridden. In some cases, it's required: If you create a *UIView* class that draws its own content, you must override the *–drawRect* method to display the view. In other cases, overriding methods provides flexibility and customization. For example, with the *–layoutSubviews* method, an override lets you explicitly control the positioning of interface elements in a way that you can't by using the default mechanisms.

Loops: For Better or For Worse

Looping with variables is probably one of the first programming concepts you learned. You'll be happy to know that Objective-C has *for* loops—and the syntax is better than that of plain C.

You'll learn all about the *NSArray* class in the next chapter, but for the moment all you need to know is that it's a class that manages zero or more objects. When you have more than one of something, it's likely that you'll want to loop over the contents at some point.

You could use the good old *for* loop in standard C:

```
NSArray *myArray = [NSArray arrayWithObjects:@"THIS",
@"IS", @"AWESOME!!!", nil];
NSUInteger count = [array count];
NSUInteger i;
for (i = 0; i < count; i++) {
    NSString *element = [myArray objectAtIndex:i];
    if ([element isAwesomeString]) {
        NSLog(@"CONGRATULATIONS!!!!!!!!!");
    }
}
```

But Objective-C has a construct called *fast enumeration* that saves a bunch of typing:

```
NSArray *myArray = [NSArray arrayWithObjects:@"THIS",
@"is!", @"AWESOME!!!!", nil];
for (NSString *element in myArray) {
    if ([element isAwesomeString]) {
        NSLog(@"CONGRATULATIONS!!!!!!!!!");
    }
}
```

More importantly, this form of enumeration is optimized by the runtime and removes the possibility for bugs with loop invariants.

Another example of smart-but-lazy code that's simpler, safer, and faster.

Your Exceptional Code

As a developer, you know that things don't always go the way you plan. Code that doesn't work like you expect is a never-ending source of joy in your life. Fortunately, Objective-C supplies compiler directives that help you deal with exceptions in blocks of code:

- Use *@try* for code that may throw an exception.
- *@catch()* lets you specify code that gets run when exceptions occur. You can use more than one block if you need to handle different types of exceptions.
- You can use *@finally* if you have code that needs to run whether or not an exception occurred.

Here's an example that uses all three:

```
NSArray *myArray =  [NSArray array]; // an array with no
elements
@try {
    // the array is empty, so this will fail:
    [myArray objectAtIndex:0];
}
@catch (NSException *exception)
{
    NSLog(@"name = %@, reason = %@", [exception name],
[exception reason]);
}
@finally
{
    NSLog(@"Glad that's over with...");
}
```

which will generate the following output in the console log:

```
name = NSRangeException, reason = *** -[NSCFArray
objectAtIndex:]: index (0) beyond bounds (0)
Glad that's over with...
```

If you hadn't caught the exception, your application would have quit and sent the user abruptly to the iPhone's home screen. Catching exceptions is a good thing.

When your code needs to generate an exception, use the `NSException` class. Suppose you come across a string that's not awesome enough; you could raise an exception like this:

```
[[NSException exceptionWithName:@"AwesomeException"
reason:@"Not awesome enough" userInfo:nil] raise];
```

You can use the `userInfo` parameter if you need to supply some additional information along with the exception (the offending string, for example).

Tip: When debugging, it's often helpful to set a breakpoint just before the exception is thrown. This way, you can see the stack trace at the point where the error occurred. Add a breakpoint at *objc_exception_throw*, and the debugger will show you everything that led up to the problem.

Learn by Crashing

It may seem odd that a chapter that's explaining a programming language includes a section on using the debugger. But think about the last time you learned a new language: You had plenty of bugs, because everything was new and you were doing things a more experienced developer would avoid. This section will help you when this happens.

Xcode has a powerful debugger built-in; it's based on *gdb* from the GNU development tools. Like with the Objective-C compiler, some extensions to the debugger make it compatible with objects and the runtime architecture.

When you run your code, you can do so with or without breakpoints enabled. Xcode's toolbar has a button that toggles your breakpoints: You can also choose Run→Debug – Breakpoints On (or press ⌘-Option-Y) to launch your app with them enabled.

Once you have breakpoints enabled, you can set new ones in a couple of ways:

- When you click in the left gutter of your source code, a blue arrow will appear (Figure 2-1). The arrow signifies that a breakpoint is set and that your application will stop when it's reached. If you click the arrow again, it becomes dimmed and disabled. You can also Control- or right-click the arrow to get a context menu with other options.

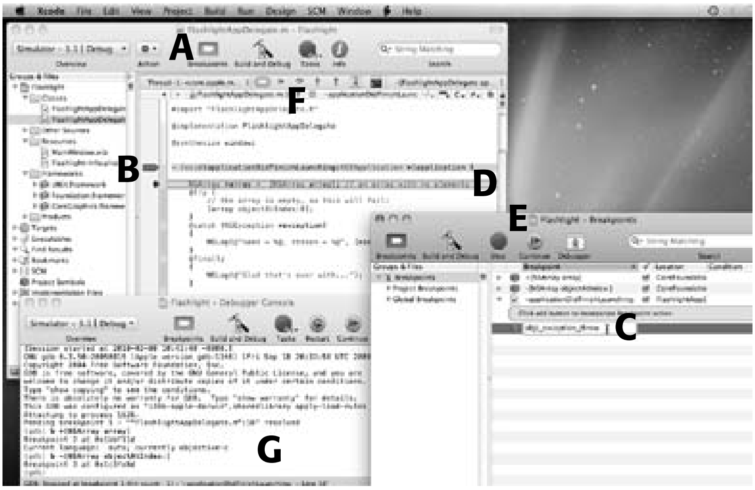


Figure 2-1: *The Xcode debugger in action. You can enable or disable breakpoints using the toolbar icon (A). To set a breakpoint in your source code, click in the left gutter until you see a blue arrow (B). The Breakpoints window shows all the breakpoints that have been defined, and you can add new ones to the end of the list (C). When a breakpoint is reached, the line of source code will be highlighted (D), and you can easily navigate the stack back-trace with a pop-up menu (E). Controls are also available to step through code (F). More advanced operations can be performed in the debugger console (G).*

- All of the current breakpoints are displayed when you choose Run→Show→Breakpoints (or press ⌘-Option-B). You can add a breakpoint to the list that's displayed by double-clicking the last line and entering the symbolic name. You'll need to use this method when you want to set a breakpoint where you don't have the source code (for example, in the Cocoa Touch frameworks). You can also remove a breakpoint in this window by selecting it and pressing Delete.

When setting breakpoints manually, you can use a function name such as *objc_exception_throw,* and the debugger will halt before it's executed. But since you're new to Objective-C, it's more likely that you'll want to break at a class method. Here's some special syntax to set these types of breakpoints:

- To stop at an instance method, use the form *-[ClassName methodName:]*. For example, to stop in `NSArray`'s *–objectAtIndex:* method, you'd use *–[NSArray objectAtIndex:]*. (Make sure to remember the semicolons when specifying the name, or the breakpoint won't fire.)
- For a class method, the syntax is similar: Just use a plus sign (+) instead of a minus sign (–) before the first bracket. To stop at the *+array* method in *NSArray*, you use *+[NSArray array]*.

Once you've stopped at a breakpoint, you can examine your variables by hovering over your source code with the mouse pointer. In many cases, you're going to want to use the *gdb* command line to examine the state of your instance variables and to send them messages. Here are a few important tricks:

- *po,* the *print object* command, displays the description for an object. In many cases, this will contain information that helps you figure out what's going on. Using the *myArray* variable from the code on page 68 as an example:

```
(gdb) po myArray
<NSCFArray 0x3b0e850>(
)
```

The print object command for an array normally shows all the elements in the array. In this case, none are shown, so you know that accessing the first one is going to be a problem.

- *p,* the *print* command, displays the value of an intrinsic type. Like the previous command, it's smart about the type of data you're examining: Structures and enumerations will be used as necessary.
- Methods can be called from both print commands. For example, if you've discovered a problem with an object not being retained correctly, you can fix the problem temporarily with:

```
(gdb) po [myArray retain]
<NSCFArray 0x3b0e850>(
)
```

  The above example returns an object, so the *po* command is used. If a method returns an intrinsic type, you need to use the *p* command, and let the debugger know what type to expect. For example, you specify that an integer result is provided when you query the array for the number of entries:

```
(gdb) p (int)[myArray count]
$1 = 0
```

- When you see *address 0x0* in the console, it's a *nil* object (page 56). For example, if you ask the empty array for the last object, there is none, so you'll get back a *nil* result:

```
(gdb) po [myArray lastObject]
Cannot access memory at address 0x0
```

- The *p* command can also be used to assign a new value to a variable. To fix the exception caused by an empty array, you can assign a new one that has a single string in it:

```
(gdb) p myArray=[NSArray arrayWithObject:@"MY CODE IS
PERFECT"]
$2 = (NSArray *) 0x3b0ee40
(gdb) po myArray
<NSCFArray 0x3b0ee40>(
MY CODE IS PERFECT
)
```

- The *b* command can be used to set a breakpoint from the *gdb* prompt. For example, to stop your application before retrieving an object at an array index, use:

```
(gdb) b -[NSArray objectAtIndex:]
Breakpoint 1 at 0xf135h90d
```

As you're probably aware, knowing your way around the debugger makes you a much more effective developer. To learn more, start by searching for the *Xcode Debugging Guide.* For more advanced information, take a look at *Debugging with GDB.* Both documents are available in the documentation viewer described on page 80.

Selector Projector

Because Objective-C is a dynamic language, it resolves methods at runtime, not at compile time. This setup provides a subtle but powerful mechanism: You can pass methods as parameters to other methods.

Selectors are special variables that identify which method implementation to use for an object instance. The Objective-C runtime uses internal data structures to match a message's selector to the code that gets executed.

Tip: If you've used other languages, you might want to think of a selector as a dynamic function pointer. The selector's name automatically points to the right function and adapts to whichever class it's used with.

This behavior lets you implement polymorphism in Objective-C. If you have three classes named *Circle, Triangle,* and *Square,* you can send a selector for a *draw* method to instances of each of these classes, resulting in different implementations for drawing the shape being executed.

Every selector is defined with a type of *SEL*. You can assign the selector variable with one of two methods. At compile time, you can use a compiler directive to create the value:

```
SEL mySelector = @selector(moreAwesomeThanEver);
```

At runtime, you can use a function that takes a single *NSString* parameter and looks up the selector:

```
SEL mySelector = NSSelectorFromString
(“moreAwesomeThanEver”);
```

The *mySelector* variable now contains a shortcut for the *–moreAwesomeThanEver* message. And here's where it gets interesting: You can use that variable to execute the method. So instead of sending the message like you have been throughout this chapter:

```
[myObject moreAwesomeThanEver];
```

you can use the variable and achieve the same result:

```
[myObject performSelector:mySelector];
```

Now you could do something crazy like take user input, convert it into an *NSString*, and then call methods based on what the person typed. Luckily, the designers of Objective-C didn't implement selectors so you could be crazy. They did it so you could be lazy and could pass references to snippets of code that help the frameworks do the hard work.

Selector variables, and the ability to pass them around, play a central role in the Cocoa Touch frameworks that you'll learn about in the next chapter. Many of these features fall into the advanced category, but here are just a few of the cool things you can do with messages stored in variables:

- You can implement the delegation and target/action patterns described in the next chapter using selectors and the *–performSelector* method described above.
- Arrays can use selectors to control sorting behavior. You can also use selectors when you want to send the same message to every item in the collection.

- If you want to keep track of when a view is animating, you can specify a callback method with a selector.
- You can configure timers with a selector that runs code after a delay or at repeated intervals.
- Objects register for system-wide notifications by specifying a method via a selector.
- You can use selectors to specify which method in an object initializes a new thread.

The number of colons in a selector's name is *really* important. While *@selector(moreAwesomeThanEver)* and *@selector(more AwesomeThanEver:)* look similar, they have different method signatures. The first case is a method that takes no parameters, and the second case takes one parameter because it has a colon at the end.

If you've ever spent hours tracking down a stray semicolon in C code, finding a missing colon in a selector is the same sort of thing. Your eyes fool you into thinking that the code looks right, even when the runtime knows it's not.

Tip: When you see crashes in *objc_msgSend,* it's often the Objective-C runtime telling you that it can't find a method implementation. If you type your *@selector* incorrectly, that will be the cause. Make sure that the object has an implementation for the method being used.

Show Your *id*

Sometimes you need to refer to an object without knowing its class. It's convenient to reference an object just by its memory address without any type information.

