iPad Application Development for Dummies

by Tony Bove and Neal Goldstein
Published in 2011 by Wiley Publishing, Inc.

100% written by Tony Bove:

100% written by Neal Goldstein:

Sample Chapters Included in this PDF

The following are sample chapters 5 (SDK), 7 (how apps work), and 8 (how apps run). Please ignore the formatting conventions and comments used for Dummies books. The page numbers do not correspond to book page numbers.

Chapter 5

Getting to Know the SDK

In This Chapter

* Getting a handle on the Xcode project
* Compiling an iPad app
* Peeking inside the Simulator
* Checking out the Interface Builder
* Demystifying nib files

Arthur C. Clarke's Third Law is that any sufficiently advanced technology is indistinguishable from magic, and Steve Jobs echoed these words when he announced the iPad as "our most advanced technology in a magical and revolutionary device." To deploy this magic and practice the alchemy of application development, you need to learn how to use the development tools.

The collection of tools known as the iOS Software Development Kit (SDK) is the crucible for grinding out an iPad app. You pick a template for the type of app; stir in the content, behavior, and user interface; and cast your spells with magical code. The SDK builds your final product. Sounds easy, and to be truthful, it's *relatively* easy.

In this chapter, I introduce you to the SDK. It's going to be a low-key, get-acquainted kind of affair. You get into the real nuts-and-bolts stuff in Parts IV and V, when you actually develop the two sample applications.

Developing Using the SDK

The iOS Software Development Kit (SDK) gives you the opportunity to develop your apps without tying your brain up in knots. It includes Xcode, Apple's development environment that runs on the Mac OS X operating system. To develop an iPad app, you have to work within the context of an Xcode project. The SDK also includes Interface Builder, which launches from Xcode when you double-click a ~~.xib~~ file. You use it to quickly build your app's user interface. The idea here is to add your code incrementally @@md step by step @@md so that you can always step back and see how what you just did affects the Big Picture.

Starting an app from scratch

This chapter assumes that you're creating a new iPad app (in particular, the DeepThoughts sample app) from scratch, using the Xcode templates to get started @@md which is certainly the fastest way to get started. The Seven Development Steps to iPad App Heaven should look something like this:

Comp Services: Please do not bold the following numbered list at layout. Thanks.

1. Start with an Xcode template.
2. Design the user interface.
3. Write the code.
4. Build and run your app.
5. Test your app.
6. Measure and tune your app's performance.
7. Do it all again (or at least Steps 3@@nd6) until you're done.

<Remember>

If you have an idea for a new iPad app, the decision to start from scratch should be obvious. But if you've already developed an iPhone/iPod touch app, you have choices in how you use Xcode to develop your iPad app.

Starting from an existing iPhone app

Besides the fact that iPhone apps already run on the iPad in "compatibility mode" (in a black box in the center of the display, or scaled up to full screen), you can also *port* the iPhone app @@md modify its code just a bit @@md to use iPad device resources. Xcode makes the porting process easier by automating much of the setup process for your project. The most noticeable difference between the iPad and iPhone, besides the absence of telephony, is the size of views you create to present your user interface.

Xcode simplifies the process of updating your existing iPhone project to include the necessary files to support the iPad. Essentially, you would be using a single Xcode project to create two separate apps: one for the iPhone (and iPod touch) and one for the iPad. After selecting the target in the Targets section of the Groups & Files list of the Xcode Project window (which I show in the next section of this chapter), you can choose Project@@-->Upgrade Current Target for iPad and then choose to either upgrade your iPhone target to one *Universal* application that supports both iPhone and iPad or create two *device-specific* applications (one for the iPad and one for the iPhone/iPod touch). Here are the differences to help you make that decision:

* **A Universal application** is optimized for all device types. Although I don't cover creating a Universal application in this chapter, creating a Universal application allows you to sell one app that supports all device types. This choice makes the download experience simpler for users. (You can set one price, and users can use the same copy of the app on both their iPhone and iPad.)
* **Device-specific applications** are designed specifically for the device @@md iPhone (and iPod touch) or iPad. Although I don't cover this method in this chapter, it gives you the advantage of reusing code from your existing iPhone app while also taking less development and testing time than developing a Universal app.

