1

Getting Started

This chapter serves as a quick introduction to the tools bundled with the SDK. It also shows you basic development steps that include coding, UI design, and debugging. You do not have to understand everything in this chapter as we will go over these concepts throughout the book. What you need to get from this chapter is a feeling of iPhone development using XCode.

We start with some basics of the XCode IDE in Section 1.1. Next, Section 1.2 talks about the UI design tool Interface Builder. After that, we show you how to use the built-in debugger in XCode in Section 1.3. Next, Section 1.4 shows you different sources of information for obtaining additional help. Finally, we summarize the chapter in Section 1.5.

1.1 SDK and IDE Basics

In this section, we walk you through the process of creating your first iPhone application. But first, you need to obtain the iPhone SDK and install it on your Mac.

1.1.1 Obtaining and installing the SDK

Obtaining and installing the iPhone SDK is easy; just follow these steps:

1. Get your iPhone developer Apple ID and password from:
   `http://developer.apple.com/iphone/`
2. Download the latest iPhone SDK for iPhone OS from the site mentioned above.
3. Install the iPhone SDK on your Intel-based Mac.

Now, you're ready to create your first project – read on!

1.1.2 Creating a project

Locate XCode and launch it. You can use `Spotlight` to find it or you can navigate to `/Developer/Applications/XCode`. XCode is the central application for writing, designing, debugging, and deploying your iPhone applications. You will use it a lot, so go ahead and add it to the `Dock`.

From XCode, select `File->New Project`. You should see a window, similar to the one shown in Figure 1.1, asking you for the type of project you want to create. Choose the default and create a window-based application. This is the most generic type of iPhone project and the one that can be customized for different needs.

Figure 1.1 Choosing window-based application in the project creation process.

Click on `Choose...` and enter the name of your project (here, we're using `My Project`) and hit `Save`. A new directory is created with the name you entered, and several files are generated for you. You should now see the newly created iPhone project as in Figure 1.2.

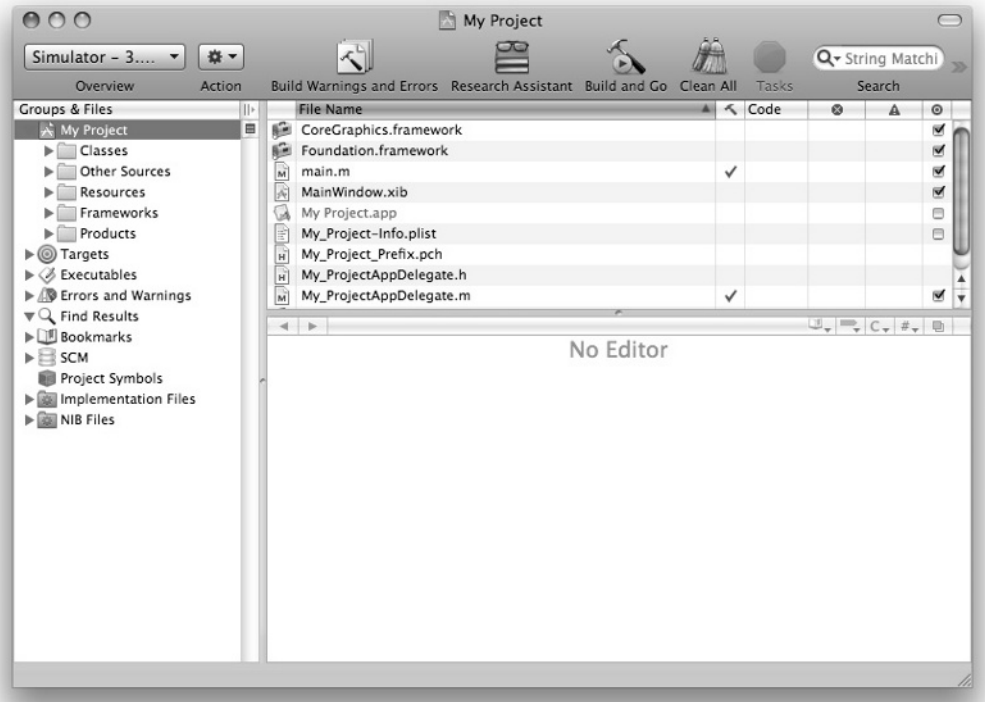

Figure 1.2 A newly created iPhone project in XCode.