You've already seen one of these anonymous pointers on page 64:

```
    - (id)init {
```

Objective-C defines *id* as a pointer to an object data structure. (It's much like a *void* pointer in C, except that the pointer is only used to reference objects.)

So why does the *–init* method return this generic type? Because you have no guarantee what class of object the superclass will return to you:

```
        if (self = [super init]) {
```

During initialization, objects are in a state of flux, so their class is purposely ignored and considered subject to change. Using a type of *id* expresses the object's lack of identity.

Even though *id* is nothing more than a pointer to a block of memory, you should never use a *void* pointer in its place. The compiler is able to perform additional checks and optimize code because the *id* lets it know that the variable points to an object.

The *id* type is also used as a way to avoid casting between classes. Some classes need to manage objects without knowing anything about the type of data. A good example of this is collections and other object containers. If objects in a collection were specified with a pointer to an *NSObject*, you'd have to cast to a subclass each time you retrieved an object from the collection. For lazy programmers, that's a lot of work.

Note: An *id* also solves the situation with overridden methods in a subclass that return a different type than the original class. By using a generic type, both the original class and any of its subclasses can share a common method signature and return the appropriate class.

Again, the *NSArray* class will be explored in the next chapter, but you already know that arrays are only useful if you can access the elements in the list. The method for doing this is:

```
- (id)objectAtIndex:(NSUInteger)index;
```

So, given an integer index, an *id* value is returned. The same is true when you add elements to the array: Combining these methods gives you a way to store any kind of object you'd like in the array.

Of course, your code usually knows what it put in the array, so it's common to do an implicit cast as you read the array element:

```
NSString *element = [myArray objectAtIndex:i];
```

This code lets the compiler check any code that accesses the element object.

Finally, there's one special *id* value that you've already seen mentioned several times in this chapter—*nil.* As with pointers in C, you'll occasionally want to represent the absence of an object. Using *nil* to represent a null object lets you do this.

Both *nil* in Objective-C and *NULL* in C are defined as *void* pointers with a value of zero. For example:

```
#define NULL ((void *)0)

#define __DARWIN_NULL ((void *)0)
#define nil __DARWIN_NULL
```

Semantically, however, you should use *nil* when dealing with pointers to objects and *NULL* for all other types of pointers. Technically, there's nothing wrong with this code:

```
NSString *myString = NULL; // PLEASE BE AWESOME AND USE
nil INSTEAD!!!
```

It will, however, cause a nervous twitch and elevated blood pressure in longtime Objective-C programmers. Which is fine until you ask for their help in an online forum.

UNDER THE HOOD

It Really Is Just C

On page 34, you saw that Objective-C is just C with a little extra syntax and a runtime.

Now that you've learned a little bit about the language, you might want to see how Objective-C performs its magic. This is advanced stuff, but if you're the kind of developer who likes to look under the hood, you'll love this box.

First, start with some standard Objective-C code from an earlier example:

```
NSString *myString = @"typing power";
NSString *myResult = [myString awesomeString];
NSLog(@"myResult = %@", myResult);
```

You can do exactly the same thing with this code:

```
#import <objc/objc-runtime.h>
id myString = @"typing power";
SEL mySelector = @selector(awesomeString);
IMP myImp = class_getMethodImplementation(object_getClass(myString),
mySelector);
id myResult = myImp(myString, mySelector);
NSLog(@"myResult = %@", myResult);
```

—Continued

What you've just done is replicate the work of *objc_msgSend:,* the code that's used by Objective-C to send messages to objects. It looks a lot like plain old C code, doesn't it?

The first step is to import the *objc-runtime.h* header file. That's where all the runtime definitions are stored.

You begin by creating a selector for the *–awesomeString* method. That's just a pointer to an internal structure:

```
typedef struct objc_selector *SEL;
```

That selector is then passed to the runtime function *class_getMethod-Implementation()* that looks up a function using the object's class and the selector. All methods conform to this function definition:

```
typedef id (*IMP)(id, SEL, ...);
```

Once you have the function pointer to the class's implementation in the `myImp` variable, you can call it with the object and selector. This example doesn't have any additional parameters, but since *IMP* is defined with a variable argument list, they can be passed in as well.

When the compiler came across your method implementation:

```
- (NSString *)awesomeString;
```

it generated a C function that looks something like this:

```
id _i_NSString_AwesomeMethods_awesomeString(id self, SEL _cmd);
```

The function name is mangled using the method type (instance or class), the class name, category, and method name. The function's arguments explain how the hidden *self* argument is passed. It also shows that there's another hidden argument called *_cmd* that your method implementation can use if it needs to know its selector.

And all this stuff happens automatically when you type those powerful square brackets.

Where to Go from Here

Now that you've gotten a taste of Objective-C, you may find that you want to know more about the language.

Sometimes, you do get something for nothing; *The Objective-C 2.0 Programming Language* book is a great example. Apple has some of the best technical writers in the business, and it shows in this book that describes the programming language. All of the language's features are described in detail along with pertinent examples. The book's glossary and index are especially helpful as you encounter new terminology and concepts. There's a link to this online book on this book's Missing CD page at *www.missingmanuals.com/cds*. (Or just do a web search for *the objective-c 2.0 programming language*. You'll find a link to the book on Apple's site.)

Another great resource is right inside this book—Appendix A. Here you'll find additional resources for each of the topics covered in this book. Each is listed along with a URL and a short review. If there's an area where you feel you need more help, head for the appendix.

Developer Documentation

It may seem odd that you've gone through an entire chapter describing a programming language, and the topic of developer documentation hasn't been mentioned. It's because there's a lot of it and it's really good. To the point of it being a distraction, in fact.

Since there's no way this book can explain every line of code you'll encounter, now's a good time to introduce the system documentation shown in Figure 2-2.

While you're working in Xcode, you can call up this documentation at any time from the Help→Developer Documentation menu. Once the window appears, you can search for anything by typing in the box at upper right: class names (such as *NSString*), method names (such as *release*), or concepts (such as *override*).

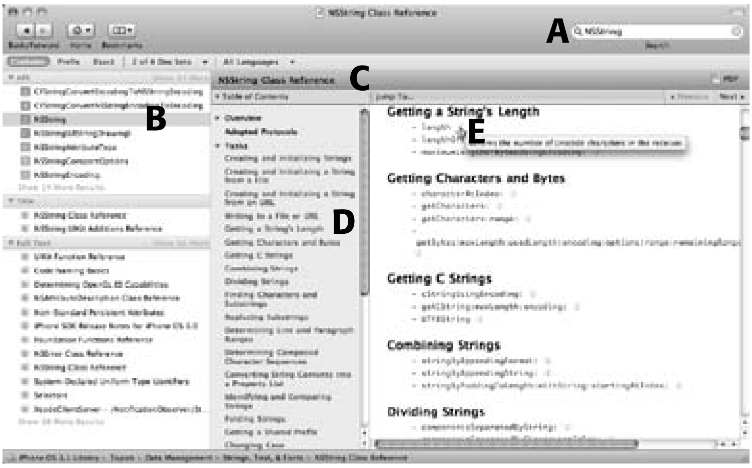


Figure 2-2: *The Xcode documentation viewer. You can search using the box at upper right (A). The results are listed on the left side of the window (B). Select one of the results, and the documentation page is displayed at right (C). From there, you can open the Table of Contents to quickly navigate through the page by clicking the links (D). For example, clicking the –length method name will take you to detailed documentation, or you can hover your mouse pointer over the info icon to get a quick description (E).*

Once your search results appear, you can drill down using various navigational elements:

Results list

The results for your search are displayed on the left side of the documentation window. You can control how much information is displayed in the API list by selecting how your search term will be used to match documentation pages. When Contains is selected, *NSString* will match *CFStringConvertEncodingToNSStringEncoding* because it's a part of the API name. If Prefix is selected, *NSString(UIStringDrawing)* will match because it begins with the search text. The Exact setting would only show the page for *NSString*.

The results are broken down into three sections:

- **API.** This section shows symbol name matches. Each match is prefixed with a colored letter. A purple *C* tells you the symbol is a class, a blue *M* is a method, an orange *T* is a type definition, and so on.
- **Title.** Each of the entries under this section is a match for the document page's title.
- **Full Text.** The last section shows matches in the full text of the document page. These results can be handy to find what you're looking for in a larger context. If you're more interested in how you use a class, these results will show where it appears within a guide, release notes, or sample code.

Documentation page view

When you select an item from the results list, the page is displayed in the right side of the window. For some classes, this documentation page can be quite long, so here are some tricks to help navigate it quickly:

- **Table of Contents.** You can open the page's table of contents to quickly navigate through a page with many methods. And when you're first starting out, the "Tasks" section of the class documentation can be a helpful guide. For example, if you're looking for ways to create and initialize strings, there's a subsection for that under Tasks for *NSString*.
- **Companion Guides.** Many of the core classes in Cocoa have companion guides. These documents provide a high-level view of how to use the class, along with samples. If you're just starting out with *NSStrings*, clicking the "String Programming Guide for Cocoa" will get you up to speed as quickly as possible.

- **Related Source Code.** Sometimes a picture is worth a thousand words; for a developer, sample code is the prettiest picture of all. If you're struggling with how to use *NSString* in a list of data, clicking *TableViewSuite* will show you how it's done in working source code.
- **Jump To…** If you're looking for a specific method in a class, this pop-up menu provides quick access to various sections and subsections on the page.
- **Use Find.** When you press ⌘-F, a new search field appears above the document page. Any text you type in this field shows up as a match in the documentation. You can click the arrow buttons (or ⌘-G and ⌘-Shift-G) to navigate through the matches.

Type less, learn more

The built-in documentation gives you easy ways to access it; don't miss these great tricks. Even using copy and paste, entering class and method names into the documentation search field gets tedious quick. The developers who created Xcode are just as lazy as you are, so they came up with some better ways to navigate through this sea of information.

Context menu

If you see a symbol in some source code that you don't understand, you can select the text and then Control-click to reveal a shortcut menu. This menu contains a lot of useful items, but toward the end you'll see two of particular interest. Clicking "Find Text in Documentation" brings up the documentation window with search results for the selected text. No typing required!

Another helpful item in that menu is "Jump to Definition". This command uses the selected text to look for the code where the symbol is defined. For a class like *NSString* where you don't have the source code, you'll be shown an *@interface* definition. If the class is one you've written, you'll be shown the *@implementation* definition.

Clicks ahoy!

But it gets even better. You can access the documentation by using just your keyboard and two clicks of the mouse button. Option-double-click a symbol name, and a small window appears that displays a short description of the class or method, a declaration that includes parameter types, and links to sample code and related documents. You'll also see the date when the API first became available.

Tip: If you want to skip this preview window and jump right to the documentation window, Option-⌘-double-click the symbol name.

To find a definition, ⌘-double-click the name. If you get multiple matches, which is common with method names, you'll be shown a list of choices before the *@interface* or *@implementation* is shown.

If you're like most iPhone developers, you'll soon be using the mouse and these two keys without even thinking.

Learn To Be Lazy

There's that *L* word again. But the fact is, with Cocoa, a lot of great code has already been written for you. When you're starting out, it's natural to have the urge to write something yourself. Especially when it's a new language and you want to start exercising your new skills. When you feel that urge, take a moment to open the documentation and do a little research.

If you find something that's close to what you need, try to use a category to extend an existing class. If that won't work, try to create a subclass and add new functionality. When all else fails, implement your own class from scratch.

And with that advice it's now time to head to the next chapter, where you'll start to learn about some of this great code you now have at your disposal.

Cocoa Touch: Putting Objective-C to Work

Now that you've gotten a taste of Objective-C, it's time to put the language to work. If you think of Objective-C as the glue for building applications, you're now going to explore the building blocks—parts of the Cocoa Touch frameworks—that get pieced together with your new adhesive.

Before you start gluing the components of the Cocoa Touch frameworks together, you'll need to learn a bit about the architecture and design patterns recommended by Apple's engineers. Just as knowing how the correct building materials work together helps to build a house, a good working knowledge of the frameworks lets you build more robust applications and save time.

You'll also learn where to look for more information about Cocoa Touch. Plenty of resources on the Web and in your favorite bookstore can help you improve your iPhone development skills.

Get in Cocoa Touch

The Cocoa Touch frameworks are extensive, and hundreds of classes are at your disposal. Learning how to use this rich collection of components takes time. The best way to approach this Herculean task is to learn the common design patterns and functions implemented by the frameworks.

In the following sections, you'll see how Cocoa Touch uses models, views, and controllers (MVC) to implement your application. You'll also see how to use a target-action design pattern to hook the visual components to the code that does the actual work. In the process, you'll see how delegation is a powerful mechanism for leveraging code from the system frameworks. Of course, you'll also learn about managing data objects and collections, along with some of the more important design patterns.

