SAMPLE CHAPTER

Objective-C Fundamentals

Christopher K. Fairbairn
Johannes Fahrenkrug
Collin Ruffenach

MANNING

Objective-C Fundamentals

by Christopher K. Fairbairn
Johannes Fahrenkrug
Collin Ruffenach

Chapter 1

brief contents

1 Building your first iOS application

This chapter covers

- Understanding the iOS development environment
- Learning how to use Xcode and Interface Builder
- Building your first application

As a developer starting out on the iOS platform, you're faced with learning a lot of new technologies and concepts in a short period of time. At the forefront of this information overload is a set of development tools you may not be familiar with and a programming language shaped by a unique set of companies and historical events.

iOS applications are typically developed in a programming language called Objective-C and supported by a support library called Cocoa Touch. If you've already developed Mac OS X applications, you're probably familiar with the desktop cousins of these technologies. But it's important to note that the iOS versions of these tools don't provide exactly the same capabilities, and it's important to learn the restrictions, limitations, and enhancements provided by the mobile device. In some cases, you may even need to unlearn some of your desktop development practices.

While developing iOS applications, most of your work will be done in an application called *Xcode*. Xcode 4, the latest version of the IDE, has Interface Builder (for creating the user interface) built directly into it. Xcode 4 enables you to create, manage, deploy, and debug your applications throughout the entire software development lifecycle. When creating an application that supports more than one type of device powered by the iOS, you may wish to present slightly different user interfaces for specific device types while powering all variants via the same core application logic underneath. Doing so is easier if the concept of model-view-controller separation is used, something that Xcode 4 can help you with.

This chapter covers the steps required to use these tools to build a small game for the iPhone, but before we dive into the technical steps, let's discuss the background of the iOS development tools and some of the ways mobile development differs from desktop and web-based application development.

1.1 Introducing the iOS development tools

Objective-C is a strict superset of the procedural-based C programming language. This fact means that any valid C program is also a valid Objective-C program (albeit one that doesn't make use of any Objective-C enhancements).

Objective-C extends C by providing object-oriented features. The object-oriented programming model is based on sending messages to objects, which is different from the model used by C++ and Java, which call methods directly on an object. This difference is subtle but is also one of the defining features that enables many of Objective-C's features that are typically more at home in a dynamic language such as Ruby or Python.

A programming language, however, is only as good as the features exposed by its support libraries. Objective-C provides syntax for performing conditional logic and looping constructs, but it doesn't provide any inherent support for interacting with the user, accessing network resources, or reading files. To facilitate this type of functionality without requiring it to be written from scratch for each application, Apple includes in the SDK a set of support libraries collectively called *Cocoa Touch*. If you're an existing Java or .NET developer, you can view the Cocoa Touch library as performing a purpose similar to the Java Class Library or .NET's Base Class Libraries (BCL).

1.1.1 Adapting the Cocoa frameworks for mobile devices

Cocoa Touch consists of a number of frameworks (commonly called *kits*). A framework is a collection of classes that are grouped together by a common purpose or task. The two main frameworks you use in iPhone applications are Foundation Kit and UIKit. Foundation Kit is a collection of nongraphical system classes consisting of data structures, networking, file IO, date, time, and string-handling functions, and UIKit is a framework designed to help develop GUIs with rich animations.

Cocoa Touch is based on the existing Cocoa frameworks used for developing desktop applications on Mac OS X. But rather than making Cocoa Touch a direct line-by-line

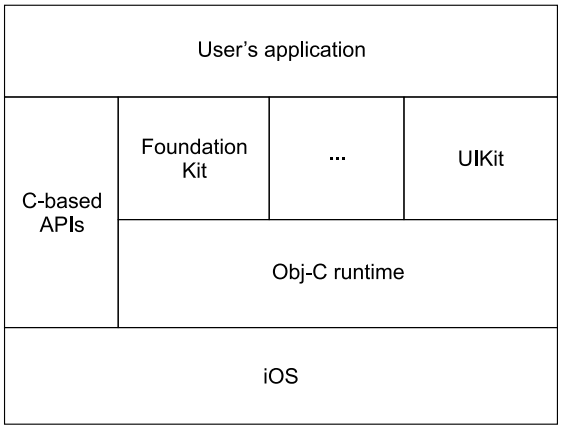

Figure 1.1 The software runtime environment for iOS applications, showing the operating system, Objective-C runtime, and Cocoa Touch framework layers

port to the iPhone, Apple optimized the frameworks for use in iPhone and iPod Touch applications. Some Cocoa frameworks were even replaced entirely if Apple thought improvements in functionality, performance, or user experience could be achieved in the process. UIKit, for example, replaced the desktop-based AppKit framework.

The software runtime environment for native iOS applications is shown in figure 1.1. It's essentially the same software stack for desktop applications if you replace iOS with Mac OS X at the lowest level and substitute some of the frameworks in the Cocoa layer.

Although the Cocoa Touch frameworks are Objective-C–based APIs, the iOS development platform also enables you to access standard C-based APIs. The ability to reuse C (or C++) libraries in your Objective-C applications is quite powerful. It enables you to reuse existing source code you may have originally developed for other mobile platforms and to tap many powerful open source libraries (license permitting), meaning you don't need to reinvent the wheel. As an example, a quick search on Google will find existing C-based source code for augmented reality, image analysis, and barcode detection, to name a few possibilities, all of which are directly usable by your Objective-C application.

1.2 Adjusting your expectations

With a development environment that will already be familiar to existing Mac OS X developers, you may mistakenly think that the iPhone is just another miniature computing device, similar to any old laptop, tablet, or netbook. That idea couldn't be any further from the truth. An iPhone is more capable than a simple cell phone but less so than a standard desktop PC. As a computing device, it fits within a market space similar to that of netbooks, designed more for casual and occasional use throughout the day in a variety of situations and environments than for sustained periods of use in a single session.

1.2.1 A survey of hardware specifications, circa mid-2011

On taking an initial look at an iPhone 4, you'll undoubtedly notice the 3.5-inch screen, 960 x 640 pixels, that virtually dominates the entire front of the device. Its general size and the fact that the built-in touch screen is the only way for users to interact with the device can have important ramifications on application design. Although 960 x 640 is larger than many cell phones, it probably isn't the screen on which to view a 300-column-by-900-row spreadsheet.

As an example of the kind of hardware specifications you can expect to see, table 1.1 outlines the specifications of common iPhone, iPod Touch, and iPad models available in mid-2010. In general, the hardware specifications lag behind those of desktop PCs by a couple of years, but the number of integrated hardware accessories that your applications can take advantage of, such as camera, Bluetooth, and GPS, is substantially higher.

Table 1.1 Comparison of hardware specifications of various iPhone and iPod Touch devices

Feature	iPhone 3G	iPhone 3GS	iPhone 4	iPad	iPad2
RAM	128 MB	256 MB	512 MB	256 MB	512 MB
Flash	8–16 GB	16–32 GB	16–32 GB	16–64 GB	16–64 GB
Processor	412 MHz ARM11	600 MHz ARM Cortex	1 GHz Apple A4	1 GHz Apple A4	1 GHz dual-core Apple A5
Cellular	3.6 Mbps	7.2 Mbps	7.2 Mbps	7.2 Mbps (optional)	7.2 Mbps (optional)
Wi-Fi	Yes	Yes	Yes	Yes	Yes
Camera	2 MP	3 MP AF	5 MP AF (back) 0.3 MP (front)	—	0.92 MP (back) 0.3 MP (front)
Bluetooth	Yes	Yes	—	Yes	Yes
GPS	Yes (no compass)	Yes	—	Yes (3G models only)	Yes (3G models only)

Although it's nice to know the hardware capabilities and specifications of each device, application developers generally need not concern themselves with the details. New models will come and go as the iOS platform matures and evolves until it becomes difficult to keep track of all the possible variants.

Instead, you should strive to create an application that will adapt at runtime to the particular device it finds itself running on. Whenever you need to use a feature that's present only on a subset of devices, you should explicitly test for its presence and programmatically deal with it when it isn't available. For example, instead of checking if your application is running on an iPhone to determine if a camera is present, you would be better off checking whether a camera is present, because some models of iPad now come with cameras.

1.2.2 Expecting an unreliable internet connection

In this age of cloud computing, a number of iOS applications need connectivity to the internet. The iOS platform provides two main forms of wireless connectivity: local area in the form of 802.11 Wi-Fi and wide area in the form of various cellular data standards. These connection choices provide a wide variability in speed, ranging from 300 kilobits to 54 megabits per second. It's also possible for the connection to disappear altogether, such as when the user puts the device into flight mode, disables cellular roaming while overseas, or enters an elevator or tunnel.

Unlike on a desktop, where most developers assume a network connection is always present, good iOS applications must be designed to cope with network connectivity being unavailable for long periods of time or unexpectedly disconnecting. The worst user experience your customers can have is a "sorry, cannot connect to server" error message while running late to a meeting and needing to access important information that shouldn't require a working internet connection to obtain.

In general, it's important to constantly be aware of the environment in which your iOS application is running. Your development techniques may be shaped not only by the memory and processing constraints of the device but also by the way in which the user interacts with your application.