You also have the choice of using *separate Xcode projects* to create separate apps for the iPad and iPhone. Essentially, this means starting from scratch. (See the later section "Starting an iPad app from scratch.") If you have to rewrite large portions of your code anyway, creating a separate Xcode project for the iPad is usually simpler. Creating a separate project gives you the freedom to tailor your code for the iPad without having to worry about whether that code runs on other devices. If your app's data objects are tightly integrated with the views that draw them, or if you just need the freedom to add more features to the iPad version, this is the way to go.

Whether you create device-specific application targets in one project or create separate projects, you still end up with two separate apps to manage. The only way to have only one app to manage for both iPhone and iPad is to create a Universal app.

In this chapter, you start at the very beginning, from scratch, with the very first step, which is Xcode. (Starting with Step 1? What a concept!) And the first step of the first step is to create your first project.

Creating Your Xcode Project

To develop an app, you work in what's called an *Xcode project.* So, time to fire one up. Here's how it's done:

1. **Launch Xcode.**

   After you've downloaded the SDK (painstakingly described in Chapter 4), it's a snap to launch Xcode. By default, it's downloaded to ~~/Developer/Applications~~, where you can track it down to launch it.

<Tip>

Here are a couple of hints to make Xcode handier and more efficient:

* Drag the icon for the Xcode application all the way down to the Finder's Dock so you can launch it from there. You'll be using it a lot, so it wouldn't hurt to be able to launch it from the Dock.
* When you first launch Xcode, you see the Welcome screen shown in Figure 5-1. (After using Xcode to create projects, your Welcome screen will list all of your most recent projects in the right column.) It's chock-full of links to the Apple Developer Connection and Xcode documentation. (If you don't want to be bothered with the Welcome screen in the future, deselect the Show This Window When Xcode Launches check box. You can also just click Cancel to close the Welcome screen.)

Figure 5-1: The Xcode Welcome screen.

2. **Choose Create a New Xcode Project from the Welcome screen (or choose File@@-->New Project) to create a new project.**

   You can also just press @@cmd+Shift+N.

   No matter what you do to start a new project, you're greeted by the New Project window, as shown in Figure 5-2.

The New Project window is where you get to choose the template you want for your new project. Note that the leftmost pane has two sections: one for iOS and the other for Mac OS X.

3. **In the upper-left corner of the New Project window, click Application under the iOS heading (if it isn't already selected).**

With Application selected, the main pane of the New Project window shows several choices. (See Figure 5-2.) Each of these choices is actually a template that, when chosen, generates some code to get you started.

4. **Select View-based Application from the template choices displayed.**

You'll use the View-based Application option to start the DeepThoughts app, the first sample app, which you develop in Part IV.

Note that when you select a template, a brief description of the template is displayed underneath the main pane. (Again, refer to Figure 5-2.) In fact, click some of the other template choices just to see how they're described as well. Just be sure to click the View-based Application template again when you're done snooping around so you can follow along with developing the DeepThoughts app.

5. **Select iPad from the Product pop-up menu, as shown in Figure 5-2, and then click Choose.**

You must choose iPad (*not* iPhone) from the Product pop-up menu to start a new iPad project from scratch @@md this choice puts the standard iPad resources into your project. After clicking Choose, the Save As dialog appears.

6. **Enter a name for your new project in the Save As field, choose a Save location (the Desktop or any folder works just fine) and then click Save.**

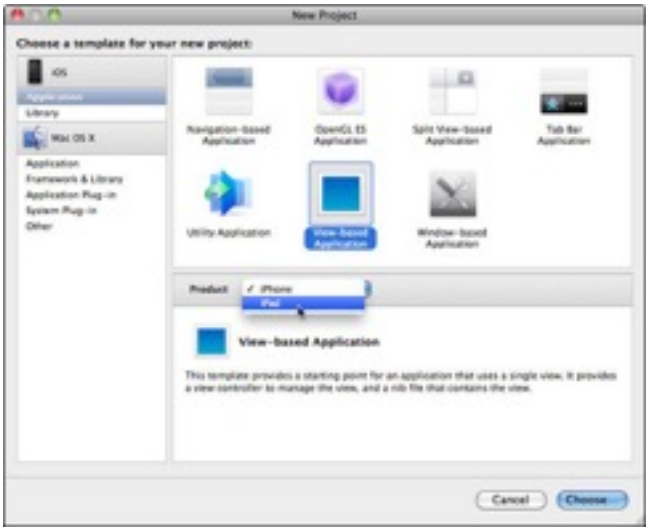

Figure 5-2: Select iPad in the Product pop-up of the New Project window.