BEHIND THE SCENES

What's in a Name?

You've probably heard the name "Cocoa Touch" used to describe the programming environment on the iPhone. But what is it?

Strictly speaking, Cocoa Touch is just two frameworks that provide the most important building blocks for your app:

- **Foundation.** This framework gives you the main building blocks. Within this framework, you'll find classes that manage data (such as *NSString*, *NSNumber*, and *NSDate*), that read and write information (*NSFileManager*, *NSUserDefaults*), that communicate with the network (*NSURLConnection*), and much more.
- **UI Kit.** Every application will have windows, views, buttons, and other interface elements. The *UIKit* framework provides the pieces that let users interact with your creation. You see these classes in the Interface Builder library palette.

—Continued

Developers sometimes use the term “Cocoa Touch” as an all-encompassing name for other development technologies that work alongside base frameworks to create an app. You’ll find yourself using many other supporting frameworks as you build your application. The most popular ones are:

- **Core Graphics.** A C-based API for drawing graphics (using the Quartz rendering engine). This low-level framework provides functions for drawing vector paths and bitmaps, 2-D coordinate transforms and masking, color and image management, and much more.
- **OpenGL ES.** Another C-based interface for accelerated rendering of 2-D and 3-D graphics. The implementation conforms to the OpenGL ES 1.1 and 2.0 specifications.
- **Core Animation.** This Objective-C API provides classes that allow sophisticated compositing and animation of a 2-D image hierarchy based on layers. This high-level framework greatly simplifies your code and improves the user experience. A nice side benefit is that it improves performance by leveraging low-level APIs such as OpenGL.
- **Core Data.** This sophisticated framework manages a graph of objects with transparent persistence in XML or SQLite files. Xcode provides tools to describe the objects and the relationships between them. The tools fetch objects using predicates and sort them using descriptors.
- **Core Audio.** A collection of frameworks that let audio be played, recorded, processed, and converted.

You can learn more about these building blocks by searching for their names in the documentation viewer.

The Big Three: Models, Views, Controllers

Cocoa Touch gives you hundreds of object classes to wrap your head around. Fortunately, most of these classes fall into three categories. And the objects in these categories interact in a simple and well-defined way.

Every iPhone application uses a simple Model-View-Controller design pattern. And since developers are just as lazy with writing as they are with coding, you'll often find this referred to as "MVC."

Note: If you've used other languages and development environments, you'll be very happy to know that MVC in Cocoa is no different than the ones you're used to. The objects change, but the concepts don't.

To see the simplicity of this design, take a look at Figure 3-1.

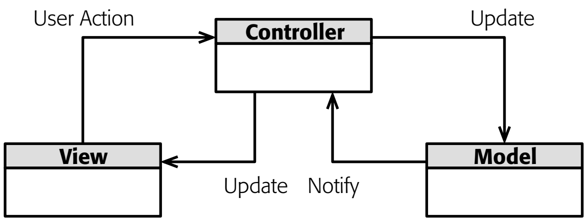


Figure 3-1: *This graphic shows how models, views, and controllers work together, as described on the following pages.*

That's all there is to an iPhone app. And things get even better when you realize that views and controllers in the frameworks already do most of the hard work. Even the creation of model objects is simplified with the help of Cocoa Touch.

So what are the roles of each of these object types? Read on.

Views

You know all those buttons, scrolling lists, web browsers, and everything else that appears on your iPhone screen? Those are all *views.* Views know how to present your application's data. Some views also know how to react to user input. A *UIButton* view responds to a user's touch. A *UITextField* view takes input from the virtual keyboard.

In many applications, you'll create your own views for displaying specific data. For example, there's no standard widget for displaying stock graphs, so if you want to do that, you'll have to come up with your own solution.

Likewise, many designers and developers want to customize the look of their application. Whether your motivation is product branding or just wanting to stand out in a crowd, creating a unique look for your app involves building new views based on standard *UIView* and *UIControl* classes.

Creating views isn't as hard as it sounds. Using subclasses of existing views, you have to do little more than customized drawing. And if you're building composite views, which combine several views into a single view, your only work is managing the layout of the subordinate views.

Models

Models are your application's heart and soul because they're responsible for managing its data. Unlike views, models know nothing about the actions the user is performing or what they're seeing on the display. A model's only function is to manipulate and process the user's data within the application. Models often implement internal logic that provides these basic behaviors.

For example, whenever you use the built-in Contacts application, you're working with model objects that represent the people in your address book. If you update your application's settings, you're modifying another kind of model object. An application that downloads stock data from the Internet would use a model object to store the price history.

Some model objects work across multiple applications: The contacts and user defaults (settings) databases are good examples. Other models, such as those used by a stock application, are application specific.

Many model objects get stored permanently in files or databases. This setup is called *object persistence* and lets you recreate the state of your application across multiple launches. Even if you don't persist your objects, models can be very helpful when dealing with in-memory data structures.

When you're building your own models, you'll often use Cocoa Touch classes in the implementation. For example, classes like *NSArray* and *NSDictionary* let you store data in ordered lists or hash tables. You'll also find that classes like *NSURLConnection* and *NSData* help you retrieve and store data from a network.

Finally, you can use the Core Data framework to store and retrieve objects in an SQLite database.

Controllers

Controllers are a little more complicated. They act as an intermediary between the view objects and the model objects. When a value changes in a model, the controller is responsible for updating the view. Likewise, the controller knows when there's some user input and can update the model data accordingly.

Your controllers let you transfer information between models and views. In Figure 3-1, you can see that the controller is the only block that includes all the arrows (messages). Like a conductor in an orchestra, it's the one in charge.

To give you an idea of how this works, imagine an application that plots stock prices:

1. **The user clicks a button (view) to refresh the graph. This action is sent to the controller.**
2. **The controller, in turn, tells the model to load new stock data.**
3. **The model opens a network connection and begins downloading data.**
4. **After the stock data is loaded, the model notifies the controller that new data has arrived.**
5. **The controller passes the new data onto the view, and the user sees the results.**

Note: The models and views don't communicate directly because if they did, your application would get much more complicated as you add more objects. You'd have problems when multiple views share the data from one model, or a single view updates multiple models. Small changes in one class would ripple into others, classes would be increasingly dependent on each other, and you'd have a hard time reusing code. Developers refer to that as a *tightly coupled design,* and it's a good idea to avoid it.

Controller objects are often responsible for setup tasks. The models and views must be loaded and initialized at some point, and the central role of the controller makes it a natural candidate for this work.

Cocoa Touch gives you specialized view controller classes to handle this separation of responsibilities. As you develop your iPhone app, you'll find that much of the code ends up residing in these controller classes.

In particular, view controllers are used heavily in navigation. Some examples are:

- When you tap on a single row in a list and a new view slides into place, you've just created a new view controller. Likewise, when you tap the Back button, the current controller is discarded and replaced with the previous one.
- If you tap the button to compose a new mail message, a new modal view controller is created; its view slides up from the bottom of the screen and is displayed until you tap Cancel or Send.
- When you tap on the Info icon in the Weather app, a new view controller is flipped into place and lets you enter a new city. When you tap Done, that view is replaced by the main view.

The *UIViewController* class and several subclasses perform these basic functions. If you must master one class before any others, this one is it.

Value Objects

When you're building your own model, view, and controller classes, you'll often have to choose how to store data within those classes. Since Objective-C is built on top of C, all the primitive data types are available. The question is, then, whether to use integers, floating-point numbers, and character arrays or to use a class that wraps those same primitive values in an object.

Let's Get Primitive

There's no harm in using a primitive type in your object's instance variables. In fact, it can often make things much easier for your implementation, because you don't have to worry about retaining and releasing objects. For example, if you're just keeping track of a counter, you don't need to add the overhead of an object.

Tip: One of the most important counters in Cocoa, the *retainCount* in *NSObject*, is defined as an unsigned integer (*NSUInteger*). Although the keywords *NSInteger* and *NSUInteger* look like they could be class names, they're really just type definitions used throughout Cocoa Touch.

```
typedef int NSInteger;
typedef unsigned int NSUInteger;
```

Other examples of commonly used type definitions are *NSRange*, a structure that defines a range of data, and *NSTime Interval*, a floating-point value that represents a period of time.

Instance variables that use primitive types are defined a little bit differently than instance variables of objects. Since there's no retain or copying involved, they're defined with the *assign* attribute for the *@property*. For example, if you had decided to use an *NSUInteger* instead of an `NSNumber` in the last chapter's *AwesomeString* class, you'd have an interface that looks like this:

```
@interface AwesomeString : NSString
{
    NSUInteger exclamationCount;
    NSString *originalString;
}
@property (nonatomic, assign) NSUInteger exclamation-
Count;
@property (nonatomic, copy) NSString *originalString;

- (NSString *)awesomeString;
@end
```

In the implementation the code in the *–init* method changes the initial assignment from a value object:

```
exclamationCount = [[NSNumber numberWithInt:8] retain];
```

You can simplify the code by assigning a primitive value:

```
exclamationCount = 8;
```

In the *–dealloc* method, you also don't need to release the *exclamation Count*. Using a primitive value has made your code much simpler and doesn't sacrifice any functionality. Until written languages start allowing you to put 8.5 exclamation points at the end of a sentence, this code will work great!

Objectified

Now that you've seen an example of value objects not being necessary, why would you ever want to use them? For two primary reasons:

- You can store values represented as objects in a collection. Collections make it easy to manage a group of objects. Methods are available to sort the objects, look up instances, and to filter data according to a pattern. If you're using primitive values, you'll have to implement this data management code yourself.
- The object's class provides a lot of functionality for manipulating the value. This gives you more flexibility and will save time in your implementation.

What are these value objects? Good question!

NSString

As you saw in the last chapter, this class is one of the fundamental classes in Cocoa. Many other classes use strings represented as objects. You may be tempted to use plain C strings in your implementation, and it's indeed possible to do so, but you're going to be swimming upstream. You'll end up doing a lot of unnecessary conversion as you use primitive strings with Cocoa.

Additionally, you'd be missing out on some important functions provided by the NSString class:

- Full Unicode support and conversion between string encodings (*–dataUsingEncoding*).
- Reading text and its encoding from a file (*–stringWithContents OfFile:usedEncoding:error:*).
- Splitting (*–componentsSeparatedByString:*) and joining (*–componentsJoinedByString:*) strings.
- Escaping strings so they can be put in a URL (*–stringByAdding PercentEscapesUsingEncoding:* and *–stringByReplacingPercent EscapesUsingEncoding:*).
- Substring searches (*–rangeOfString:*) and getting the number of Unicode characters (*–length*).
- Converting strings into numbers (*–boolValue, –integerValue, –floatValue, –doubleValue*).
- Case conversions (*–capitalizedString, –lowercaseString,* and the best of all, *–uppercaseString*).
- String formatting (*–stringWithFormat:*) and localization (*NSLocalizedString*).
- String comparison using the user's current language settings (*–localizedCompare:*) and with many options, including ignoring diacritical marks (so "ö" is equivalent to "o") and sorting by numeric value (so "CHOCK9.TXT" appears in a list before "CHOCK9000.TXT").

NSNumber

You'll want to use this value object when you're doing value conversions. If you create a number object with a floating-point value, and then ask the object for an unsigned character value (*–unsigned CharValue*), an internal conversion will be performed without any loss of information in the original object.

If you're doing currency-based calculations in your application, you can use the *NSDecimalNumber* subclass for maximum accuracy. Since the subclass lets you use 38 significant digits and an exponent in the range of –128 to 127, you don't need to worry about rounding errors and other data loss inherent in floating-point calculations.

NSDate

A date object provides some basic functions for manipulating instants in time. You can compare which date is earlier (*–earlier Date:*) and how many seconds are between two dates (*–timeInterval SinceDate:*). *NSDate* really shines when it's used alongside *NSCalendar*; if you've ever tried to compute the number of months and days between two instants in time while taking things like time zones and leap years into account, you'll appreciate this work being done for you.

NSData

As a value object for unstructured streams of bytes, *NSData* provides the mechanisms you need to manage a buffer of data. Data objects often need to be stored on disk, so methods to read (*+dataWithContentsOfFile:*) and write (*–writeToFile:atomically:*) are provided.

These chunks of data are often used to create instances of other objects. For example, an *NSString* can be created with an *NSData* object. Likewise, you can create an image using the *UIImage* object's *–initWithData:* method.

NSNull

This object serves only one purpose: to represent null values. Collections don't allow *nil* objects, so you can use *NSNull* when you need to put an empty value in an array, dictionary (hash table), or set.