That's enough of the background information. Let's dive right in and create an iOS application!

1.3 Using Xcode to develop a simple Coin Toss game

Although you might have grand ideas for the next iTunes App Store smash, let's start with a relatively simple application that's easy to follow without getting stuck in too many technical details, allowing the unique features of the development tools to shine through. As the book progresses, we dig deeper into the finer points of everything demonstrated. For now the emphasis is on understanding the general process rather than the specifics of each technique.

The application you develop here is a simple game that simulates a coin toss, such as is often used to settle an argument or decide who gets to go first in a competition. The user interface is shown in figure 1.2 and consists of two buttons labeled Heads and Tails. Using these buttons, the user can request that a new coin toss be made and call the desired result. The iPhone simulates the coin toss and updates the screen to indicate if the user's choice is correct.

In developing this game, the first tool we need to investigate is Xcode.

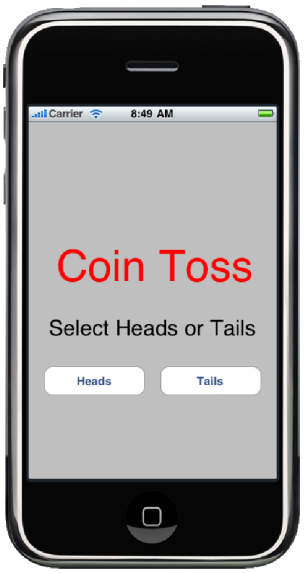

Figure 1.2 Coin Toss sample game

1.3.1 Introducing Xcode—Apple's IDE

As mentioned earlier in this chapter, Xcode is an IDE that provides a comprehensive set of features to enable you to manage the entire lifecycle of your software development project. Creating the initial project, defining your class or data model, editing your source code, building your application, and finally debugging and performance-tuning the resultant application are all tasks performed in Xcode.

Xcode is built on the foundation of several open source tools: LLVM (the open source Low-Level Virtual Machine), GCC (the GNU compiler), GDB (the GNU debugger), and DTrace (instrumentation and profiling by Sun Microsystems).

1.3.2 Launching Xcode easily

Once you install the iOS software development kit (SDK), the first challenge to using Xcode is locating the application. Unlike most applications that install in the/Applications folder, Apple separates developer-focused tools into the/Developer/Applications folder.

The easiest way to find Xcode is to use the Finder to open the root Macintosh HD folder (as shown in figure 1.3). From there, you can drill down into the Developer folder and finally the Applications subfolder. As a developer, you'll practically live within Xcode, so you may wish to put the Xcode icon onto your Dock or place the folder in the Finder sidebar for easy access.

Once you locate the/Developer/Applications folder, you should be able to easily locate and launch Xcode.

It's important to note that Xcode isn't your only option. Xcode provides all the features you require to develop applications out of the box, but that doesn't mean you can't complement it with your own tools. For example, if you have a favorite text editor in which you feel more productive, it's possible to configure Xcode to use your external text editor in favor of the built-in functionality. The truly masochistic among you could even revert to using makefiles and the command line.

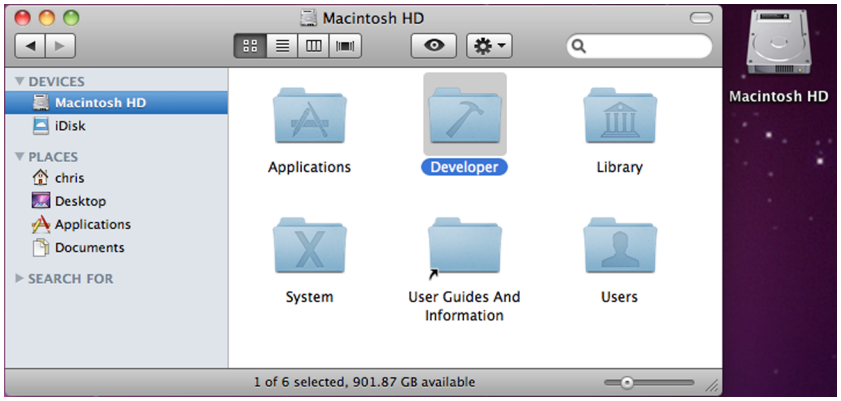

Figure 1.3 A Finder window showing the location of the Developer folder, which contains all iPhone developer–related tools and documentation

Help! I don't see the Xcode application

If you don't have a/Developer folder or you can't see any references to iPhone or iPad project templates when Xcode is launched, refer to appendix A for help on how to download and install the required software.

1.3.3 Creating the project

To create your first project, select the New Project option in the File menu (Shift-Cmd-N). Xcode displays a New Project dialog similar to the one displayed in figure 1.4.

Your first decision is to choose the type of project you want to create. This is done by selecting a template that determines the type of source code and settings Xcode will automatically add to get your project started.

For the Coin Toss game, you want the View-based Application template. You first select Application under the iOS header in the left pane, and then select View-based Application. Then click Next in the lower-right corner, which prompts you to name the project and allows you to specify the company identifier required to associate the application with your iOS Developer account. For this project, use the name CoinToss and enter a suitable company identifier.

Xcode uses the product name and company identifier values to produce what is called a *bundle identifier*. iOS uniquely identifies each application by this string. In

Figure 1.4 The New Project dialog in Xcode showing the View-based Application template

order for the operating system to allow the CoinToss game to run, its bundle identifier must match up with one included in a provisioning profile that's been installed on the device. If the device can't find a suitable profile, it refuses to run the application. This is how Apple controls with an iron fist which applications are allowed in its ecosystem. If you don't have a suitable company identifier or are unsure what to enter here, follow the instructions in appendix A before proceeding with the rest of this chapter.

Once all the details are entered, deselect the Include Unit Tests check box and click Next, which prompts you to select where you want the project and generated source code files to be saved.

Help! I don't see any iOS-related options

If you see no iOS-based templates in the New Project dialog, it's possible you haven't correctly installed the iOS SDK. The copy of Xcode you're running is probably from a Mac OS X Install DVD or perhaps was downloaded directly from the Apple Developer Connection (ADC) website and is suitable only for development of desktop applications.

Installing the iOS SDK as outlined in appendix A should replace your copy of Xcode with an updated version that includes support for iPhone and iPad development.

You may wonder what other kinds of projects you can create. Table 1.2 lists the most common iOS project templates. Which template you choose depends on the type of user interface you want your application to have. But don't get too hung up on template selection: the decision isn't as critical as you may think. Once your project is created,

Table 1.2 Project templates available in Xcode for creating a new iOS project

Project type	Description
Navigation-based Application	Creates an application similar in style to the built-in Contacts application with a navigation bar across the top.
OpenGL ES Application	Creates an OpenGL ES–based graphics application suitable for games and so on.
Split View–based Application	Creates an application similar in style to the built-in Mail application on the iPad. Designed to display master/detail-style information in a single screen.
Tab Bar Application	Creates an application similar in style to the built-in Clock application with a tab bar across the bottom.
Utility Application	Creates an application similar in style to the built-in Stocks and Weather applications, which flip over to reveal a second side.
View-based Application	Creates an application that consists of a single view. You can draw and respond to touch events from the custom view.
Window-based Application	Creates an application that consists of a single window onto which you can drag and drop controls.