I named the first sample app project DeepThoughts. (You should do the same if you're following along with developing DeepThoughts.)

After you click Save, Xcode creates the project and opens the Project window, which should look like what you see in Figure 5-3.

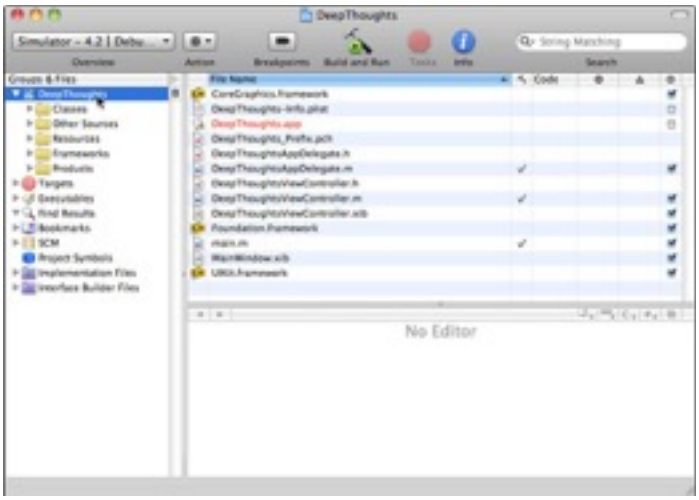

Figure 5-3: The DeepThoughts Project window.

Exploring Your Project

It turns out that you do most of your work on projects using a Project window. If you have a nice, large monitor, expand the Project window so you can see everything in it as big as life. This is, in effect, Command Central for developing your iPad app; it displays and organizes your source files and the other resources needed to build your app.

You have control over Command Central @@md you can organize your source files and resources as you see fit. The Groups & Files list on the left is an outline view of all of your project's files @@md source code, frameworks, and graphics, as well as some settings files. You can move files and folders around and add new folders.

<Tip>

You may notice that some of the items in the Groups & Files list are folders, whereas others are just icons. Most items have a little triangle (the disclosure triangle) next to them. Clicking the little triangle to the left of a folder/icon expands the folder/icon to show what's in it. Click the triangle again to hide what it contains.

To see more of the code that's already provided with the View-based Application template, select Classes in the Groups & Files list on the left side of the Project window, as shown in Figure 5-4. The first file should already be selected in the Detail view of the Project window: ~~DeepThoughtsAppDelegate.h~~. (Actually, you can select any file in the Detail view to see code.) The code appears in the Editor view.

Here's a summary of what you see in Figure 5-5:

* **The Groups & Files list:** As described earlier, the Groups & Files list provides an outline view of everything in your project. If you select an item in the Groups & Files list, the contents of the item are displayed in the topmost-pane to the right @@md otherwise known as the Detail view.
* **The Detail view:** Here you get detailed information about the item you selected in the Groups & Files list.
* **The Toolbar:** Here you can find quick access to the most common Xcode commands. You can customize the toolbar to your heart's content by right-clicking it and choosing Customize Toolbar from the contextual menu that appears. You can

also choose View@@-->Customize Toolbar. (Because you can customize the toolbar, it may differ somewhat from Figure 5-4.)

Figure 5-4: Code appears in the Editor view of the Project window.

* The Overview menu lets you specify the active SDK and active configuration, which I describe in “Building and Running Your Application” in this chapter.
* The Action menu lets you perform common operations on the currently selected item in the Project window. The actions change depending on what you’ve selected. (The same actions are available in the context-sensitive shortcut menu that appears when you Control-click a selected item.)
* Pressing the Build and Run button compiles, links, and launches your app in the Simulator.
* The Breakpoints button turns breakpoints on and off and toggles the Build and Run button to Build and Debug. (I explain breakpoints in Chapter 12.)
* The Tasks button allows you to stop the execution of the app that you’ve built.
* The Info button opens a window that displays information and settings for your project.
* The Search field lets you search the items currently displayed in the Detail view. I show you how to search for items in Chapter 10.
* The Show/Hide Toolbar button shows or hides the entire Toolbar.