NSValue

Sometimes you need to create objects that closely mirror a primitive type or data structure. This situation is most likely if you have some legacy or third-party data and need to add it to a collection.

Whenever this need arises, look at the *NSValue* class. It can wrap any variable type that's valid in Objective-C as a value object. Pretend you have a legacy system that defines a bizarre data structure called *Chockitude*:

```
typedef struct {
    unsigned char palmCount;
    int isFleshy;
    float height; // in inches
} Chockitude;
```

If you need to wrap this data in an object, it's simple:

```
Chockitude ch;
ch.palmCount = 2;
ch.isFleshy = true;
ch.height = 79.25;
NSValue *value = [NSValue valueWithBytes:&ch
objCType:@encode(Chockitude)];
```

Whenever you need to retrieve the contents of the value object, you'd write this code:

```
Chockitude theChockitude;
[value getValue:&theChockitude];
```

```
NSLog(@”palmCount = %d, isFleshy = %s, height = %f",
theChockitude.palmCount, (theChockitude.isFleshy ? "YES
OF COURSE" : "no"), theChockitude.height);

// this is displayed in the console log:
// palmCount = 2, isFleshy = YES OF COURSE, height =
79.250000
```

Another use for `NSValue` objects is with points and other geometry on the iPhone's display. If you wanted to keep track of a series of taps on the screen, you'd probably want to use a collection of `CGPoint` structures (that contain the x and y). You can use an `NSValue` category that takes these common structures and encodes them into objects: *+valueWithCGPoint:* and *+valueWithCGRect:* are used to wrap points and rectangles used by the graphics framework.

Collections

When you're dealing with objects, you're often dealing with many of them. It's essential to stay organized, and the easiest way to do this is by grouping similar objects into *collections*.

For example, when you look at a scrolling list on your iPhone, the data in each row is represented by an object. And if that that row contains a person's name and age, you're likely to have other objects for those personal details.

`NSArray` is a natural collection for this kind of list view. You can specify the order within the collection so the first row in the list is the first object in the array. This array class also provides mechanisms for sorting and filtering the data, making it easy to model that behavior in a user interface.

When you want to collect objects by using a unique key, use the `NSDictionary` class. It implements an associative array that lets you look up objects (values) associated with a key.

Note: If you've used other languages, associative arrays are called a lot of different things: hash tables, hashes, maps, and containers are but a few. Regardless of name, they all let you find a value with a key.

You might use an *NSDictionary* to store the data for each row in your list. Since you can store *NSString*, *NSDate*, and *NSNumber* in a collection, it's easy to create a collection for that information:

```
NSDictionary *personalDetails = [NSDictionary
dictionaryWithObjectsAndKeys:
    @"Craig", @"name", [NSNumber numberWithInt:50],
@"age"];
```

You can then query the dictionary collection whenever you need one of the objects:

```
NSInteger age = [[personalDetails objectForKey:@"age"]
integerValue];
BOOL crankyOldMan = NO;
if (age > 50) {
    crankyOldMan = YES;
}
```

You can use *NSSet* to collect distinct objects whose order isn't important. Typically, you'll use this type of collection as you would use a set in mathematics. Methods let you test for equality, intersection, and subsets.

As your application gets more complex, you'll find that you start to combine these collections. For example, your list view could use an *NSArray* for each instance of a row and an *NSDictionary* for the actual row data.

Likewise, it's possible to have an *NSDictionary* that contains an *NSArray* at a key. Imagine that your personal details include a *siblingNames* key; that data would be well represented by an array. If you don't have any brothers or sisters, the array is empty, and no matter how prolific your parents were, the array would grow to fit the size of your family.

Copying in Depth

When you make a copy of a collection by sending the *–copy* message, the result is a shallow copy of the object. The copied collection can be modified independently of the original collection, but the objects in both collections are shared.

To perform a deep copy of a collection, you use a *copy items* method. For example, with an `NSArray`, you use *–initWithArray:copyItems:* like this:

```
NSArray *original = [NSArray
arrayWithObjects:@"Mimeoscope", @"Cyclostyle", nil];
NSArray *shallowCopy = [original copy];
NSArray *deepCopy = [[NSArray alloc]
initWithArray:original copyItems:YES];
```

In *shallowCopy*, both objects in the array are the same instances that were in the original. The *deepCopy*, on the other hand, has new instances of *"Mimeoscope"* and *"Cyclostyle"* in the new array.

The *NSDictionary* and *NSSet* collections use the same pattern with the *–initWithDictionary:copyItems:* and *–initWithSet:copyItems:* methods.

Property Lists

When you go to the effort of organizing your objects in collections, it's likely that you'll want to save that work. Both *NSArray* and *NSDictionary* provide a *–writeToFile:atomically:* method. They also provide *+arrayWithContentsOfFile:* and *+dictionaryWithContentsOfFile:* methods to read the information back from the file.

What gets stored in this file is a *property list,* a standard format that's used throughout Cocoa Touch as a lightweight and portable mechanism for persistence. Property lists store instances of objects in a file with a .plist extension. The contents of the data can be stored as XML for maximum portability, or in a more efficient binary format.

If you opened your Flashlight application's resource folder, you might have noticed a file named Flashlight-Info.plist. This important property list contains information that describes your application to the iPhone OS. It's where you specify the icon that appears on the home screen, for example. Similarly, when a user updates your application's settings, the configuration is written into a property list.

When you're using property lists, it's important to use only the following classes: *NSArray, NSDictionary, NSString, NSData, NSDate,* and *NSNumber* (integer, floating-point, and Boolean values). Any hierarchy of these objects can be stored in the file. An array of dictionaries containing numbers, strings, and dates is completely valid.

Tip: If you need to archive an object graph that contains classes not supported in a property list, take a look at the *NSKeyedArchiver* class and *NSCoding* protocol.

Mutable vs. Immutable

While looking at *NSString*, you may have noticed that the content of the string is never modified directly. If you've checked out the *NSArray* class, you might be concerned that there doesn't appear to be any way to add or remove items to the array. Don't worry; the folks who designed Cocoa weren't *that* lazy!

Many of the classes for managing data are broken in two: one class to manage the data that never changes and a subclass for the part that can change. If the class name for the constant data is *NSString*, the class name *NSMutableString* is used for the one that can be modified.

Why use two classes when they could have just implemented one? For the same reason that constant values in other code are a good thing: You know that the data is never going to change as it's processed. You can make assumptions that you wouldn't have otherwise made.

Note: If you've ever had an array's size change as you're iterating over its elements, you know the pain of the bugs that these kinds of bad assumptions can incur.

When you're building your own classes, you'll often have to choose which of these two variants to use for your instance variables. A good rule of thumb is to use an immutable object like *NSString* or *NSNumber* when the value is replaced entirely. When you were making awesome strings, there would have been no point in using an *NSMutableString* since the *–setOriginalString:* method updated the whole string rather than making changes to individual characters.

Collections, on the other hand, are often updated incrementally. It's common to add or remove items from an array, so you'd want to use *NSMutableArray* and gain the *–insertObject:atIndex:* and –*removeObjectAtIndex:* methods.

Make It Mutable

What do you do when you have an immutable object and you need a mutable one? You'll find that a lot of methods in the foundation classes return immutable objects as results. These system classes assume that you're probably not going to want to modify the results. And for the most part, this is true. But as a developer, you know that it's the special cases that cause the most headaches!

Here's a simple example of a method that returns immutable data—the `NSString` method that splits a string using a separator:

```
- (NSArray *)componentsSeparatedByString:(NSString *)
separator;
```

If you're parsing a comma-separated string to just loop through the values, an immutable copy works just great:

```
NSString *oneHand = @"Thumb,Index,Middle,Ring,Little";
NSArray *fingers = [oneHand componentsSeparatedByS-
tring:@","];
for (NSString *finger in fingers) {
    NSLog(@"The finger is %@", finger);
}
```

But what if you want to modify the fingers? This is one of those rare and important cases where you need a mutable copy. Everyone knows the last finger is called the "Pinkie".

Luckily, it's an easy fix using the *–mutableCopy* method that's defined by `NSObject`:

```
NSString *oneHand = @"Thumb,Index,Middle,Ring,Little";
NSArray *fingers = [oneHand componentsSeparatedBy
String:@","];
NSMutableArray *mutableFingers = [fingers mutableCopy];
[mutableFingers removeLastObject]; // OUCH THAT HURT!!!!
[mutableFingers addObject:@"Pinkie"]; // IT'S A MIRACLE
OF MODERN SCIENCE
for (NSString *finger in mutableFingers) {
    NSLog(@"The finger is %@", finger);
}
```

It's important to note that you can't make a mutable copy of just any object. The class you're copying needs to support the *NS MutableCopying* protocol. If the class definition doesn't include *<NSMutableCopying>,* an exception will be raised when you try to make the copy.

Note: *–mutableCopy* behaves like *–copy* when you're dealing with collections. A shallow copy is performed.

Luckily, all of the value objects and collections support the protocol. And when a class doesn't support the protocol, it's probably for good reason. What would it mean to have a mutable copy of a *UIView* or an *NSURLConnection*? You'd have too many dependencies for such an abstraction to be understandable.

Protect Your Data

It's common to use mutable data in a method implementation. You have no idea how *–componentsSeparatedByString:* is implemented, but it's likely that the array was built up as the parser moved from the beginning to the end of the string. So why return it as an immutable array?

Cocoa Touch is preventing you from shooting yourself in the foot. Besides the fact that you probably don't need the mutable copy, when you have two objects with a copy of the same data, someone is going to get hurt if one object changes the data without the knowledge of the other object.

Imagine an array that's shared between two objects that are adding and removing items. The first object computes the length of an array. Then another object removes the last object in the array. When the first object goes to access what it thinks is the last item, an exception occurs because the array index is out of bounds. It's a hideous bug to track down, because each object appears to be doing the right thing. It's far from obvious that the root of the problem is a shared collection.

To return an immutable copy of a mutable instance, you rely on the fact that the mutable class is a subclass and that its superclass implements the <*NSCopying*> protocol. When you send a *–copy* message, you get back a nonmodifiable copy.

Suppose the *mutableFingers* array you created above is an instance variable in a *MyHand* class. You've decided to keep the variable mutable so you can accommodate those ignorant people who insist on calling the last finger "Little." When another object asks the instance for the finger names, you'd return them like this:

```
@implementation MyHand
…
- (NSArray *)fingers {
    NSArray *result = (NSArray *)[[mutableFingers copy]
autorelease];
    return result;
}
```

The resulting object won't break your *mutableFingers*. Even if someone wants to call that last finger "Auricular" and stick it in his ear.

Delegation and Data Sources

Have you ever needed to perform a task and had no idea how to do it? Maybe it's fixing a dishwasher or knitting a pair of socks. You can either learn to do it yourself, or find someone to help you with the parts you don't understand.

Cocoa Touch has a design pattern called *delegation* that lets your application help out system classes that don't know what you want to do. The way it works is simple: You introduce one object to another object that's able to answer any questions. By assigning a *delegate*, you provide hooks into code that can respond to requests and to state changes.

If your object is willing to be a delegate, it needs to follow a protocol. As you saw on page 44, the protocol is like a contract. Many of the methods in that contract will be optional, but others will be required.

This pattern is used extensively throughout Cocoa. If you type *delegate* into the documentation viewer, you'll get back over 100 results. You can tell which classes support delegation, because they have a *delegate* property. The actual protocols for delegation are usually suffixed with *Delegate*.

The best way to see what this looks like in practice is by checking out one of the classes that uses delegation: the *UIPickerView* control that selects values from a scrolling list (Figure 3-2).

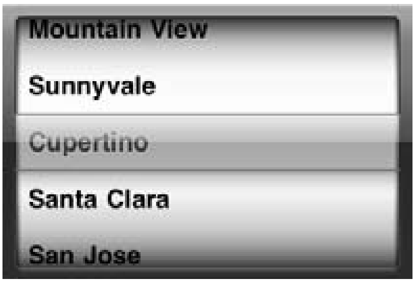


Figure 3-2: *The standard picker view for Cocoa Touch. The values that are displayed come from a delegate and a data source.*

When you look at the documentation for the *UIPickerView* class, you'll see the delegate instance variable defined like so:

```
@property (nonatomic, assign) id<UIPickerViewDelegate>
delegate;
```

That declaration tells you that the delegate can be any class (represented by *id*) as long as it supports the *UIPickerViewDelegate* protocol. When you open the documentation, you'll see that the protocol has three primary tasks for the delegate. You'll be asked to provide dimensions and content, as well as to respond to selections. All of these tasks are optional, but your picker is going to be pretty boring if you ignore all the questions.