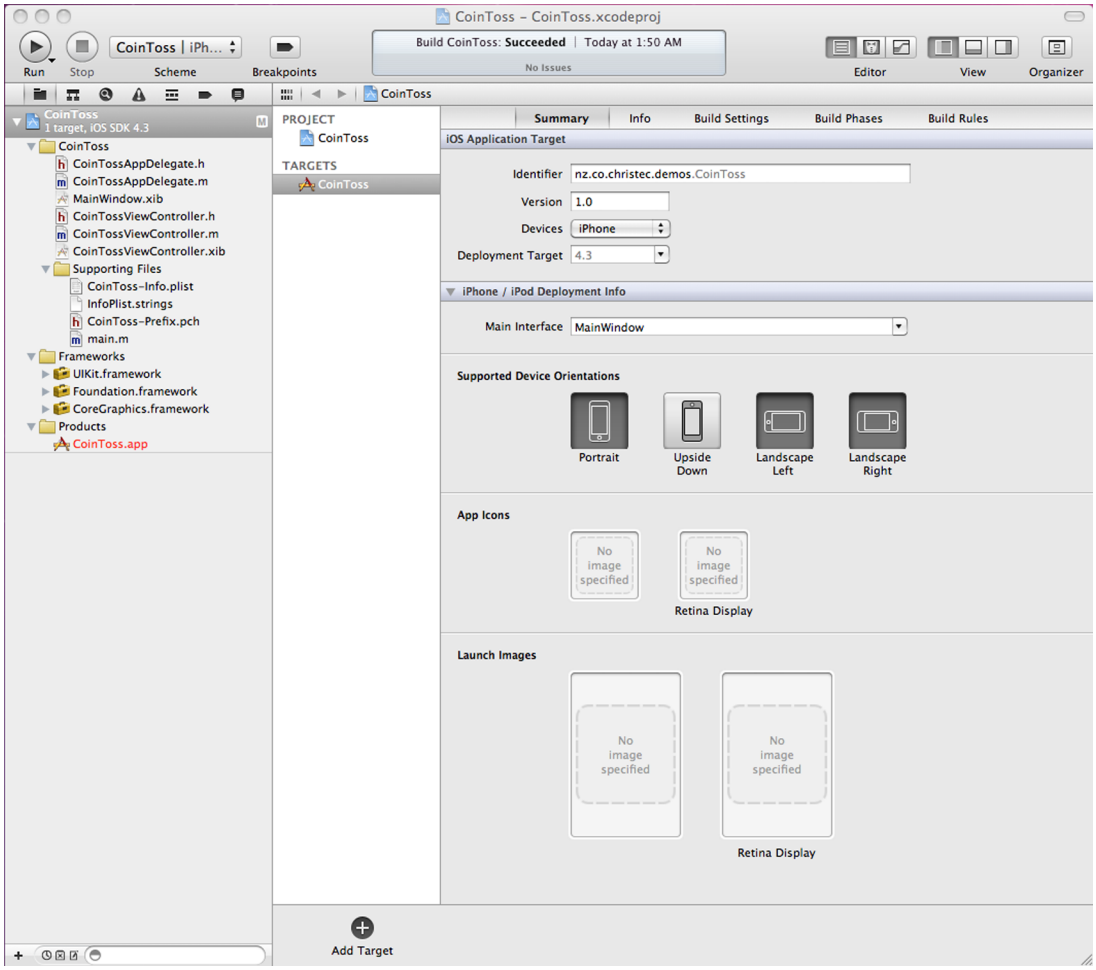

Figure 1.5 Main Xcode window with the CoinToss group fully expanded to show the project's various source code files

you can alter the style of your application—it just won't be as easy because you won't have the project template automatically inserting all of the required source code for you; you'll need to write it yourself.

Now that you've completed the New Project dialog, a project window similar to the one in figure 1.5 is displayed. This is Xcode's main window and consists of a Project Navigator pane on the left and a large, context-sensitive editor pane on the right.

The pane on the left lists all the files that make up your application. The group labeled CoinToss represents the entire game, and if you expand this node, you can drill down into smaller subgroups until you eventually reach the files that make up the project. You're free to create your own groupings to aid in organizing the files in any manner that suits you.

When you click a file in the left pane, the right pane updates to provide an editor suitable for the selected file. For *.h and *.m source code files, a traditional source code text editor is presented, but other file types (such as *.xib resource files) have more complex graphical editors associated with them.

Some groups in the left pane have special behaviors associated with them or don't represent files at all. For example, the items listed under the Frameworks group indicate pre-compiled code libraries that the current project makes use of.

As you become more comfortable with developing applications in Xcode, you'll become adept at exploring the various sections presented in the Project Navigator pane. To begin your discovery, let's write the source code for your first class.

1.3.4 Writing the source code

The View-based Application template provides enough source code to get a basic game displayed on the iPhone—so basic, in fact, that if you ran the game right now, you would simply see a gray rectangle on the screen.

Let's start implementing the game by opening the CoinTossViewController.h file in the Xcode window and using the text editor to replace the contents with the following listing.

Listing 1.1 CoinTossViewController.h

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

@interface CoinTossViewController : UIViewController {
  UILabel *status;
  UILabel *result;
}

@property (nonatomic, retain) IBOutlet UILabel *status;
@property (nonatomic, retain) IBOutlet UILabel *result;

- (IBAction)callHeads;
- (IBAction)callTails;

@end
```

Don't worry if the contents of listing 1.1 don't make much sense to you. At this stage, it's not important to understand the full meaning of this code. Learning these sorts of details is what the rest of the book is designed for—all will be revealed in time!

For now, let's focus on understanding the general structure of an Objective-C–based project. Objective-C is an object-oriented language, meaning that a large portion of your coding efforts will be spent defining new classes (types of objects). Listing 1.1 defines a new class called `CoinTossViewController`. By convention, the definition of a class is kept in a header file that uses a *.h file extension.

In the `CoinTossViewController` header file, the first two lines declare that the class stores the details of two `UILabel` controls located somewhere in the user interface. A `UILabel` can display a single line of text, and you use these labels to display the results of the coin toss.

The second set of statements allows code external to this class to tell you which specific `UILabels` you should be using. Finally, you specify that your class responds to two messages called `callHeads` and `callTails`. These messages inform you when the user has called heads or tails and a new coin toss should be initiated.

A header (*.h) file specifies what you can expect a class to contain and how other code should interact with it. Now that you've updated the header file, you must provide the actual implementation of the features you've specified. Open the matching CoinTossViewController.m file, and replace its contents with that of the following listing.

Listing 1.2 CoinTossViewController.m

```
#import "CoinTossViewController.h"
#import <QuartzCore/QuartzCore.h>

@implementation CoinTossViewController

@synthesize status, result;

- (void) simulateCoinToss:(BOOL)userCalledHeads {
  BOOL coinLandedOnHeads = (arc4random() % 2) == 0;

  result.text = coinLandedOnHeads ? @"Heads" : @"Tails";

  if (coinLandedOnHeads == userCalledHeads)
    status.text = @"Correct!";
  else
    status.text = @"Wrong!";

  CABasicAnimation *rotation = [CABasicAnimation
     animationWithKeyPath:@"transform.rotation"];
  rotation.timingFunction = [CAMediaTimingFunction
     functionWithName:kCAMediaTimingFunctionEaseInEaseOut];
  rotation.fromValue = [NSNumber numberWithFloat:0.0f];
  rotation.toValue = [NSNumber numberWithFloat:720 * M_PI / 180.0f];
  rotation.duration = 2.0f;
  [status.layer addAnimation:rotation forKey:@"rotate"];

  CABasicAnimation *fade = [CABasicAnimation
     animationWithKeyPath:@"opacity"];
  fade.timingFunction = [CAMediaTimingFunction
     functionWithName:kCAMediaTimingFunctionEaseInEaseOut];
  fade.fromValue = [NSNumber numberWithFloat:0.0f];
  fade.toValue = [NSNumber numberWithFloat:1.0f];
  fade.duration = 3.5f;
  [status.layer addAnimation:fade forKey:@"fade"];
}

- (IBAction) callHeads {
  [self simulateCoinToss:YES];
}

- (IBAction) callTails {
  [self simulateCoinToss:NO];
}

- (void) viewDidUnload {
  self.status = nil;
```

1 **Match with @property**

2 **Set up two objects**

3 **Affect the label**