1.1.3 Familiarizing yourself with the IDE

As you can see from Figure 1.2, the main window is divided into several areas. On the top, you will find the Toolbar (Figure 1.3). The Toolbar provides quick access to common tasks. It is fully configurable; you can add and remove tasks as you want. To customize the Toolbar, Control-click it and choose `Customize Toolbar...`. There, you can drag your favorite task on the Toolbar. Hit `Done` when you're finished. To remove an item, Control-click on it and choose `Remove Item`.

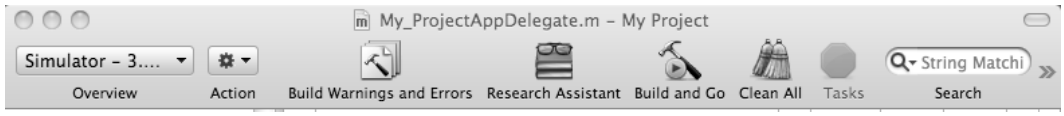

Figure 1.3 The XCode Toolbar.

On the left-hand side, you'll see the `Groups & Files` list (Figure 1.4).

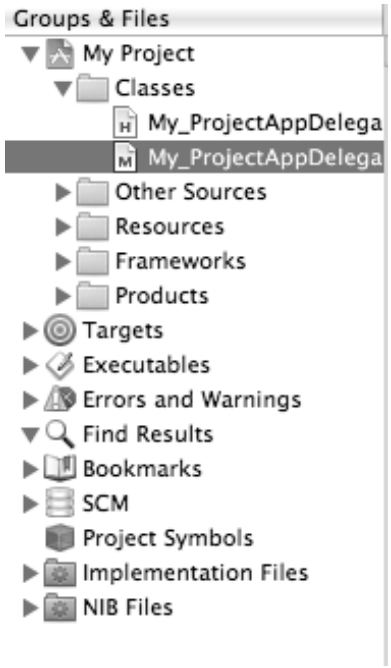

Figure 1.4 The `Groups & Files` list in XCode.

This list is used to organize the source code, frameworks, libraries, executables, and other types of files in your project.

The list shows several files and groups. Groups can contain other groups and files. You can delete a group as well as create a new one. The group indicated by the blue icon whose name is the same as the name you've chosen as the project name is a *static group*. Underneath it, you see all your headers, implementations, resources (images, audio files, etc.), and other related files. The folder-like yellow groups act conceptually as containers. You can have containers inside other containers and all files inside these containers live in the same directory on the disk. The hierarchy only helps you organize things. You have full freedom to organize your project's layout as you like. The compiler will pick up the resources, headers, and implementation files when it builds your application.

The other kind of groups that are listed below the project group are called *smart groups*. There are two types of smart groups: 1) built-in smart groups, and 2) custom smart groups. The content of the built-in smart groups cannot be customized. Examples of these groups include executables, bookmarks, errors/warnings, and targets. Customized smart groups are shown in purple, and two predefined groups are created for you when you create a new project.