Initially, you'll probably want to customize the list of names in each row. The picker view doesn't know how to label the rows, so it will ask you what you want with the *–pickerView:titleForRow:forComponent:* message. As a delegate, you could implement a method like this:

```
- (NSString *)pickerView:(UIPickerView *)pickerView
titleForRow:(NSInteger)row forComponent:(NSInteger)
component {
        return [NSString stringWithFormat:@"Row %d",
row];
}
```

And each row in the picker would be labeled "Row 0", "Row 1", "Row 2", and so on.

You'd also like to know when someone taps on a row. If you don't implement the following method, you'll never know when the picker value changes:

```
- (void)pickerView:(UIPickerView *)pickerView
didSelectRow:(NSInteger)row inComponent:(NSInteger)compo-
nent {
    NSLog(@"You picked row %d", row);
}
```

The best place to implement both of these methods is in a *UIView-Controller* subclass. This controller follows the MVC design pattern described on page 88, so it's easy to coordinate the activities between the picker view and a model that supplies the titles and that updates its state when a selection changes.

The picker view knows how many rows to display because the delegation pattern has a slight variation—the data source. As with a delegate, an instance variable with the name of *dataSource* must implement the *UIPickerViewDataSource* protocol:

```
@property (nonatomic, assign) id<UIPickerViewDataSource>
dataSource;
```

And the *dataSource* is required to implement two methods: *– numberOfComponentsInPickerView:* and *– pickerView:numberOfRowsInComponent:*.

The first method tells the picker view how many columns to display, while the second method specifies the number of rows. For one column and 10 rows, you'd use these implementations:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView
*)pickerView {
    return 1;
}

- (NSInteger)pickerView:(UIPickerView *)pickerView number
OfRowsInComponent:(NSInteger)component {
    return 10;
}
```

As with the delegate methods, it's likely that you'd use your controller's model to fill in these values rather than using hard coded results.

You use this same delegate and data source pattern when you're dealing with lists of scrolling data, like the bookmarks in Safari or the inbox in Mail. Both use a *UITableView*, which is controlled using the same techniques as the *UIPickerView*.

Don't think that this delegate design pattern is limited to views in your user interface. Delegates are frequently used when downloading data from a URL. The *NSURLConnectionDelegate* defines a protocol to provide any connection authentication and to notify you as the download proceeds.

Targets and Actions

You use a mechanism similar to delegation when you're working with controls in Cocoa Touch. When you tap on a button, drag a slider to adjust the volume, or toggle a switch, these user interface elements need to notify other parts of the application of this state change. So how do these controls let everything know what's going on?

The notification occurs using a *target-action* design pattern. In other words, you can set up each control with a *target*—an object that will be notified of the change. As with a delegate, you can choose any object.

Unlike with a delegate, the action can be any method defined by the object. The important thing is that the method conforms to one of these two signatures:

```
- (IBAction)actionOne {
}
- (IBAction)actionTwo:(id)sender {
}
```

Interface Builder uses `IBAction` to identify the actions in your code (explained in detail later). The second form takes a single parameter that contains the object that sent the action. The *sender* can be used while you're processing the action.

If you have a single action that's used by several instances of *UIButton*, you can use the *sender* to determine which button triggered the action. Another use for the sender is to query the control's state. For example, if you received an action from a *UISwitch* control, you'll want to know whether the switch is on or off:

```
- (IBAction)toggleSwitch:(id)sender {
    // sender is an instance of UISwitch, so use its
-on method:
    if ([sender on]) {
        NSLog(@"AFFIRMATORY DIXIE CUP");
    }
    else {
        NSLog(@"NEGATORY MR CLEAN");
    }
}
```

This code also has the advantage of being easy to read and understand. Especially if you understand CB slang from the 1970s.

Next up, you need to learn how these controls get hooked up with the target object and the action method. Actually, you can use two ways of defining targets and actions: one uses code and the other uses Interface Builder. You'll start with the code to get a deeper understanding of what's going on behind the scenes.

User Interface: The Hard Way

Every control in Cocoa Touch is a subclass of the `UIControl` class. This class defines the following method:

```
- (void)addTarget:(id)target action:(SEL)action forControl
Events:(UIControlEvents)controlEvents
```

This method takes three parameters. The first is the target object that will be notified about the control event. Second, the action parameter, defines the message that will be sent to the target object. Third, *controlEvents*, lets you specify the types of events that will trigger the action.

The most common events are *UIControlEventValueChanged* and *UIControlEventTouchUpInside*. The first, *UIControlEventValueChanged*, is used for controls where a changed value is important. Examples here are dragging a slider (*UISlider*), typing in text fields (*UITextField*), and flipping switches (*UISwitch*).

UIControlEventTouchUpInside events are typically used with `UIButton` controls because you only want to know that the button was released with the user's finger inside the bounds of the button. It's important not to use the *UIControlEventTouchDown*, even if you're a fan of the NFL. This event doesn't give users a chance to drag their finger off the button to cancel the button press.

Here's how you'd put this all together in your controller object:

```
#define LOUDER 11.0f

- (void)updateVolume:(id)sender {
    float volume = [(UISlider *)sender value];
    if (volume >= LOUDER) {
        NSLog(@"NIGEL TUFNEL WOULD BE PROUD");
    }
}

- (void)hitOrMiss {
    if ([mySlider value] > LOUDER) {
        NSLog(@"OF COURSE IT'S A HIT!!!");
    }
}

- (void)setupControls {
    [mySlider addTarget:self action:@selector
(updateVolume:) forControlEvents:UIControlEventValue
Changed];
    [myButton addTarget:self action:@selector(hitOrMiss)
forControlEvents:UIControlEventTouchUpInside];
}
```

The *–updateVolume:* method is attached to the `mySlider` view reference in the controller object. When the slider moves and its value changes, the volume will be checked; if it goes past 11, a message appears in the console log.

The action signature without a sender parameter was used for the *–hitOrMiss* method. This method also shows how you can use other view states to act on the button press. If the slider is above 11, the button lets you know that you have a hit.

Note: Page 75 warned you about getting the number of colons in the selector statement correct. If you had typed *@selector(updateVolume)* or *@selector(hitOrMiss:)* in the code above, the colons won't match the methods you'd implemented. When you tested the controls, you'd be greeted with a crash and an "unrecognized selector sent to instance" message in your console log.

Along the Way

You're now probably expecting to see that easy way of setting up action messages on target objects. But before you can dive into Interface Builder, you need to prepare your controller's code so that objects and actions are exposed. This small amount of work will save you loads of time as your project evolves. It's well worth the investment.

As you've seen in the MVC design pattern, one or more view objects are managed by a controller. So where do you define these views for your controller? If you said "in the *@interface*," you win a prize!

To get some hands-on experience, download *MissingCD_Xcode_Projects.zip* from the Missing CD page at *www.missingmanuals.com*.

In this exercise, your controller is named *HitMakerViewController*, and it has three views: a slider, a text view, and a button. You define the instance variables just as you did on page 95:

```
@interface HitMakerViewController : UIViewController {
    UISlider *mySlider;    UIButton *myButton;
    UILabel *myLabel;
}
```

Also define properties for these instance variables so they're retained objects:

```
@property (nonatomic, retain) IBOutlet UISlider
*mySlider;
@property (nonatomic, retain) IBOutlet UIButton
*myButton;
@property (nonatomic, retain) IBOutlet UILabel *myLabel;
```

And here are the action method definitions:

```
- (IBAction)updateVolume:(id)sender;
- (IBAction)hitOrMiss;
```

By now, you've probably figured out that the first two letters in Objective-C identifiers are important. The *IB* is a hint that helps you solve the mystery. It stands for Interface Builder.

If you look up the definitions of *IBOutlet* and *IBAction*, you may be confused by what you find in UINibDeclarations.h:

```
#define IBOutlet
#define IBAction void
```

When you add *IBOutlet* and *IBAction* to your source code, you're not adding any additional functionality. As you see above, the definitions of these two identifiers are essentially NOPs. All you're doing is marking your code so that Interface Builder has something to parse.

IBOutlet is used to mark objects that will be used in the graphical editor and in your source code. Similarly, *IBAction* is used to identify methods that are shared between the two editing environments.

The Magical Way

From your source code's point of view, the *mySlider*, *myButton*, and *myLabel* objects will magically appear when your application launches. This kind of sorcery makes your life a lot easier, but it's important to understand the trick so you can take advantage of the underlying sleight of hand.

It all begins with the application configuration file. HitMaker-Info.plist contains a Main NIB file base name (*NSMainNibFile*) value. This tells Cocoa Touch to load the named file at launch. When *MainWindow* is specified, all the objects in MainWindow.xib will be loaded into memory.

Note: The name "NIB" is short for "NeXT Interface Builder" and was used as the extension for files saved with older versions of the software. Many developers and most parts of the framework continue to use this term, although the files now use the .xib extension. This new format is based on XML, hence the first letter in the new name.

In some contexts, developers talk about NIB files; in others they use the term "XIB files." They're the same thing: a file that contains part of your user interface.

A part of the loading process is setting all the instance variables you've defined as being an *IBOutlet*. The NIB loading mechanism uses your accessors to set the instance variables, so *–setMySlider:*, *–setMyButton:*, and *–setMyLabel:* are all called using the object in memory.

The coolest part is that the objects that are loaded have all the settings you've made in the Interface Builder. If you've changed a view's background color, that change will be reflected in memory, and your instance variable has access to it.

After an object is read from the NIB/XIB file, it's sent an *–awakeFrom Nib* message. This gives you a chance to perform any additional setup on your objects. This point is important, because not every object property is editable using Interface Builder. Also, in some cases, you'll want to configure objects based on internal logic.

You'd be right in thinking that now would also be a good time to update your view outlets. However, if you send the *–setValue:* message to your *mySlider* instance variable in the *–awakeFromNib* implementation, nothing will happen. These views haven't been loaded yet, so the instance variables still have nil values. When you send a message to a nil object, the message is quietly discarded. This seems like a bug, but the engineers who designed Cocoa Touch did you a favor: views are loaded *lazily*.

Here's how it works: Views can use a substantial amount of memory, especially if you're using heavyweight objects like `UIWebView` that include a cache of images and other web content. On the iPhone, every byte counts, so view objects are loaded only when they need to appear on the display. An added benefit to this approach is that it takes less time for your app to load when it contains many views.

To find out when your view is loaded from the NIB/XIB file, override the *–viewDidLoad* method. The following code shows how you could initialize the position of your *mySlider* object:

```
- (void)viewDidLoad {
    mySlider.value = 11.0; // LOUDER
    [super viewDidLoad];
}
```

Cocoa Touch not only loads views lazily, but it can also remove them automatically when memory is running low. The framework knows which views are on the display and safely reclaims storage by removing any that aren't visible. When it does, it also sends your view controller a message letting you know about the low memory condition:

```
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}
```

If your view relies on a large cache of information or other data that can be reconstructed easily, then the *–didReceiveMemoryWarning* method is a great way to clear out those objects as well.

Of course, you'll want to know when your view is removed, so you can release your instance variables to save even more memory. If you don't override the *–viewDidUnload* method, your instance variables will continue to occupy space, but will be useless because they don't have a parent view. The implementation is simple:

```
- (void)viewDidUnload {
    self.mySlider = nil;
    self.myButton = nil;
    self.myLabel = nil;
}
```

This brings up a subtle point: The *–viewDidLoad* and *–viewDidUnload* methods can be called multiple times, so avoid any initialization that should be performed only once.

Tip: You don't have to use a NIB/XIB file created with Interface Builder in order to construct a view hierarchy. Overriding *–loadView* lets you build your own view hierarchy with code. It's harder to do it this way, but with some user interfaces it's a necessity.

If you're going to great lengths to adjust views loaded from a NIB/XIB file, typically when you're doing extensive animation, it's often easier to just override *–loadView* with code that allocates the view objects, initializes, and adds them to the main view with *–addSubview:*.

Finally, since the NIB loader retained your outlet objects after storing them in memory, you need to do one final thing in your view controller: Release the views when the view controller is deallocated:

```
- (void)dealloc {
    [mySlider release]; mySlider = nil;
    [myButton release]; myButton = nil;
    [myLabel release]; myLabel = nil;