```
  self.result = nil;
}

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

@end
```

❹ **Memory management**

Listing 1.2 at first appears long and scary looking, but when broken down into individual steps, it's relatively straightforward to understand.

The first statement ❶ matches up with the `@property` declarations in CoinTossViewController.h. The concept of properties and the advantage of synthesized ones in particular are explored in depth in chapter 5.

Most of the logic in the CoinTossViewController.m file is contained in the `simulateCoinToss:` method, which is called whenever the user wants the result of a new coin toss. The first line simulates a coin toss by generating a random number between `0` and `1` to represent heads and tails respectively. The result is stored in a variable called `coinLandedOnHeads`.

Once the coin toss result is determined, the two `UILabel` controls in the user interface are updated to match. The first conditional statement updates the result label to indicate if the simulated coin toss landed on heads or tails; the second statement indicates if the user correctly called the coin toss.

The rest of the code in the `simulateCoinToss:` method sets up two `CABasicAnimation` objects ❷, ❸ to cause the label displaying the status of the coin toss to spin into place and fade in over time rather than abruptly updating. It does this by requesting that the `transform.rotation` property of the `UILabel` control smoothly rotate from `0` degrees to `720` degrees over `2.0` seconds, while the `opacity` property fades in from 0% (`0.0`) to 100% (`1.0`) over `3.5` seconds. It's important to note that these animations are performed in a declarative manner. You specify the change or effect you desire and leave it up to the framework to worry about any timing- and redrawing-related logic required to implement those effects.

The `simulateCoinToss:` method expects a single parameter called `userCalledHeads`, which indicates if the user expects the coin toss to result in heads or tails. Two additional methods, `callHeads` and `callTails`, are simple convenience methods that call `simulateCoinToss:` with the `userCalledHeads` parameter set as expected.

The final method, called `dealloc` ❹, deals with memory management–related issues. We discuss memory management in far greater depth in chapter 9. The important thing to note is that Objective-C doesn't automatically garbage collect unused memory (at least as far as the current iPhone is concerned). This means if you allocate memory or system resources, you're also responsible for releasing (or deallocating) it. Not doing so will cause your application to artificially consume more resources than it needs, and in the worst case, you'll exhaust the device's limited resources and cause the application to crash.

Now that you have the basic logic of the game developed, you must create the user interface in Xcode and connect it back to the code in the `CoinTossViewController` class.

1.4 Hooking up the user interface

At this stage, you can determine from the `CoinTossViewController` class definition that the user interface should have at least two `UILabel` controls and that it should invoke the `callHeads` or `callTails` messages whenever the user wants to call the result of a new coin toss. You haven't yet specified where on the screen the labels should be positioned or how the user requests that a coin toss be made.

There are two ways to specify this kind of detail. The first is to write source code that creates the user interface controls, configures their properties such as font size and color, and positions them onscreen. This code can be time consuming to write, and you can spend a lot of your time trying to visualize how things look onscreen.

A better alternative is to use Xcode, which allows you to visually lay out and configure your user interface controls and connect them to your source code. Most iOS project templates use this technique and typically include one or more *.xib files designed to visually describe the user interface. This project is no exception, so click the CoinTossViewController.xib file in the Project Navigator pane and notice that the editor pane displays the contents of the file (figure 1.6).

Along the left edge of the editor pane are some icons. Each icon represents an object that's created when the game runs, and each has a tooltip that displays it name. The wireframe box labeled File's Owner represents an instance of the `CoinTossViewController` class; the white rectangle represents the main view (or screen) of the application. Using Xcode, you can graphically configure the properties of these objects and create connections between them.

1.4.1 Adding controls to a view

The first step in defining the user interface for your game is to position the required user interface controls onto the view.

To add controls, find them in the Library window, which contains a catalog of available user interface controls, and drag and drop them onto the view. If the Library window isn't visible, you can open it via the View > Utilities > Object Library menu option (Control-Option-Cmd-3). For the Coin Toss game, you require two Labels and two Rounded Rect Buttons, so drag two of each control onto the view. The process of dragging and dropping a control onto the view is shown in figure 1.7.