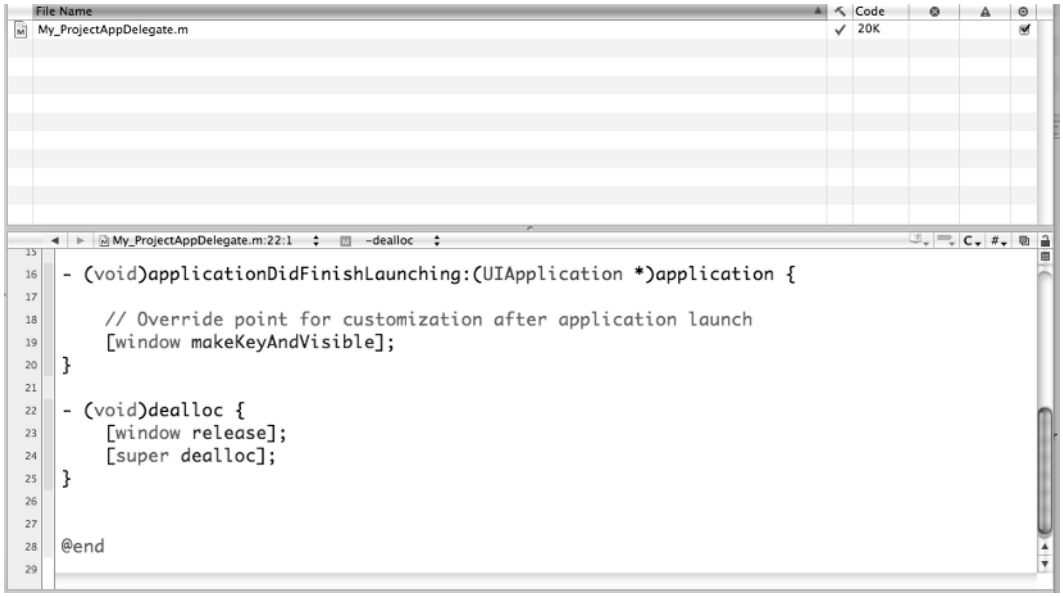

Figure 1.5 The Details view with the text editor view.

Figure 1.5 shows the `Details` view and the text editor beneath it.

Selecting an item in the `Groups & Files` list will result in its details being shown in the `Details` view. You can go to a full-editor window using `Command-shift-E`.

1.1.4 Looking closely at the generated code

Expand the `Classes` and `Other Sources` groups. You will notice several files that live underneath these two groups. Click on the `main.m` file and expand to a full-editor view.

The `main.m` file looks very similar to a C file with a `main()` function. As we will see later in this book, all that `main()` does is prepare for memory management and launch the application.

Click on the `My_ProjectAppDelegate.h` file under the `Classes` group. You will notice that the editor changes its content. This file contains the declaration of the application delegate class. Every application that runs on the iPhone OS has a delegate object that handles critical phases of its lifecycle.

Click on `My_ProjectAppDelegate.m`. This file with the `.m` extension is the counterpart of the previous `.h` file. In it, you see the actual implementation of the application delegate class. Two methods of this class are already implemented for you. The `applicationDidFinishLaunching:` method is one of those methods that handles a particular phase of the application lifecycle. The other

method, `dealloc`, is a method where memory used by this object is released. In iPhone OS, you manage the allocation and freeing of memory as there is no garbage collection. Memory management is crucial in iPhone development, and mastering it is very important. The first chapters are dedicated to teaching you exactly that – and much more.

The generated files and resources are adequate for starting the application. To launch the application, click on `Build and Go` in the Toolbar or press the `Command-Enter` key combination. You'll notice that the application starts in the Simulator and it only shows a white screen with the status bar on top. Not very useful, but it works!

1.2 Creating Interfaces

To be useful, an iPhone application needs to utilize the amazing set of UI elements available from the SDK. Our generated iPhone application contains a single UI element: a window.

All iPhone apps have windows (usually one.) A window is a specialized view that is used to host other views. A view is a rectangle piece of real-estate on the 320×480 iPhone screen. You can draw in a view, animate a view by flipping it, and you can receive multi-touch events on it. In iPhone development, most of your work goes towards creating views, managing their content, and animating their appearance and disappearance.

Views are arranged into a hierarchy that takes the shape of a tree. A tree has a root element and zero or more child elements. In iPhone OS, the window is the root element and it contains several child views. These child views can in turn contain other child views and so on and so forth.

To generate views and manage their hierarchy, you can use both Interface Builder (IB) and Objective-C code. IB is an application that comes with the SDK that allows you to graphically build your view and save it to a file. This file is then loaded at run-time and the views stored within it come to life on the iPhone screen.

As we mentioned before, you can also use Objective-C code to build the views and manage their hierarchy. Using code is preferred over using IB for the following reasons. First, as beginner, you need to understand all aspects of the views and their hierarchy. Using a graphical tool, although it simplifies the process, does hide important aspects of the process. Second, in advanced projects, your views' layouts are not static and change depending on the data. Only code will allow you to manage this situation. Finally, IB does not support every UI element all the time. Therefore, you will sometimes need to go in there and generate the views yourself.