    [super dealloc];
}
```

Now that you're an expert with the magic behind NIB/XIB files, you're ready to take a look at how they're created!

User Interface: The Easy Way

At first glance, this step-by-step guide may not look so easy. Don't worry, once you learn the work flow, creating objects and hooking them up with Interface Builder is a snap. Time to get started!

A Head Start

Download the "HitMaker – Start" project from the Missing CD page at *www.missingmanuals.com/cds*. This project contains all the code for the actions and targets you'll be using during the exercise. To begin, you'll get Xcode and Interface Builder running, and open the files for editing:

1. **1. Open the HitMaker.xcodeproj file in the HitMaker – Start project folder.**

   Xcode launches and shows you the project contents.

2. **Open the disclosure triangle to get a list of files.**

   XIB files are located in the Resources group.

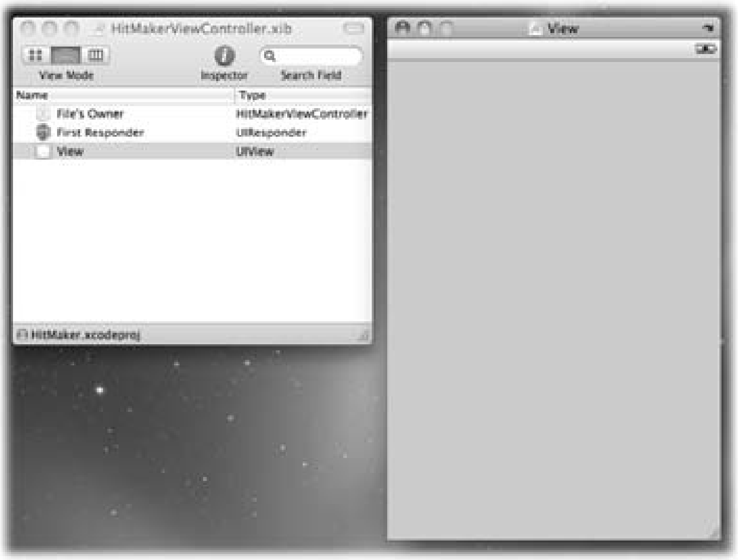
3. **Double-click HitMakerViewController.xib to open the XIB file with Interface Builder.**

   A new icon with a drafting triangle appears in your Dock as the file is loaded.

After the document file opens, you see a list of objects that are contained in the file.

Tip: The document's name appears in the title bar and in the Window menu. It's common to have several NIB files open at once, so use these tools to find the correct document.

You also see a gray View window. It's a preview of the View object contained in the document. Meanwhile, in the document window, you see the word "View" with no disclosure triangle next to it (Figure 3-3). That indicates that the *UIView* object does not have any children, or subviews. You're about to change that.



Name	Type
File's Owner	HitMakerViewController
First Responder	UIResponder
View	UIView

Figure 3-3: *Here's the blank palette you're going to working with. The document window is on the left, and the application's only view is on the right. Note that the View listed in the document window doesn't have any children (subviews) yet.*

Tip: If you accidentally close the View window, you can reopen it by double-clicking View in the document window.

Your First View

That gray window looks pretty boring, doesn't it? Time to start adding new views for the labels, slider, and button!

1. **Choose Tools→Library. If you're the kind of developer who loves the keyboard, use the ⌘-Shift-L shortcut.**

   The Library palette opens.

2. **Scroll the list of Objects in the Library, and select the cell with the text "Label".**

   It should be in the first row of "Cocoa Touch – Inputs & Values". When you select the cell, you'll see a short description of the control and its function at the bottom of the palette.

3. **Using your mouse, drag the Label cell onto the gray View window.**

   See Figure 3-4 for a demonstration. When you're dragging over the View window, a green plus sign appears, which indicates that the object will be added when you drop it.

4. **When your cursor is over the View window, let go of the mouse button.**

   The label drops onto the view, and four resize handles indicate that it's selected.

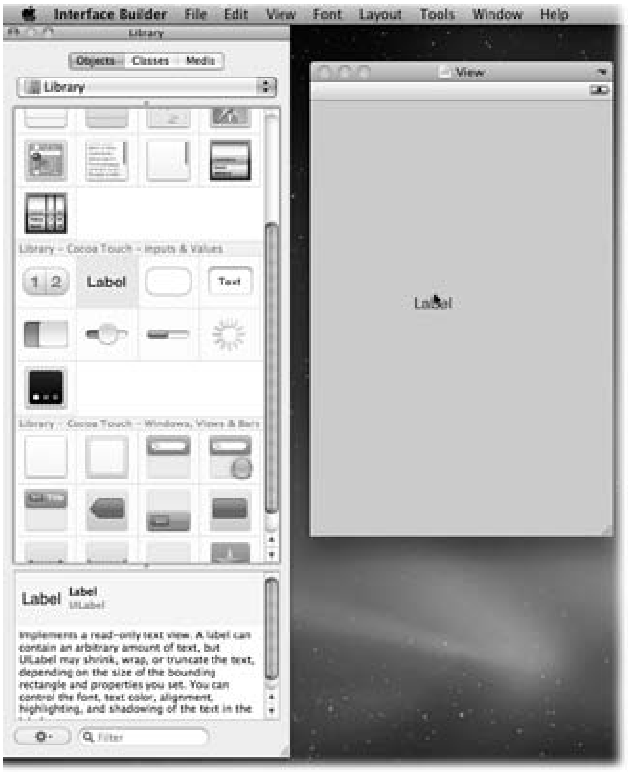
5. **Double-click Label and type *VOLUME NOB*.**

   Behind the scenes, you just replaced a line of code:

   ```
   [label setTitle:@"VOLUME NOB"];
   ```

6. **Drag the left and right resize handles until the dashed blue lines appear.**

   The edges of the label snap to the lines, helping you align your controls.



Implements a read-only text view. A label can contain an arbitrary amount of text, but UILabel may shrink, wrap, or truncate the text, depending on the size of the bounding rectangle and properties you set. You can control the font, text color, alignment, highlighting, and shadowing of the text in the

Figure 3-4: *On the left side, you see the Library palette with the Label control selected. You can drag any item in the Library to create a new instance of the object. Here, the Label is being dragged onto the View window.*

Tip: If you want to see the layout rectangles in View, press ⌘-L to display the bounds. If you press Option while the label is selected, you'll see the number of pixels between various interface elements.

7. **Make sure the label is still selected, and choose Tools→Attributes Inspector or press ⌘-1. In the Attributes Inspector's Layout section, click the Alignment (center).**

   You've just replaced another line of code:

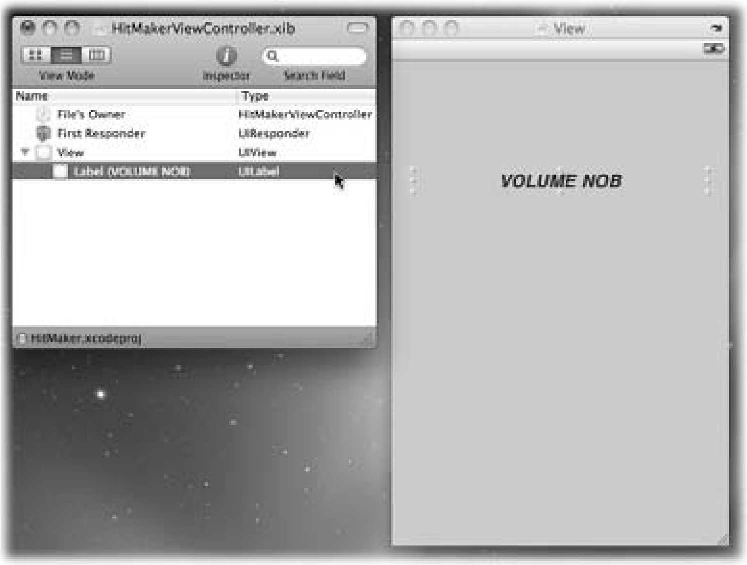
   ```
   [label setTextAlignment:UITextAlignmentCenter]
   ```

8. **With the label selected, choose Font→Bold (⌘-B). If you want to go really crazy, press ⌘-I to make it italic, too.**

   Che bellezza! But even more beautiful is this code you didn't have to write:

   ```
   [label setFont:[UIFont boldSystemFontOfSize:17.0f];
   ```

As shown in Figure 3-5, the document window now shows a disclosure triangle for View. When you click it, you see the name is Label (VOLUME NOB) with a type of *UILabel*.



Name	Type
File's Owner	HitMakerViewController
First Responder	UIResponder
View	UIView
Label (VOLUME NOB)	UILabel

***Figure 3-5:** Once you add a* UILabel *containing the text VOLUME NOB, it appears under the main* UIView *for the window. When you double-click a view in the document window, it gets selected and displays resize handles–helpful when you're looking for a subview in a complex user interface.*

Congratulations! You just added your first subview to the hierarchy.

You're in Control

You have your first object. But it's just a label, and you're not using it in your code. Time to remedy that with a control:

1. **Just below the Label in the Library palette, you'll see an icon that looks like a slider. Drag it onto the gray window.**

   A slider control appears, with resize handles on the left and right. Some controls can't adjust their width or height—the slider is one of them.

2. **Drag the slider so it's below the VOLUME NOB label.**

   The dashed blue lines help you center things.

3. **Drag the handles until they snap onto the dashed blue lines.**

You've created an object that you can use for the *mySlider* instance variable. All that's left to do is to hook it up to an outlet:

1. **In the document window, select File's Owner.**

   The type is *HitMakerViewController*. That's the same as the class you're writing!

2. **Choose Tools→Inspector and then select the second tab at the top (a blue circle with white arrow pointing right). Or just press ⌘-2 as a shortcut to select this Connections Inspector tab.**

   You see *myButton*, *myLabel*, and *mySlider*. Thanks to the *IBOutlet* identifiers you added to your source code, those show up automatically.

Tip: To prove this to yourself, open HitMakerViewController.h and remove the *IBOutlet* before *UILabel *myLabel* and save the file. When you return, the outlet won't appear in the inspector for File's Owner. After you're done with this experiment, put the *IBOutlet* back; you'll need it in the following steps.

3. **To link the outlet to the view, click the circle to the right of *mySlider* and drag the mouse.**

   A blue line appears that lets you know you're connecting to an object.

4. **Continue dragging the mouse until you're hovering over the slider control. When a blue box appears around the slider, release the mouse button and you're done.**

   In your list of outlets for File's Owner, *mySlider* is connected to a Horizontal Slider (Figure 3-6). Yay!

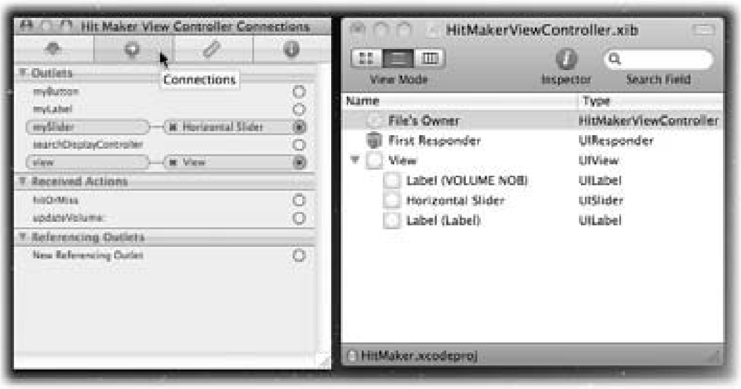


Figure 3-6: *After you add the slider control as an outlet, the Connection Inspector for the* HitMakerViewController *looks like this. As you can see,* mySlider *is connected to a Horizontal Slider. When you're working with Interface Builder, it's a good idea to use this inspector on File's Owner to check that all your outlets and actions are connected correctly. If a connection is missing, your controller code won't communicate with the view correctly.*

Tip: You can also connect outlets by Control-dragging from File's Owner to the control you want to connect. When you release the mouse button, you'll see a list of outlets to choose from.

Your document is starting to look like a user interface. You have two more outlets to connect:

1. **Drag another *UILabel* object from the Library onto the gray View window. Place it just below the slider, and adjust it like you did for the VOLUME NOB label, but don't bother changing the text of the label.**

   Unlike with the first label, you're going to update this one with code.

2. **Now go to the Connection Inspector after selecting File's Owner, and drag a blue line from the circle next to *myLabel* to the new label.**

   Finally, you need a button.

3. **In the Library Inspector, click the button icon (to the right of the label icon).**

   The description you should see at the bottom of the window is the Rounded Rect Button and the *UIButton* class name.

4. **Drag the Rounded Rect Button to the View window, and drop it in place just below the other controls.**

   As with a label, you can double-click the button to enter text for the button to display.

5. **Double-click the button and type *HIT OR MISS*. Then add another connection from the *HitMakerViewController* outlet named *myButton* to the button you just added.**

   The drag-to-connect procedure is the same as you've just done for the other two outlets.

6. **Press ⌘-S to save your work.**

   Interface Builder writes a new XIB file with the name HitMaker ViewController.xib. Don't close the document window; you're not done with it yet.

Next, you're going to head back to the connection attributes for the *HitMakerViewController* and connect `myButton` up to this new button.

Note: Some developers prefer to place and configure their controls all at once instead of switching back and forth between inspectors. Choose whichever work flow works best for you; just remember to connect your outlets after all the control placement is complete.

When you're just starting out, it's easy to forget these connections. Any code that references an unconnected outlet will be sending a message to a nil object and nothing will happen. Before you spend a lot of time debugging your code, make sure that everything is hooked up correctly!

Switch back over to Xcode so you can build the application with this new XIB file.

When you build and run the app using ⌘-Return in Xcode, you'll see that the app starts up and the controls get initialized (Figure 3-7). The *–viewDidLoad* code adjusted the slider to the middle position and changed the label text and color. But nothing happens when you drag the slider or press the button.

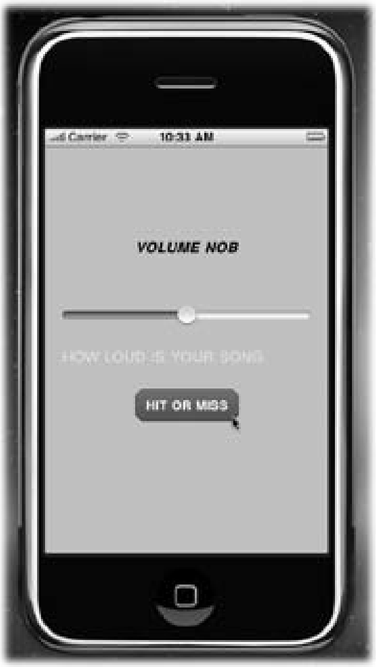


Figure 3-7: *After connecting the outlets to the slider and label controls, the* HitMakerViewController *code in* –viewDidLoad *can update the position of the slider along with the label's text and color. When you launch the application now, nothing happens if you try clicking the button or slider.*

Get a Little Action

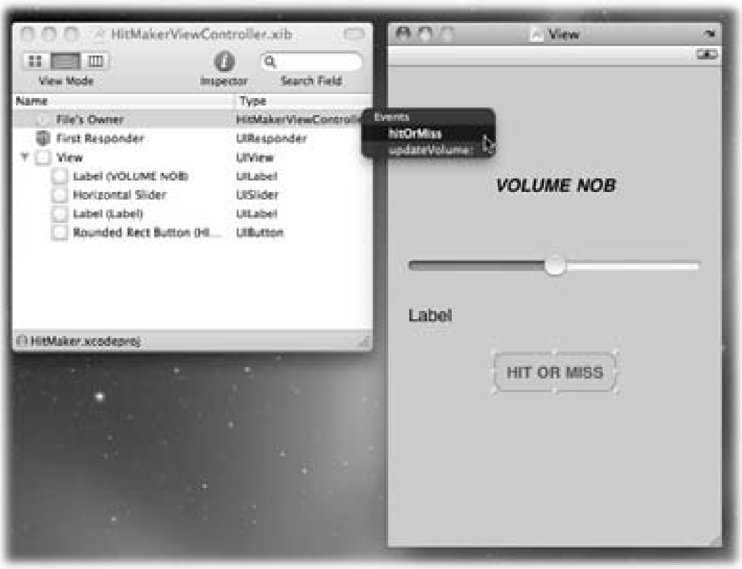
You've made a great start, but you need to get some action. No, not that kind. It's time to hook up an action to your new controls:

1. **Switch back to Interface Builder by clicking on its Dock icon.**
2. **In the HitMakerViewController.xib document window, select File's Owner.**
3. **Press ⌘-2 to bring up the Connection Inspector.**

Remember those *IBAction* identifiers you added to your controller's header file? You can see them listed under Received Actions. Make sure you can see both the gray view window and the document window where File's Owner is listed.

1. **Press Control and click the HIT OR MISS button. Drag the mouse, and you'll see a blue line. Since you want this action to be performed by the *HitMakerViewController*, move the blue line to that type in the document window.**

   When you let go of the mouse button, you'll see a menu appear with the events supported by that view controller (Figure 3-8). Click *hitOrMiss*, and you're done.



Name	Type
File's Owner	HitMakerViewControll
First Responder	UIResponder
View	UIView
Label (VOLUME NOB)	UILabel
Horizontal Slider	UISlider
Label (Label)	UILabel
Rounded Rect Button (HI...	UIButton

Figure 3-8: *After dragging a connection from the HIT OR MISS button to the* HitMakerViewController, *you'll select the* hitOrMiss *action. This move causes the named method to be called when the button is pressed.*

2. **Control-click the slider control. Drag the blue line onto *Hit MakerViewController*, release the mouse button, and then click** *updateVolume:*.
3. **Save the changes with ⌘-S.**

Go to the document window, select File's Owner, and press ⌘-2 to open the Connection Inspector. You'll see *hitOrMiss* is connected to a Rounded Rect Button for the Touch Up Inside event. Similarly, the Horizontal Slider's Value Changed event is connected to the *updateVolume:* method (Figure 3-9).

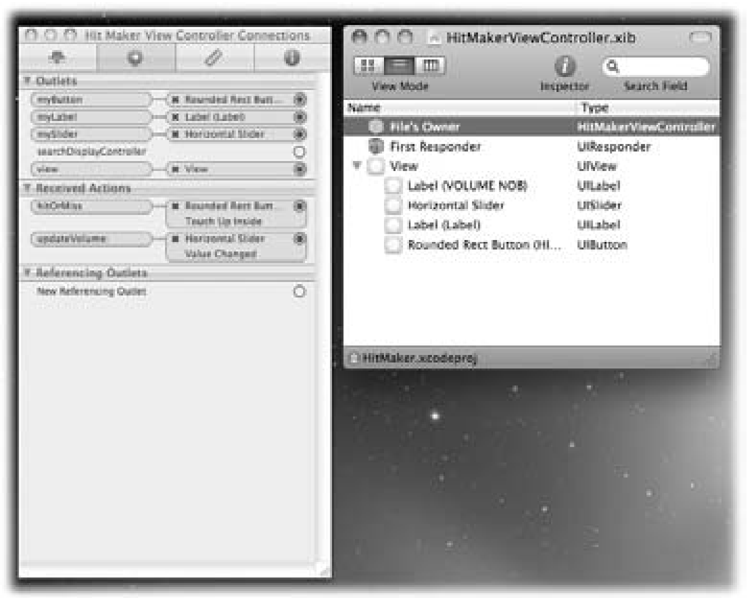


Figure 3-9: *The* HitMakerViewController *fully connected. When you select File's Owner, you'll see that the controller's three outlets* myButton, myLabel, *and* mySlider *are connected to views. Similarly, the actions* –hitOrMiss *and* –updateVolume: *will be invoked by the button and slider controls.*

Tip: You can even Control-drag between the items listed in the document window. Some developers prefer this method since it doesn't involve opening the View window.

On page 112, you learned how events are processed while you were writing the *–addTarget:action:forControlEvents:* code. In essence, you've just replaced the code shown in *–setupControls* with a couple of clicks.

Now it's time to switch back to Xcode and test the app. Click its Dock icon and press ⌘-Return to build and run the app.

Squash Those Bugs

Just because you're using visual development tools doesn't mean there won't be bugs. As you run the app, you notice a couple of problems:

- **Bug.** The slider value only goes from 0.0 to 1.0 (Figure 3-10). It has to go to at least 11.0 before it can
- **Visual.** The text colors don't look so great on a gray background. It would also be nice if the text were center aligned. As is often the case with view attribute changes that are done in code, you don't know how things are going to look until runtime.

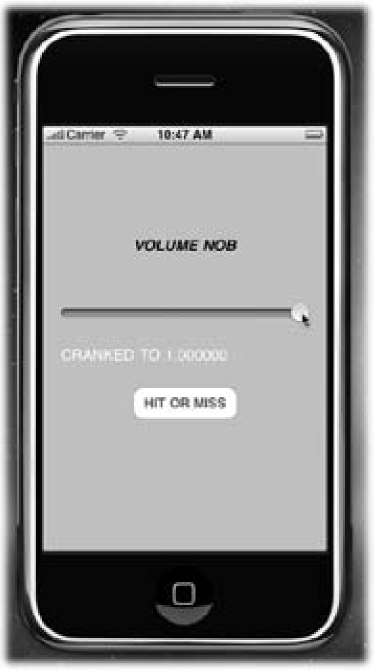


Figure 3-10: *When you run the application, the slider and button controls now function, but some problems linger. The slider control value only goes up to 1.0, the contrast for the label is too low, and the text isn't aligned correctly. You can fix all of these bugs with Interface Builder.*

Luckily, both of these things are easy to fix with Interface Builder. Click the Dock icon to bring your XIB file back into view. In the following steps, you'll fix first the bug, then the visual problems:

1. **In the document window, double-click *UISlider*. Then press ⌘-1.**

   The Attribute Inspector opens.

2. **Change Maximum value from 1.0 to 12.0. Because your app is just a bit more awesome than Mr. Tufnel's amp.**

3. Double-click the second label listed under View in the document window. In the View section, click the Background color well. When the color picker appears, select black with 100% opacity.

4. You'll also want to change the label text to white so you can see it against the new background. Do so by clicking the Text color well and picking the color (Figure 3-11).

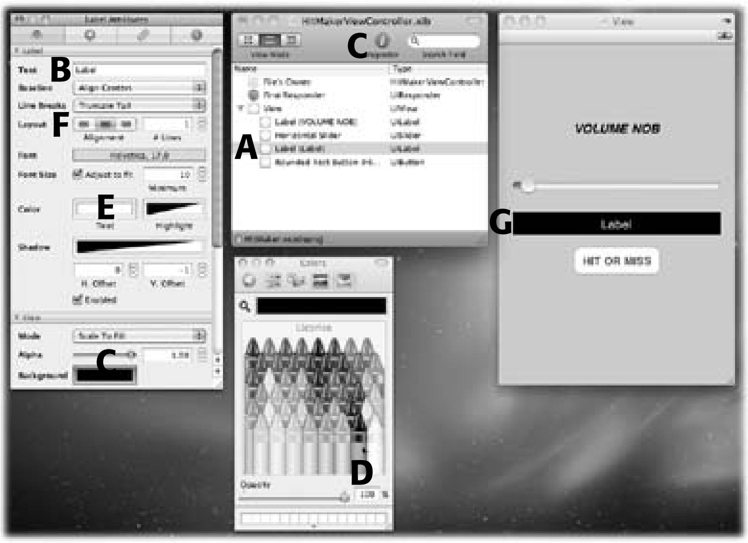


Figure 3-11: *Adjusting the colors and layout of the second label. Select the label (A) and then go to the Attributes Inspector (B). Click the Background color well (C), and select a black background (D). You can change the text color using its own color well (E). You adjust text alignment using the Layout control (F). The results are shown in the View window (G).*

5. Back under the label attributes, in the Layout section, click the center Alignment button.

6. In the View window, drag the label's height resize handle, and make it about 30 pixels tall.

Save your work, and then switch back to Xcode to build and run your application. As you run the app, everything looks good (Figure 3-12). And when you slide all the way to the right, you're ready to rock!

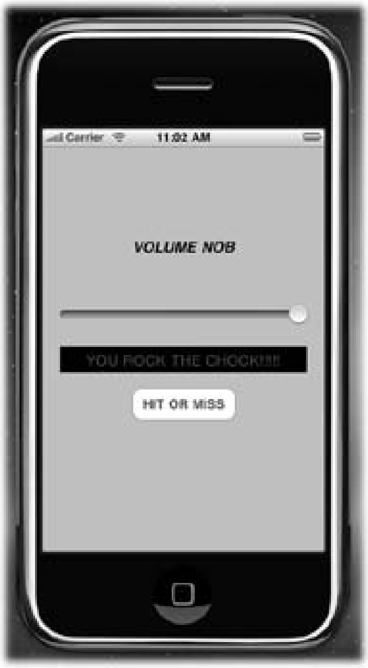


Figure 3-12: *After fixing the bugs with Interface Builder, the final build of your application rocks. Not only because you can move the slider all the way to the right and click the button, but also because you did all this work without writing a single line of code.*

But that's not the best part: Think about how much code you haven't written. All of this is possible just because you typed a few *IBOutlet* and *IBAction* identifiers in your header file. That's awesomely lazy.

Notifications

In a complex system like Cocoa Touch, the state of the various components is constantly changing. Many of the things that happen are out of your control, but if your app adapts to the new conditions, it'll provide a smoother user experience.

Wouldn't it be great for your app to know when these things happen? Cocoa Touch provides a general mechanism for letting your application know of these changes via *notifications*. For example:

- The iPhone is locked or unlocked (*UIApplicationWillResignActive Notification* and *UIApplicationDidBecomeActiveNotification*).
- The device orientation changes from portrait to landscape (*UIDeviceOrientationDidChangeNotification*).
- A user interface element causes the keyboard to appear on-screen (*UIKeyboardDidShowNotification*).
- The battery charge level changes or the device is plugged in (*UIDeviceBatteryLevelDidChangeNotification* and *UIDeviceBattery StateDidChangeNotification*).
- A text editing view is updated (*UITextViewTextDidChange Notification*).
- The pasteboard (Clipboard) changes (*UIPasteboardChanged Notification*).

Many classes generate notifications, and any object can generate them. The ones shown in this list are just a few posted by *UIApplication*, *UIWindow*, *UIDevice*, *UITextView*, and *UIPasteboard*. As you can probably guess by looking at the list, the standard convention is to use the class name at the beginning and the word "Notification" at the end.

When you want your app to be notified, you provide an object, a selector, and the name of the notification. A central service called *NSNotificationCenter* is responsible for moving the information from the objects that generate it to the ones that want to consume it. The objects that are distributed are instances of the *NSNotification* class. The notification center acts like a post office, and the notifications themselves are like letters.

When you want to receive a notification about a system event, you tell the notification center a little about yourself. Suppose you're interested in knowing when your application becomes active, either at launch or when the iPhone is unlocked. Typically, you'd add your view controller as an observer:

```
@implementation MyViewController
…
- (NSObject *)initWithNibName:(NSString *)nibNameOrNil
bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
bundle:nibBundleOrNil]) {
        …
  [[NSNotificationCenter defaultCenter] addObserver:self
     selector:@selector(becomeActive:
     name:UIApplicationDidBecomeActiveNotification
     object:nil];
        …
    }
```

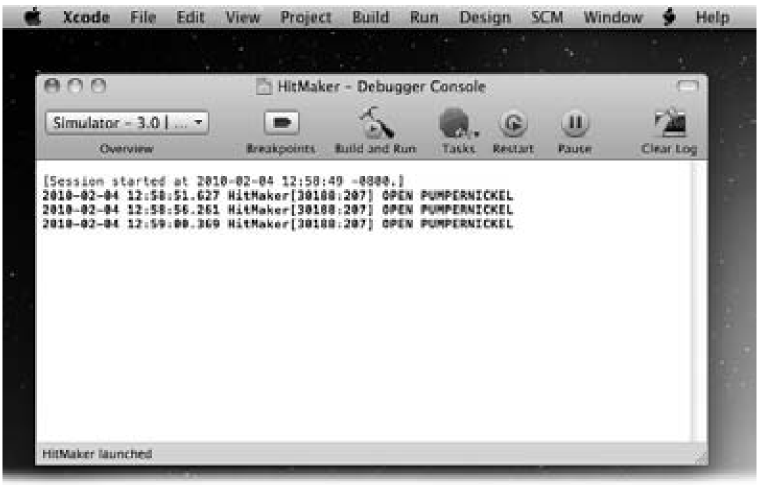
The first parameter is *self,* which tells the notification center that the view controller will handle the notification. The selector *become Active* specifies the message that is sent to the view controller, and the name indicates the controller's interests. The *UIApplicationDid BecomeActiveNotification* is delivered to the controller when a user is able to interact with the user interface.

Note: The object parameter isn't used in this example, but it can be used to specify the sender of the notification. By specifying *nil,* you're saying that you want to receive every notification of this type, regardless of the originating object.

Since you've registered to receive the notification, you now need to implement the method that will handle it:

```
- (void)becomeActive:(NSNotification *)notification {
    NSLog(@"OPEN PUMPERNICKEL");
}
```

Now OPEN PUMPERNICKEL will appear in the console log each time you start or unlock your app (Figure 3-13). Great feature, huh?



```
[Session started at 2010-02-04 12:58:49 -0800.]
2010-02-04 12:58:51.627 HitMaker[30188:207] OPEN PUMPERNICKEL
2010-02-04 12:58:56.261 HitMaker[30188:207] OPEN PUMPERNICKEL
2010-02-04 12:59:00.369 HitMaker[30188:207] OPEN PUMPERNICKEL
```

HitMaker launched

Figure 3-13: *When running an application in Xcode, you can access a debugging console using Run→Console or ⌘-Shift-R. Here the console is displaying an* NSLog *message when the application becomes active. The "OPEN PUMPERNICKEL" message is preceded by the date and time along with information about the process and thread that logged it.*

OK, so it's not a great feature. But what if that notification initiated a connection to refresh some data using the Internet? Or what about updating the device's location? It's a good trigger to let you know that the user is actively working with your application.

Similarly, you can use the *UIApplicationWillResignActiveNotification* to cancel any pending operations, especially ones that are keeping the wireless connections alive. If you stop the network operations, the radios can power down, and you'll increase the battery life. Everyone's happy!

It's very important to tell the notification center when your view controller object is no longer interested in knowing about these events. If you don't, your application can crash, because when you add an observer, notification is done by reference. If the object that was registered gets freed, the notification will be sent to an instance that no longer exists, and a memory fault will occur.

The fix is quite simple. Make sure you match every *–addObserver:selector:name:object:* with a corresponding *–removeObserver:name:object:*. You can also use *–removeObserver:* to remove all notifications at once. Since the observer was added during initialization, it makes sense to remove it during *–dealloc*:

```
@implementation MyViewController
…
- (void)dealloc {
    …
    [[NSNotificationCenter defaultCenter]
removeObserver:self];
    [super dealloc];
}
```

Note: These are notoriously hard bugs to track down. You'll see failures in *objc_msgSend* coming from code you didn't write, and they'll happen at times you don't expect. If you use notifica-tions, make sure that they get cleaned up properly.

Notifications aren't limited to ones generated by the system. You can define your own to get the various parts of your app in sync. Imagine several controllers in your application need to know when a network connection finishes. *NSURLConnection* is based on delegation, so only one object will know when the operation completes.

That object could pass the information onto interested parties by defining its own notification name. A notification name is nothing more than a string object, so you'd define its existence in the class interface file like this:

```
extern NSString *MyConnectionDidFinishNotification;
```

In the object's implementation, you'd define the value for the notification name:

```
NSString *MyConnectionDidFinishNotification = @"MyConnec-
tionDidFinishNotification";
```

And when the connection completes, you would post the notification like this:

```
[[NSNotificationCenter defaultCenter] postNotification-
Name: MyConnectionDidFinishNotification object:self];
```

Any object in your application that has called *–addObserver:selector:name:object:* with *MyConnectionDidFinishNotification* will then get a message and can update views, controllers, and models accordingly.

Singletons

The singleton design pattern is used in several key classes in Cocoa Touch. Singletons allow a class to return the same object instance anytime another object requests it. This is very handy when you *never* want more than one object of that type.

The *NSNotificationCenter* you just read about is an example. All notifications need to be routed through a centralized service for them to be effective. Just as importantly, objects need to be able to locate that central instance, and that's accomplished with the notification center's class method:

```
+ (id)defaultCenter;
```

Every object that sends a +*defaultCenter* message gets back the same object. That lets all objects add and remove observers in the same place.

Many times a class will use a singleton as a way to model that the thing only occurs once, either physically or virtually. You have only one application, set of settings, and hardware accelerometer, for example.

Unfortunately, there's no one standard for naming singletons. Older classes, such as those in the foundation, use the *default* prefix on the method name. Newer classes tend to use *shared* as a prefix. The most common singleton classes are:

- *NSFileManager (defaultManager)*
- *NSUserDefaults (standardUserDefaults)*
- *UIApplication (sharedApplication)*
- *UIAccelerometer (sharedAccelerometer)*
- *NSNotificationCenter (defaultCenter)*

Singletons as Globals

Many experienced developers recommend against using singletons. If you've ever programmed using global variables, you'll realize that singletons are really just a fancy way of representing a global state. You'll also appreciate that as applications grow and evolve, maintaining state that's shared by many components is a headache.

When you're implementing a singleton, you supply a class method that returns an instance of the class. So if you have a class named *MyFoot*, you'd define the method +*sharedFoot* like this:

```
@implementation MyFoot

+ (id)sharedFoot {
    static MyFoot *foot = nil;
    if (! foot) {
        foot = [[self alloc] init];
    }
    return foot;
}
```

The trick in this code is the static variable in the implementation; it holds a global variable that points to the shared object. When your application is launched, the value of that variable is *nil*. After the first message is received, the instance is allocated and returned for this and successive invocations.

Note: This singleton implementation isn't thread safe, nor does it prevent you from doing dumb things like releasing the global instance. If you're going to create singletons, take a moment to read Apple's recommendations in the "Creating a Singleton Instance" section of the *Cocoa Fundamentals Guide* (type *Singleton* in the documentation viewer).

Everything works great as long as your app needs to keep track only of one foot. But what happens when you add that new feature that uses both feet?

```
MyFoot *myLeftFoot = [MyFoot sharedFoot];
MyFoot *myRightFoot = [MyFoot sharedFoot];
[myLeftFoot shoot];
```

Not only are you shooting yourself in the foot, you're shooting them both at once. And you can't tell your left foot from your right because they're the same shared foot. The only way out of this kind of mess is by doing a lot of tedious refactoring. Singletons aren't inherently evil; just be sure you really need them before you go ahead with this design pattern.

Where to Go from Here

You've only scratched the surface of a set of frameworks that have been in active development since the 1980s. You've seen some of the most important parts of that framework, but you've certainly not seen anywhere near all of it.

The *Cocoa Fundamentals Guide* in the developer documentation (page 80) is a great place to start learning more about this essential framework. Just search for its name in the documentation viewer. This book's appendix shows you books, websites, and other resources for learning more about Cocoa and Cocoa Touch.

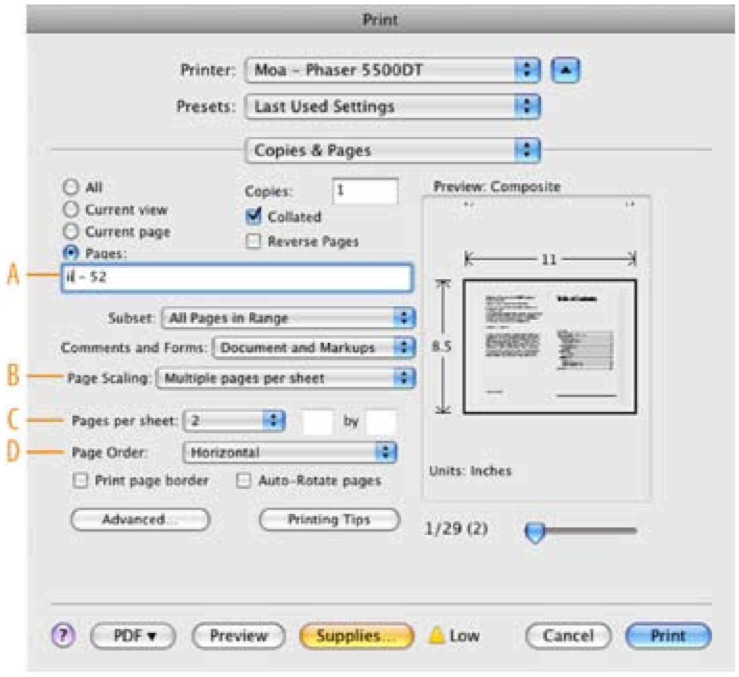
As a new developer, make sure you also take advantage of the source code that Apple provides on the developer site. Studying these examples can teach you a lot about the right way to build an iPhone application. In a similar vein, the appendix contains links to many open source projects. Some very talented developers have contributed this code to the iPhone community, and you can learn from their work.

Colophon

Nan Barber was the Editor for *Creating iPhone Apps with Cocoa Touch: The Mini Missing Manual*. Nellie McKesson was the Production Editor.

Nellie McKesson designed the interior layout, based on a series design by Phil Simpson. The text font for the PDF version of this book is Myriad Pro; and the heading and note font is Adobe Formata.

For best printing results of the PDF version of this book, use the following settings in the Adobe Reader Print dialog box: A: Pages: ii–[last page number]; B: Page Scaling: Mulitple pages per sheet; C: Pages per sheet: 2; D: Page Order: Horizontal.



Printer: Moa – Phaser 5500DT
Presets: Last Used Settings
Copies: 1
Collated ✓
Pages: ii - 52
Subset: All Pages in Range
Comments and Forms: Document and Markups
Page Scaling: Multiple pages per sheet
Pages per sheet: 2 by
Page Order: Horizontal