After you drag and drop the controls onto the view, you can resize and adjust their positions to suit your aesthetics. The easiest way to change the text displayed on a button or label control is to double-click the control and begin typing. To alter other properties, such as font size and color, you can use the Attributes Inspector pane, which can be displayed via the View > Utilities > Attributes Inspector menu option (Alt-Cmd-4). While styling your view, you can refer back to figure 1.2 for guidance.

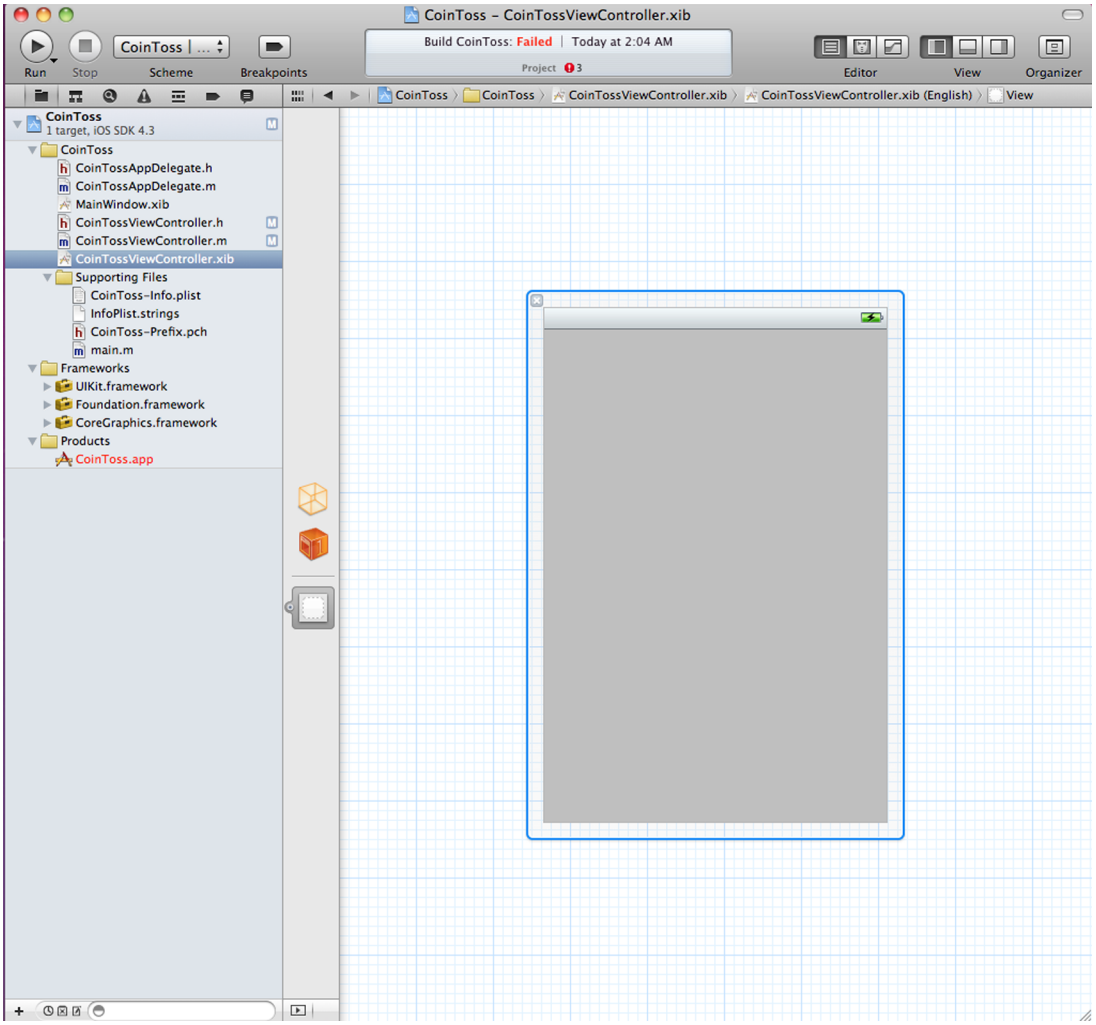

Figure 1.6 The main Xcode window demonstrating the editing of a *.xib file. Along the left edge of the editor you can see three icons, each representing a different object or GUI component stored in the .xib file.

With the controls positioned on the user interface, the only task left is to connect them to the code you previously wrote. Remember that the class defined in the CoinTossViewController.h header file requires three things from the user interface:

- Something to send the `callHeads` or `callTails` messages whenever the user wishes to initiate a new coin toss
- A `UILabel` to display the results of the latest coin toss (heads or tails)
- A `UILabel` to display the status of the latest coin toss (correct or incorrect)

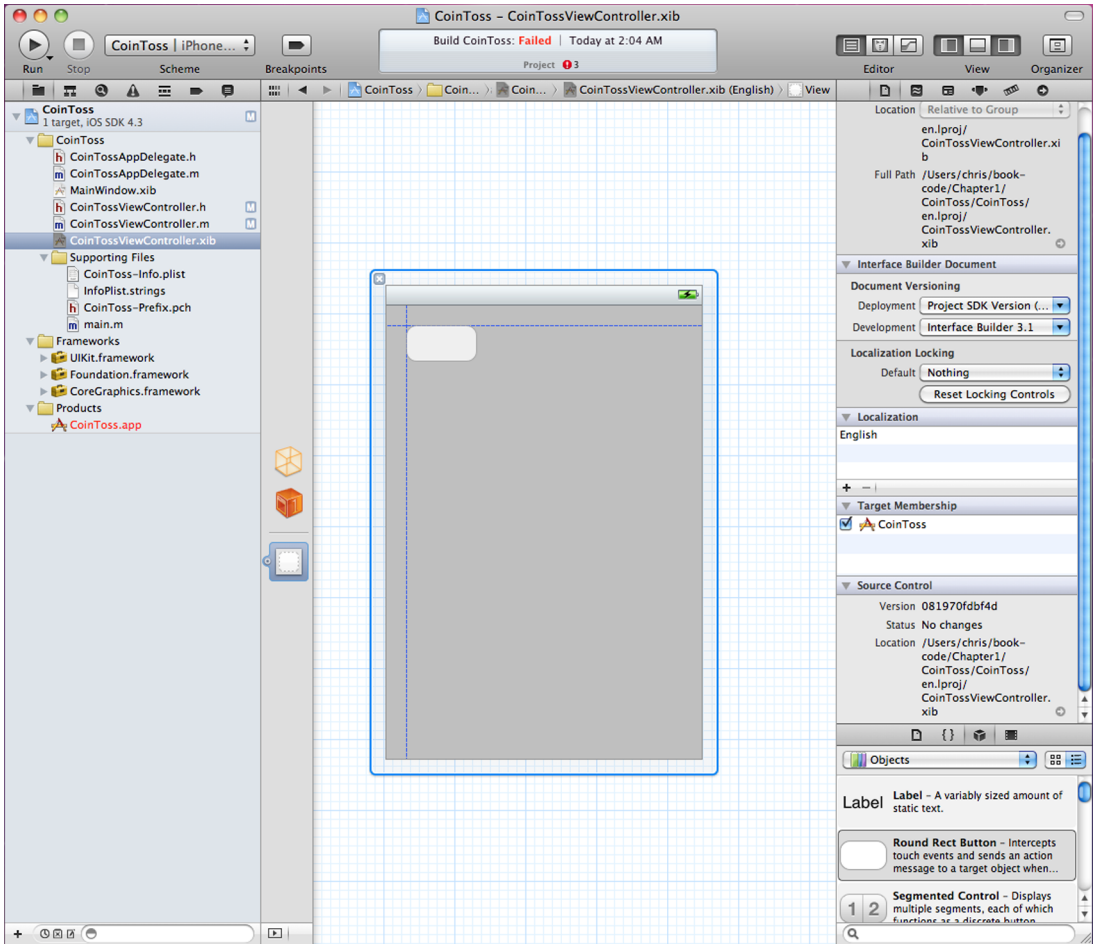

Figure 1.7 Dragging and dropping new controls onto the view. Notice the snap lines, which help ensure your user interface conforms to the iOS Human Interface Guidelines (HIG).

1.4.2 Connecting controls to source code

The user interface you just created meets these requirements, but the code can't determine which button should indicate that the user calls heads or tails (even if the text on the buttons makes it inherently obvious to a human). Instead, you must explicitly establish these connections. Xcode allows you to do so graphically.

Hold down the Control key and drag the button labeled Heads toward the icon representing the `CoinTossViewController` instance (File's Owner) located on the left edge of the editor. As you drag, a blue line should appear between the two elements.

When you let go of the mouse, a pop-up menu appears that allows you to select which message should be sent to the `CoinTossViewController` object whenever the

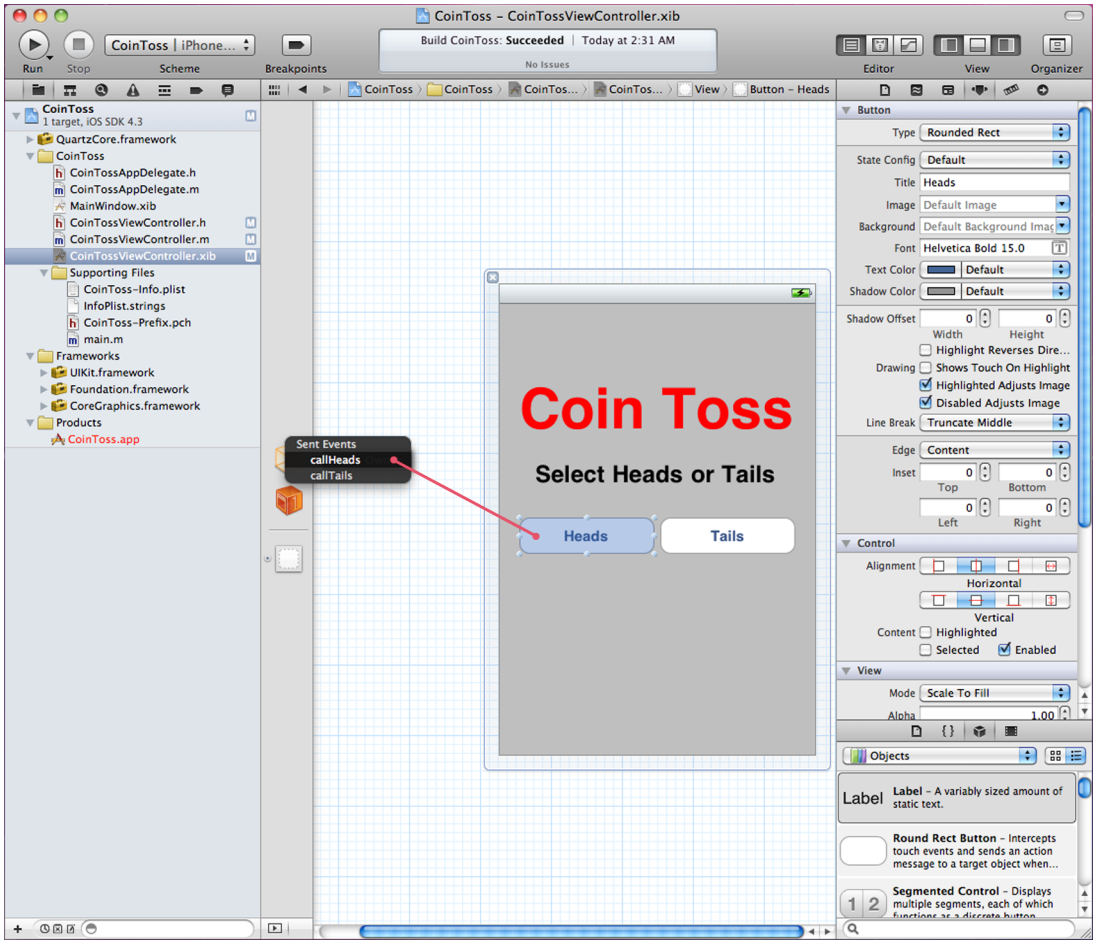

Figure 1.8 Visually forming a connection between the button control and the `CoinTossViewController` class by dragging and dropping between items

button is tapped, as shown in figure 1.8. In this case, you select `callHeads` because this is the message that matches the intent of the button.

You can repeat this process to connect the button labeled Tails to the `callTails` method. Making these two connections means that tapping either of the buttons in the user interface will cause the execution of logic in the `CoinTossViewController` class. Having these connections specified graphically rather than programmatically is a flexible approach because it enables you to quickly and easily try out different user interface concepts by swapping controls around and reconnecting them to the class.

If Xcode refuses to make a connection between a user interface control and an object, the most probable cause is a source code error, such as a simple typo or incorrect data type. In this case, make sure the application still compiles, and correct any errors that appear before retrying the connection.

With the buttons taken care of, you're left with connecting the label controls to the `CoinTossViewController` class to allow the code to update the user interface with the results of the latest coin toss.

To connect the label controls, you can use a similar drag-and-drop operation. This time, while holding down the Control key, click the icon representing the `CoinTossViewController` instance and drag it toward the label in the view. When you release the mouse, a pop-up menu appears that allows you to select which property of the `CoinTossViewController` class you want to connect the label control to. This process is demonstrated in figure 1.9. Using this process, connect the label titled Coin Toss to the `status` property and the label titled Select Heads or Tails to the `result` property.

When deciding which way you should form connections between objects, consider the flow of information. In the case of the button, tapping the button causes a method

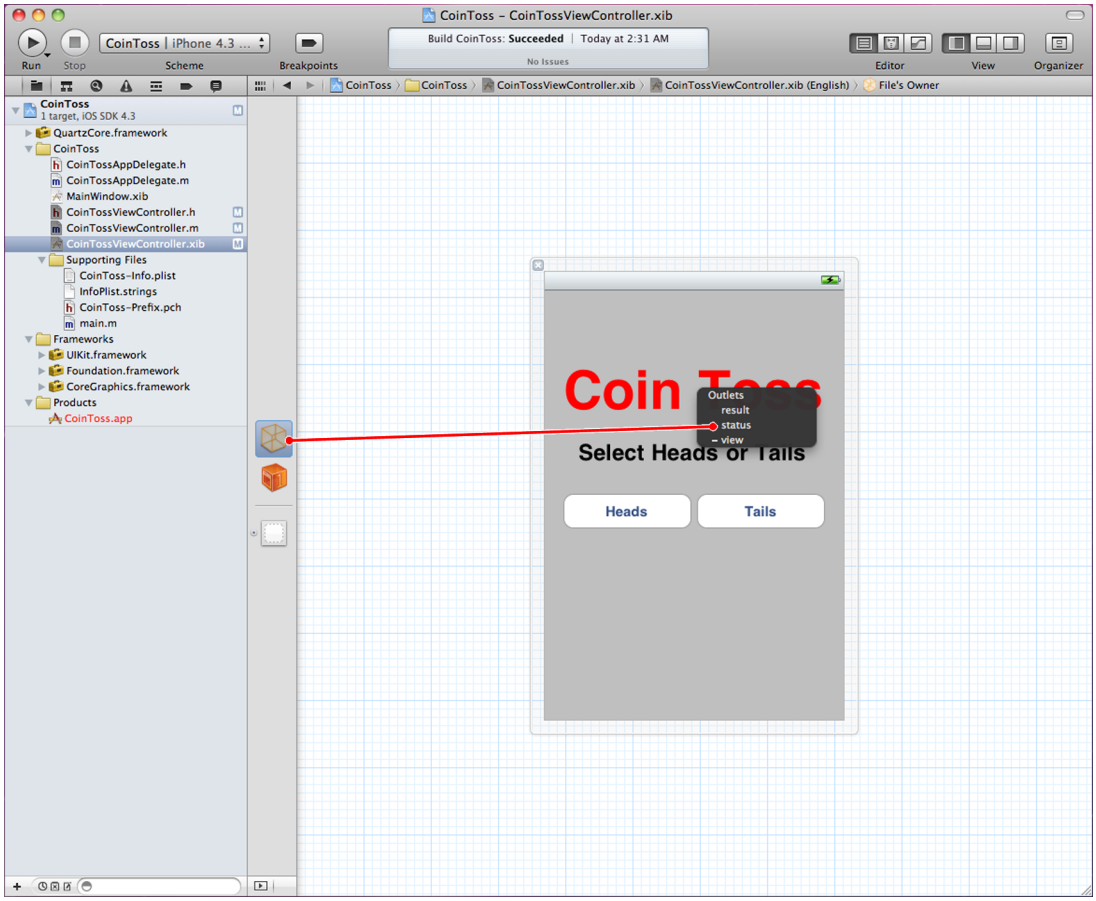

Figure 1.9 Visually forming a connection between the status instance variable and the label control in the user interface by dragging and dropping between the items (with the Control key held down)

in the application to be executed, whereas in the case of connecting the label, changing the value of the instance variable in the class should update the user interface.

You may wonder how Xcode determines which items to display in the pop-up menu. If you refer back to listing 1.1, the answer can be seen by way of the special `IBOutlet` and `IBAction` keywords. Xcode parses your source code and allows you to connect the user interface to anything marked with one of these special attributes.

At this stage, you may like to verify that you've correctly made the required connections. If you hold down the Control key and click the icon representing the `CoinTossViewController` instance, a pop-up menu appears allowing you to review how all the outlets and actions associated with an object are connected. If you hover the mouse over one of the connections, Xcode even highlights the associated object. This feature is shown in figure 1.10.

At this stage you're done with the user interface. You're now ready to kick the tires, check if you've made mistakes, and see how well your game runs.