The following section teaches you how to use IB. However, for the most part in this book, Objective-C code is used to illustrate the UI concepts. For extensive coverage of Interface Builder, please see Appendix F.

1.2.1 Interface Builder

The project has a basic window resource file. This file can be found under the `Resources` group. Expand the `Resources` group and locate the file `MainWindow.xib`. This file contains the main window of the application. This file is an `.xib` file that stores the serialized objects in the interface. When the project is built, this file is converted to the more optimized format `.nib` and loaded into memory when one or more of the UI components stored in it are requested.

Double-click on the `MainWindow.xib` file to launch IB. IB starts by opening four windows. The first window shows the main window stored in the file. The second window shows the document window listing the different objects stored in the file. The third window is the Library window containing all the UI objects that you can add to the file. The fourth and final window is the Inspector window with its four panes.

The Inspector window shows the attributes of the currently selected object. If you click on an object, the Inspector windows shows you its attributes distributed among four different panes. Each pane has several sections. You can change these attributes (such as color, position, and connections) and the changes will propagate to your project's user interface.

The main window of the application is white; let's change it to yellow. Click on the window object in the document window. In the Inspector window, make sure that the left-most pane is selected. In the View section of this pane, change the background color to yellow as shown in Figure 1.6.

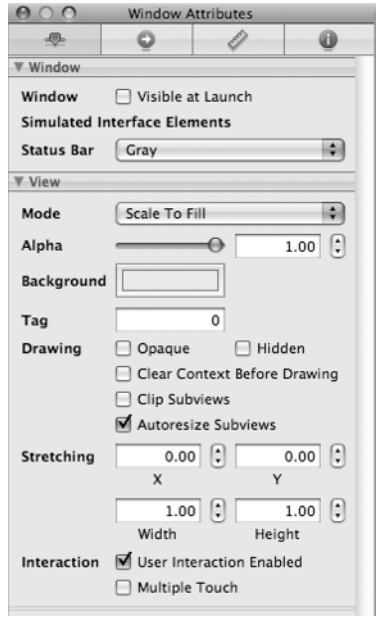

Figure 1.6 The attributes pane in the Inspector window of Interface Builder.

Go to XCode and run the application. Notice how the main window of the application has changed to yellow. It is important to keep the project open in XCode while working with IB. XCode and IB communicate well when both applications are open.

To build a user interface, you start with a view and add to it subviews of different types. You are encouraged to store separate views in separate `.xib` files. This is important as referencing one object in a file will result in loading all objects to main memory. Let's go ahead and add a label view to our window. This label will hold the static text "Hello iPhone."

A label is one of the many UI components available for you. These components are listed under several groups in the `Library`. Locate the `Library` window and click on `Inputs & Values` as shown in Figure 1.7.

Figure 1.7 The Library window of Interface Builder.

Click on the `Label` item and drag it onto the middle of the window. Expand the dimensions of the label as shown in Figure1.8.

When the label is selected, the Inspector window changes to reflect the attributes of the label. Figure 1.9 shows a portion of the attributes of a label in the Inspector window. You can change these attributes and observe the effect they have on the object instantaneously.

The label's text is left justified; let's make it center. In the `Layout` item of the attributes, click on the icon indicating center. Notice how the label text becomes centered. The text of the label can be changed in the Text item. Change `Label` to `Hello iPhone`. Go to XCode and hit `Build and Go`. You will notice the window showing `Hello iPhone` in the middle.

Figure 1.8 Adding a label view to a window in IB.