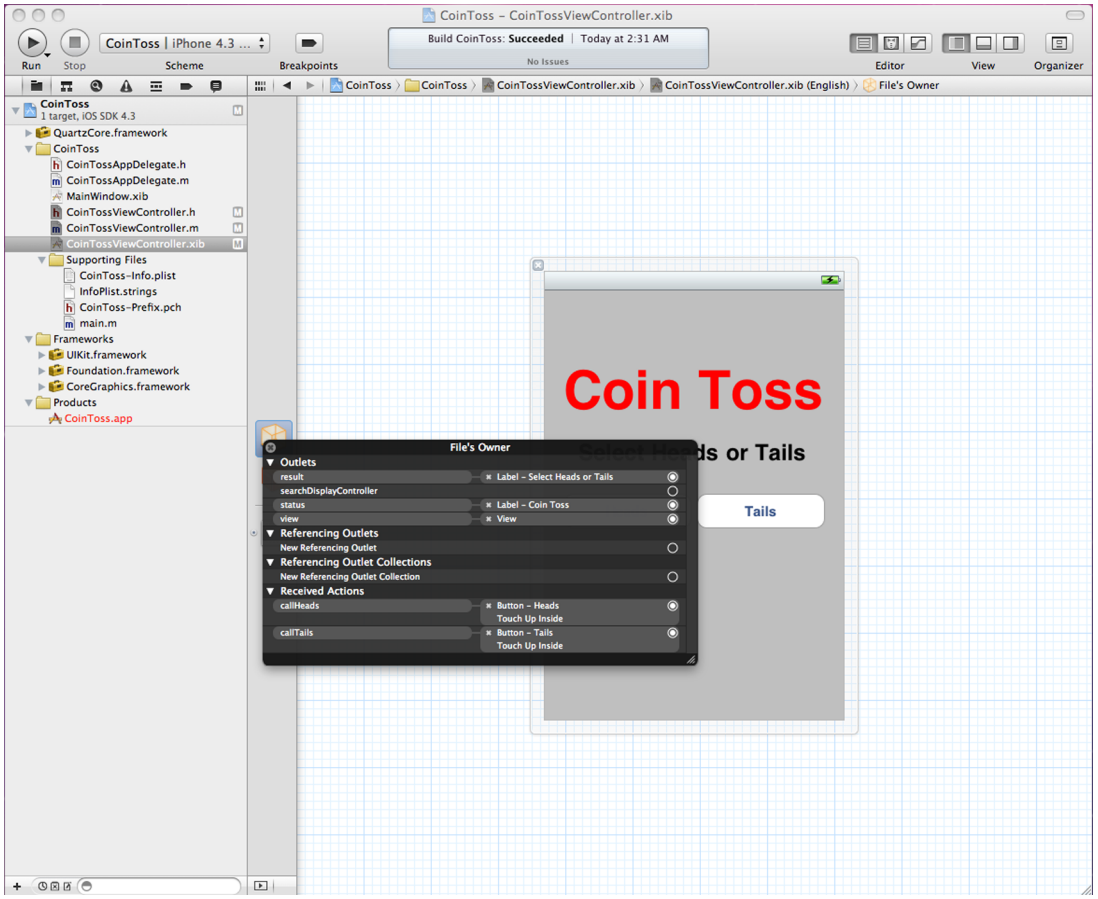

Figure 1.10 Reviewing connections made to and from the `CoinTossViewController` object

NIBs vs. XIBs

The user interface for an iOS application is stored in a .xib file. But in the documentation and Cocoa Touch frameworks, these files are commonly called *nib*s.

These terms are used pretty interchangeably: a .xib file uses a newer XML-based file format, which makes the file easier to store in revision control systems and so on.

A .nib, on the other hand, is an older binary format, which leads to more efficient file sizes, parsing speed, and so on.

The documentation commonly refers to NIB files instead of XIB files because, as Xcode builds your project, it automatically converts your *.xib files into the *.nib format.

1.5 Compiling the Coin Toss game

Now that you've finished coding your application, you need to convert the source code into a form useable by the iPhone. This process is called *compilation*, or *building* the project. To build the game, select Build from the Product menu (or press Cmd-B).

While the project is building, you can keep track of the compiler's progress by looking at the progress indicator in the middle of the toolbar. It should read "Build CoinToss: Succeeded." If you've made mistakes, you'll see a message similar to "Build CoinToss: Failed." In this case, clicking the red exclamation icon underneath the text (or pressing Cmd-4) displays a list of errors and warnings for you to resolve.

Clicking an error in this list displays the matching source code file with the lines containing errors highlighted, as illustrated in figure 1.11. After correcting the problem, you can build the application again, repeating this process until all issues are resolved.

When you compile the Coin Toss game, you should notice errors mentioning `kCAMediaTimingFunctionEaseInEaseOut`, `CAMediaTimingFunction`, and `CABasicAnimation`. To correct these errors, select the CoinToss project in the Project Navigator (topmost item in the tree view). In the editor that appears for this item, switch to the Build Phases tab and expand the Link Binary with Libraries section. The expanded region displays a list of additional frameworks that your application requires. For the user interface animations to work, you need to click the + button at the bottom of the window and select `QuartzCore.framework` from the list that appears.

To keep things tidy, once you add the QuartzCore framework reference, you may prefer to move it within the project navigator tree view so that it's located under the Frameworks section, alongside the other frameworks on which your application depends.

1.6 Taking Coin Toss for a test run

Now that you've compiled the game and corrected any obvious compilation errors, you're ready to verify that it operates correctly. You could run the game and wait for it to behave incorrectly or crash, but that would be rather slow going, and you would have to guess at what was happening internally. To improve this situation, Xcode provides

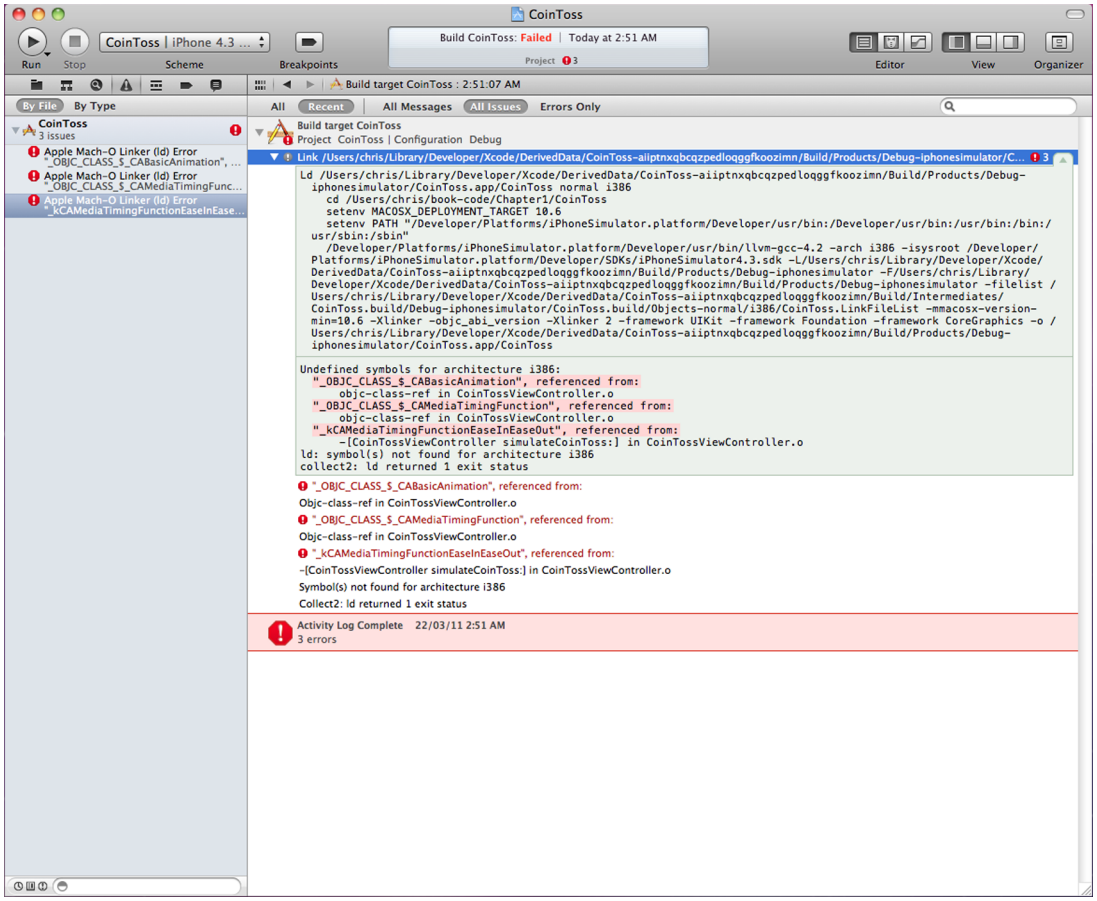

Figure 1.11 Xcode's text editor visually highlights lines of source code with compilation errors. After correcting any errors, building the project will indicate if you have successfully corrected the problem.

an integrated debugger that hooks into the execution of your application and allows you to temporarily pause it to observe the value of variables and step through source code line by line. But before you learn how to use it, we must take a slight detour.

1.6.1 Selecting a destination

Before testing your application, you must decide where you want to run it. During initial development, you'll commonly test your application via the iOS Simulator. The simulator is a pretend iPhone or iPad device that runs in a window on your desktop Mac OS X machine. Using the simulator can speed up application development because it's a lot quicker for Xcode to transfer and debug your application in the simulator than it is to work with a real iPhone.

Developers with experience in other mobile platforms may be familiar with the use of device emulators. The terms *simulator* and *emulator* aren't synonymous. Unlike an

> **Always test on a real iPhone, iPod Touch, or iPad device**
>
> The code samples in this book are designed to run in the iOS Simulator. This is a quick and easy way to iteratively develop your application without worrying about device connectivity or the delay involved in transferring the application to a real device.
>
> Because the iOS Simulator isn't a perfect replica of an iPhone, it's possible for an application to work in the simulator but fail on an actual device. Never publish an application to the iTunes App Store that hasn't been tested on a real device, or better yet, try to test your application out on a few variants, such as the iPhone and iPod Touch.

emulator that attempts to emulate the device at the hardware level (and hence can run virtually identical firmware to a real device), a simulator only attempts to provide an environment that has a compatible set of APIs.

The iOS Simulator runs your application on the copy of Mac OS X used by your desktop, which means that differences between the simulator and a real iPhone occasionally creep in. A simple example of where the simulation "leaks" is filenames. In the iOS Simulator, filenames are typically case insensitive, whereas on a real iPhone, they're case sensitive.