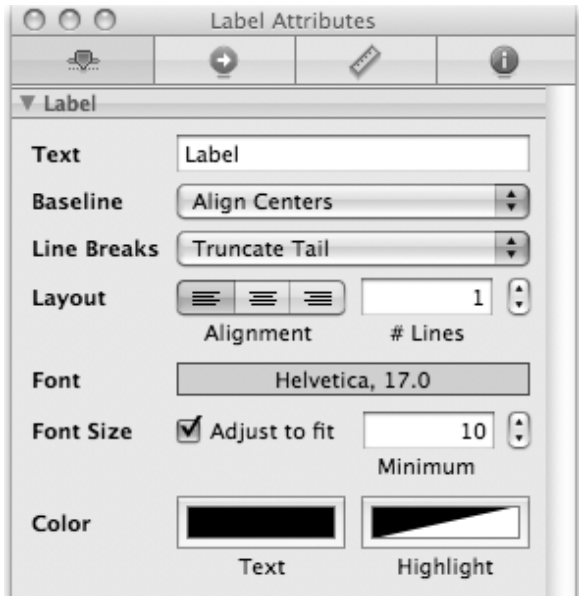

Figure 1.9 Attributes of a label in the Inspector window.

The text of the label is small, so let's make it bigger. Click on the `Text` item and choose a text size of 48 points. Go to XCode and hit `Build and Go`. Figure 1.10 shows a screenshot of the completed `Hello iPhone` application.

Figure 1.10 A screenshot of the completed `Hello iPhone` application.

Congratulations on your first successful iPhone application!

You deliver the product to the client and he is happy. However, he wants the application to have more interaction with the user. He asks you to revise the application by adding a button that the user can tap on to change the text displayed in the label.

Open the `MainWindow.xib` document if it is not already open. Locate the `Round Rect Button` item under `Items & Values` in the `Library` window. Drag and drop it under the label in the main window. Change the button's title by entering "Change" in the `Title` field found in the fourth section of the attributes window. The main window should look like the one shown in Figure 1.11.

Now that we have a button, we want to have a method (a function) in our code to get executed when the user touches the button. We can achieve that by adding a connection between the button's touch event and our method.

Figure 1.11 The main window after adding a new button.

Click on the button so that it becomes selected. Click on the second pane in the Inspector window. This pane shows the connections between an object and our code. The pane should look like the one in Figure 1.12.

Figure 1.12 The connections pane of our new button.

Now, we want to add a connection between the Touch Down event and a method we call buttonTapped. Let's first add this method in My_ProjectAppDelegate class.

In the My_ProjectAppDelegate.h file, add the following before **@end**.

```
-(IBAction)buttonTapped;
```

In the My_ProjectAppDelegate.m file, add the buttonTapped method body. The My_ProjectAppDelegate.m file should look something like the one in Listing 1.1.