By default, most project templates are configured to deploy your application to the iOS Simulator. To deploy your application to a real iPhone, you must change the destination from iPhone Simulator to iOS Device. The easiest way to achieve this is to select the desired target in the drop-down menu found toward the left of the toolbar in the main Xcode window, as shown in figure 1.12.

Changing the destination to iOS Device ensures that Xcode attempts to deploy the application to your real iPhone, but an additional change is needed before this will succeed.

1.6.2 *Using breakpoints to inspect the state of a running application*

While testing an application, it's common to want to investigate the behavior of a specific section of source code. Before you launch the application, it can be handy to configure the debugger to automatically pause execution whenever these points are reached. You can achieve this through the use of a feature called *breakpoints*.

A breakpoint indicates to the debugger a point in the source code where the user would like to automatically "break into" the debugger to explore the current value of variables, and so on.

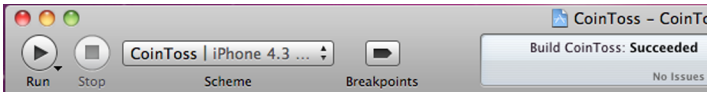

Figure 1.12 The top-left corner of the main Xcode window. Selecting the CoinToss | iPhone 4.3 Simulator drop-down menu allows you to switch between iPhone Simulator and iOS Device.

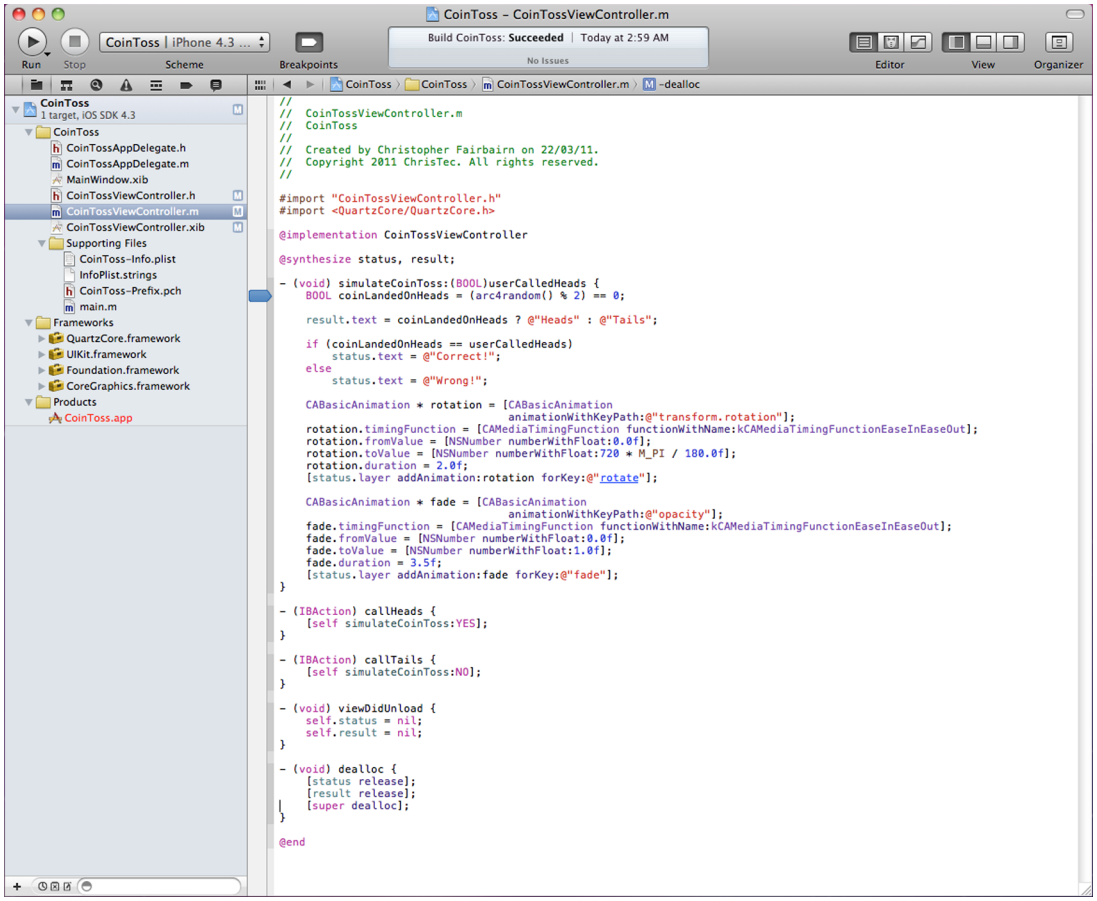

Figure 1.13 Setting a breakpoint to break into the debugger whenever the first line of the `simulateCoinToss:` method is called. Notice the arrow in the margin indicating an active breakpoint.

For the Coin Toss game, let's add a breakpoint to the start of the `simulateCoinToss:` method. Open the CoinTossViewController.m file and scroll down to the source code implementing the `simulateCoinToss:` method. If you then click the left margin beside the first line, you should see a little blue arrow appear, as shown in figure 1.13.

The blue arrow indicates that this line has an enabled breakpoint. If you click the breakpoint, it becomes a lighter shade of blue, indicating a disabled breakpoint, which causes the debugger to ignore it until it's clicked again to re-enable it. To permanently remove a breakpoint, click and drag the breakpoint away from the margin. Releasing the mouse will show a little "poof" animation, and the breakpoint will be removed.

1.6.3 *Running the CoinToss game in the iPhone simulator*

With the breakpoint in place, you're finally ready to run the application and see it in action. Select Run from the Product menu (Cmd-R). After a few seconds, the application

will appear on your iPhone. All that hard work has finally paid off. Congratulations—you're now officially an iPhone developer!

If you want to run the game but don't want any of your breakpoints to be enabled, you can click each one to disable them individually, but this would take a while, and you would need to manually re-enable all the breakpoints if you wanted to use them again. As a handy alternative, you can temporarily disable all breakpoints by selecting Product > Debug > Deactivate Breakpoints (Cmd-Y).

1.6.4 Controlling the debugger

Now that you've seen your first iPhone application running, you'll have undoubtedly felt the urge and tapped one of the buttons labeled Heads or Tails. When you tap a button, notice that the Xcode window jumps to the foreground. This is because the debugger has detected that execution of the application has reached the point where you inserted a breakpoint.

The Xcode window that appears should look similar to the one in figure 1.14. Notice that the main pane of the Xcode window displays the source code of the currently executing method. Hovering the mouse over a variable in the source code displays a data tip showing the variable's current value. The line of source code that's about to be executed is highlighted, and a green arrow in the right margin points at it.

While the debugger is running, you'll notice the left pane of the Xcode window switches to display the call stack of each thread in the application. The call stack lists the order in which currently executing methods have been called, with the current method listed at the top. Many of the methods listed will be gray, indicating that source code isn't available for them, in this case because most are internal details of the Cocoa Touch framework.

A new pane at the bottom of the screen is also displayed; it shows the current values of any variables and arguments relevant to the current position of the debugger as well as any textual output from the debugger (see figure 1.14).

Along the top of the bottom debug pane, you may notice a series of small toolbar buttons similar to those shown in figure 1.15.

These toolbar options enable you to control the debugger and become important when the debugger pauses the application or stops at a breakpoint. These toolbar buttons (which may not all be present at all points in time) allow you to perform the following actions:

- *Hide*—Hide the debugger's console window and variables pane to maximize the screen real estate offered to the text editor.
- *Pause*—Immediately pause the iPhone application and enter the debugger.
- *Continue*—Run the application until another breakpoint is hit.
- *Step Over*—Execute the next line of code and return to the debugger.
- *Step Into*—Execute the next line of code and return to the debugger. If the line calls any methods, step through their code as well.
- *Step Out*—Continue executing code until the current method returns.

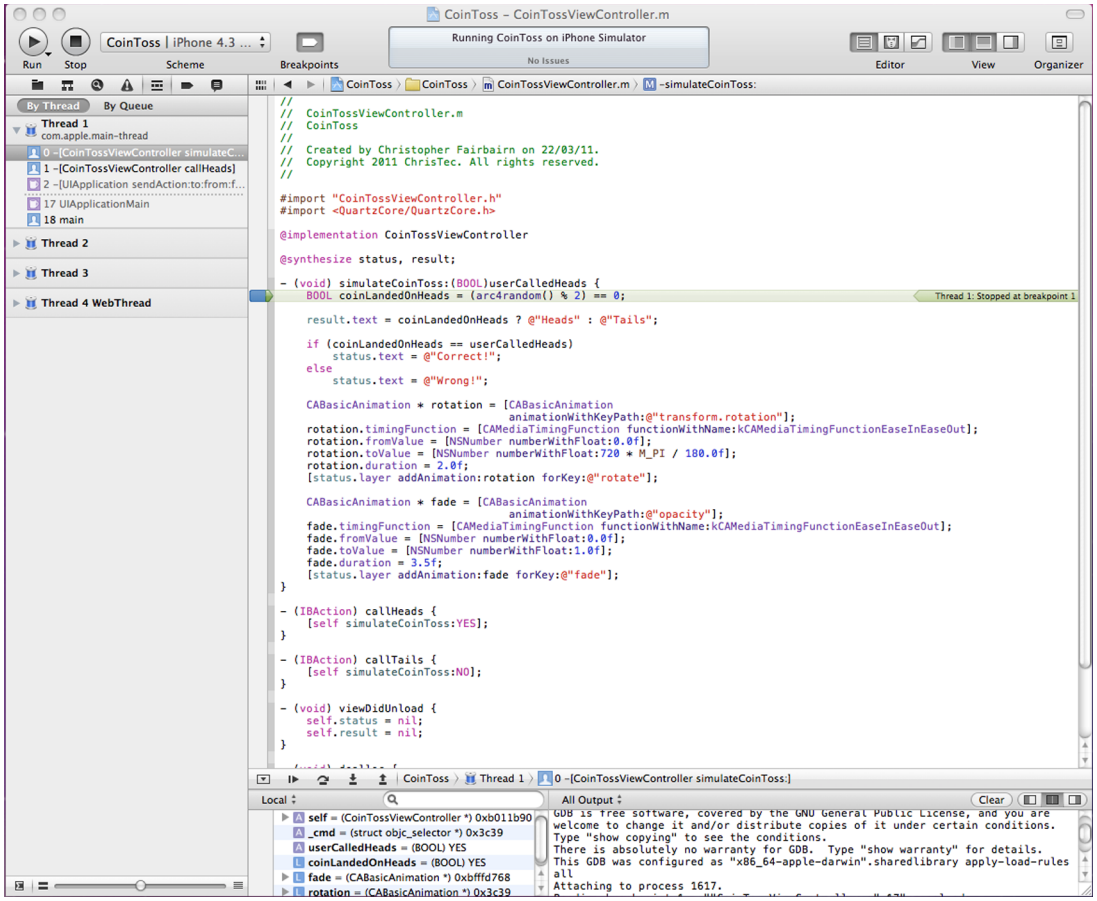

Figure 1.14 The Xcode debugger window after execution has reached a breakpoint

Your breakpoint caused the debugger to pause execution at the beginning of a simulated coin toss. If you view the variables pane or hover the mouse over the `userCalledHeads` argument, you can determine if the user has called heads (`YES`) or tails (`NO`).

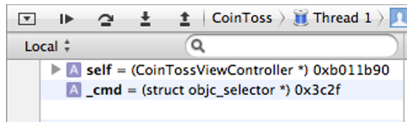

Figure 1.15 The toolbar options in Xcode for controlling the debugger

The first line of the `simulateCoinToss:` method simulates flipping a coin (by selecting a random number, 0 or 1). Currently, the debugger is sitting on this line (indicated by the red arrow in the margin), and the statements on this line haven't been executed.

To request that the debugger execute a single line of source code and then return to the debugger, you can click the Step Over button to "step over" the next line of source code. This causes the coin toss to be simulated, and the red arrow should jump down to the next line that contains source code. At this stage, you can determine the

result of the coin toss by hovering the mouse over the `coinLandedOnHeads` variable name; once again, `YES` means heads and `NO` means tails.

Using the step-over feature a couple more times, you can step over the two `if` statements, which update the result and status `UILabels` in the user interface. Unlike what you may expect, however, if you check the iPhone device at this stage, the labels won't have updated! This is because of how the internals of Cocoa Touch operate: the screen will only update once you release the debugger and allow this method to return to the operating system.

To allow the iPhone to update the user interface and to see the fancy animations that herald in a new coin toss result, you can click Continue (or press Cmd-Option-P) to allow the application to continue execution until it hits another breakpoint or you explicitly pause it again. Taking a look at the iPhone, you should see that the results of the coin toss are finally displayed onscreen.

1.7 Summary

Congratulations, you've developed your first iPhone application! Show your friends and family. It may not be the next iTunes App Store blockbuster release, but while putting together this application, you've mastered many of the important features of the Xcode IDE, so you're well on your way to achieving success.

Although Objective-C is a powerful language with many capabilities, you'll find using visual tools such as Xcode can lead to a productivity boost, especially during initial prototyping of your application. The decoupling of application logic from how it's presented to the user is a powerful mechanism that shouldn't be underestimated. It's doubtful the first user interface you design for your application will be perfect, and being able to alter it without having to modify a single line of code is a powerful advantage.

By the same token, you were able to rely on the Cocoa Touch framework to handle the minutiae of how to implement many of the features of your game. For example, the animations were implemented in a fairly declarative manner: you specified starting and stopping points for the rotations and fading operations and left the Quartz Core framework to worry about the specifics of redrawing the screen, transitioning the animation, and speeding up or slowing down as the effect completed.

As you'll continue to see, there's great power in the Cocoa Touch frameworks. If you find yourself writing a vast amount of code for a particular feature, chances are you aren't taking maximum advantage of what Cocoa has to offer.

In chapter 2, we dive into data types, variables, and constants and are introduced to the Rental Manager application that you'll build throughout this book.

iPHONE/iPAD DEVELOPMENT

Objective-C Fundamentals

Fairbairn • Fahrenkrug • Ruffenach

Objective-C Fundamentals guides you gradually from your first line of Objective-C code through the process of building native apps for the iPhone. Starting with chapter one, you'll dive into iPhone development by building a simple game that you can run immediately. You'll use tools like Xcode 4 and the debugger that will help you become a more efficient programmer. By working through numerous easy-to-follow examples, you'll learn practical techniques and patterns you can use to create solid and stable apps. And you'll find out how to avoid the most common pitfalls.

What's Inside

- Objective-C from the ground up
- Developing with Xcode 4
- Examples work unmodified on iPhone

No iOS or mobile experience is required to benefit from this book but familiarity with programming in general is helpful.

Christopher Fairbairn, Johannes Fahrenkrug, and **Collin Ruffenach** are professional mobile app developers, each with over a decade of experience using different systems including iOS, Palm, Windows Mobile, and Java.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/Objective-CFundamentals

"A handy and complete reference."
—Glenn Stokol
Oracle Corporation.

•

"The essential iOS programming how-to guide."
—Dave Bales, Whitescape

•

"A tour-de-force of Objective-C...I want to grok this stuff!"
—Dave Mateer, Mateer IT

•

"A superb introduction to essential iPhone application development tools."
—Carl Douglas, NZX

•

"Become a hot commodity on the market... with this book."
—Ted Neward, Principal,
Neward & Associates

ISBN 13: 978-1-935182-53-5
ISBN 10: 1-935182-53-6
54499
9 781935 182535

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]