Listing 1.1 The application delegate class after adding a new method.

```
#import "My_ProjectAppDelegate.h"
@implementation My_ProjectAppDelegate
@synthesize window;
- (void)applicationDidFinishLaunching:(UIApplication *)application {

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

-(IBAction)buttonTapped{
  UILabel *label = (UILabel*)[window viewWithTag:55];
  if([label.text isEqualToString:@"Hello iPhone"])
    label.text = @"Hello World";
  else
    label.text = @"Hello iPhone";
}

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

The buttonTapped method simply obtains a reference to the label and changes its text to either "Hello World" or "Hello iPhone". You don't need to understand this code at this stage. All you need to understand is that the label on the screen is encapsulated by the UILabel class and it's tagged with the number 55.

Now, let's switch to IB and add a tag to the label so that it can be retrieved from the code. Click on the label and in the Inspector window, choose the first pane. In the second section, enter 55 for the Tag field (fourth item.)

We still need to perform one last step. We need to connect the touch event with the method we just created. Click on the button and choose the connections pane (second pane). Control-click or right-click on the circle on the right-hand side of Touch Down event and drag it on top of the My_ProjectAppDelegate object in the Document window and let go as shown in Figure 1.13.

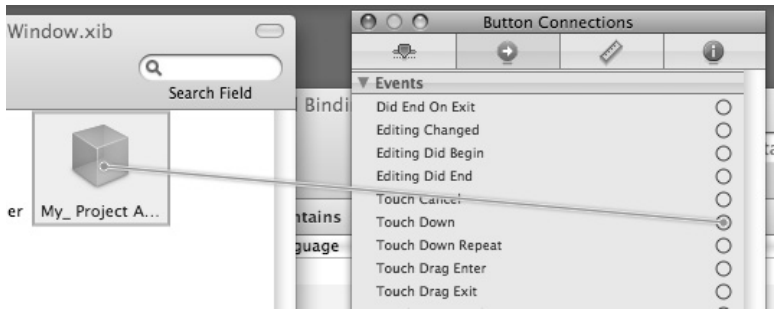

Figure 1.13 Making a connection between an event and a method in another object.

When you release the mouse, IB shows you potential methods (actions) that you can connect this event to. Right now we only have one action and that action is `buttonTapped`. Select that action and you'll notice that a connection has been made as shown in Figure 1.14.

Figure 1.14 A connection between a touch event and an action.

Now, switch to XCode and hit `Build and Go`. You'll notice that tapping on the button changes the text value of the label.

1.3 Using the Debugger

During the development of your applications, often things go wrong and the feature that you've just added is not functioning properly. At these moments, the built-in debugger becomes invaluable.

Let's introduce a bug into our code. Go to `My_ProjectAppDelegate.m` file and change the tag's value used to obtain the label from 55 to 54, then `Build and Go`. Now, tapping the button has no effect on the label's text.

First, you want to make sure that the `buttonTapped` method gets called. In XCode, click in the left margin of the first line in the `buttonTapped` method as shown in Figure 1.15. After you click there, a breakpoint (shown in blue) is added.

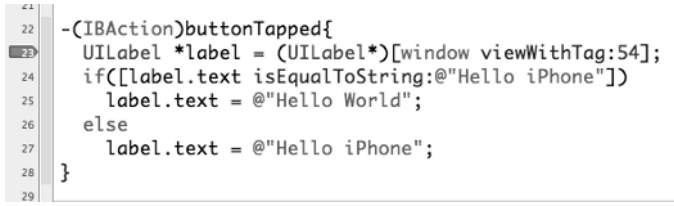

Figure 1.15 Adding a breakpoint in the `buttonTapped` method.

Click `Build and Go` to debug the application. When the application launches, tap on the button. You'll notice that the execution hits the breakpoint as shown in Figure 1.16. At least we know that we made our connection correctly.

Figure 1.16 Hitting a breakpoint in the `buttonTapped` method.

Let's step over the statement that obtains the label from the window. Click on the `Step Over` button located beneath the Toolbar as shown in Figure 1.17.

After stepping over the statement, we need to inspect the value obtained. Hover the mouse over `label` in the statement just executed as shown in Figure 1.18. A tip appears showing its value. Notice that the value is $0x0$. In Objective-C, this value is called `nil` and means that no object is stored in this variable. After inspecting the tag value and going back-and-forth between XCode and `IB`, we find the problem, fix it, remove the breakpoint by clicking on it to turn it off, and hit `Build and Go`.

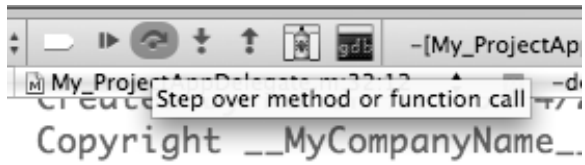

Figure 1.17 Step over a function or a method call button.

Figure 1.18 Inspecting the value of the label after obtaining it from the window.

1.4 Getting More Information

There are plenty of sources for information on the SDK. These sources include the following:

- **Developer Documentation**. The best locally stored source of information is the Developer Documentation. In XCode, select `Help->Documentation`. The documentation window appears as shown in Figure 1.19. You can search using the search box on the left-hand corner for any defined type in the SDK. The documentation is hyper-linked and you can go back-and-forth between different pieces of information. It's easy to use and it will become your friend.
- **Developer Documentation from within XCode**. If you're in XCode and you need more information about something, Option-double-click it and the Developer Documentation opens with more information.
- **Other help from within XCode**. If you're in XCode and you need to get the declaration and possible implementation of a given token (e.g., class, tag, variable, etc.), Command-double-click it. If there are multiple pieces of information, or disambiguation is needed, a list of items to choose from will be shown.
- **iPhone Dev Center**. The center is located at `http://developer.apple.com/iphone/`. The iPhone Dev Center has a large collection of technical resources and sample code to help you master the latest iPhone technologies.
- **Apple's Fora**. You can start with the site at `https://devforums.apple.com/`.
- **The Web**. There is plenty of information on the web. Just enter a relevant query and let Google do its magic!

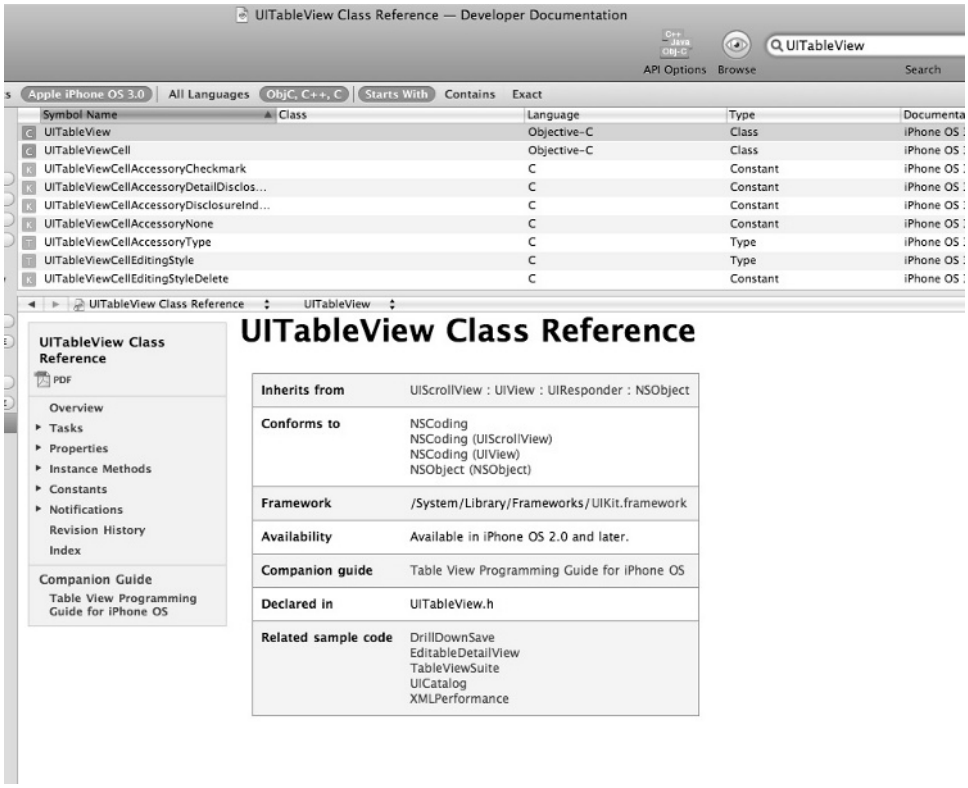

Figure 1.19 The Developer Documentation in XCode.

1.5 Summary

This chapter provided a gentle introduction to the world of iPhone development. We showed you in Section 1.1 how to download and install the iPhone SDK. After that, we iterated through the development of an iPhone application and showed you how to use Interface Builder to design user interfaces. Next, Section 1.3 discussed how to debug an iPhone application using the built-in visual debugger in XCode. You were also exposed to different sources for obtaining further help on the tools and the SDK in general in Section 1.4.

The rest of the book will detail all aspects of iPhone development. However, from now on, since we want to teach you everything you need, we will stop using Interface Builder and show you how to build your UI using code. This will help you gain a solid understanding of the process. You can, of course, mix and match with Interface Builder as you wish.

The next two chapters cover the Objective-C language and the coding environment that you will be working with: `Cocoa`. We hope you're as excited as we are!

Problems

(1) Check out the `UILabel.h` header file and read about the `UILabel` class in the documentation.

(2) What's an `IBOutlet` and `IBAction`? Use Command-double-click to see their definitions in the `UINibDeclarations.h` header file.

(3) Explore the XCode IDE by reading the `XCode Workspace Guide` under the `Help` menu of the XCode application.

(4) Explore Interface Builder by choosing `Interface Builder Help` under the `Help` menu of the Interface Builder application.