* **The status bar:** Look here for messages about your project. (There are none yet in Figure 5-4; for a peek at a status message, see Figure 5-6.) For example, when

you're building your project, Xcode updates the status bar to show where you are in the process @@md and whether or not the process completed successfully.

* **The favorites bar:** The favorites bar appears under the Toolbar and works like other favorites bars @@md you can bookmark places in your project. This bar isn't displayed by default (nor is it shown in Figure 5-4); to put it onscreen, choose View@@-->Layout@@-->Show Favorites Bar from the main menu.

* **The Text Editor navigation bar:** As shown in Figure 5-5, this navigation bar contains a number of shortcuts (I explain more about them as you use them):

    * *Bookmarks menu:* You create a bookmark by choosing Edit@@-->Add to Bookmarks.

    * *Breakpoints menu:* Lists the breakpoints in the current file @@md I cover breakpoints in Chapter 12.

    * *Class Hierarchy menu:* The superclass of this class, the superclass of that superclass (if any), and so on. In Objective-C, you can base a new class definition on a class already defined, so that the new class inherits the methods of the base class it is based on. The base class is called a superclass; the new class is its subclass, and the hierarchy defines the relationship between a superclass, its subclass, and subclasses of the subclass (and so on). For a background in Objective-C, see Neal's *Objective-C For Dummies*.

    * *Included Files menu:* Lists both the files included by the current file, as well as the files that include the current file.

    * *Counterpart button:* Due to the natural split in the definition of an Objective-C class into interface and implementation, a class's code is often split into two files. The Counterpart button allows you to switch between the header (or interface) file, such as ~~DeepThoughtsAppDelegate.h~~, and the implementation file, such as ~~DeepThoughtsAppDelegate.m~~. The header files define the class's interface by specifying the class declaration (and what it inherits from); instance variables (a variable defined in a class @@md at runtime all objects have their own copy); and methods. The implementation file, on the other hand, contains the code for each method.

    * *Lock button:* Indicates whether the selected file is unlocked for editing or locked (preventing changes). If it's locked, you can click the button to unlock the file (if you have permission).

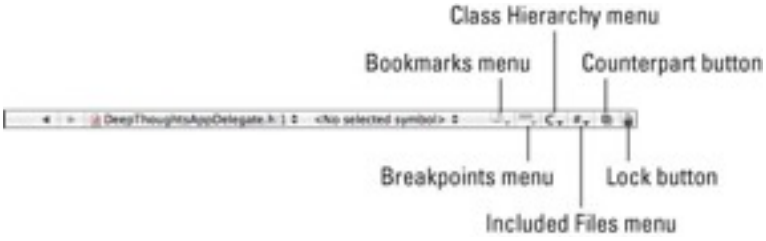

Figure 5-5: The Text Editor navigation bar.

* **The Editor view:** Displays a file you've selected, in either the Groups & Files list or Detail view. You can also edit your files here @@md after all, that's what you'd expect from the Editor view @@md although some folks prefer to double-click a file in Groups & Files list or Detail view to open the file in a separate window.

  To see how the Editor view works, refer to Figure 5-4, where I've clicked the Classes folder in the Groups & Files list, and the ~~DeepThoughtsAppDelegate.h~~ class in the Detail view. You can see the code for the class in the Editor view.

  Right under the Lock button (refer to Figure 5-4) is a tiny window shade icon that lets you split the Editor view. Click it to look at the interface and implementation

files at the same time, or even the code for two different methods in the same or different classes.

<Tip>

If you have any questions about what something does, just position the mouse pointer above the icon, and a tooltip explains it.

The first item in the Groups & Files list @@md selected and thus highly visible back in Figure 5-3 @@md is labeled DeepThoughts. This is the container that contains *all* the source elements for the project, including source code, resource files, graphics, and a number of other pieces that will remain unmentioned for now (but I get into those in due course). You can see that this project container has five distinct groups @@md Classes, Other Sources, Resources, Frameworks, and Products. Here's what gets tossed into each group:

* **Classes** is the group in which Xcode places all the template code for DeepThoughts, and you should also place new classes you create in the Classes group, although you aren't obliged to. The Classes group has four distinct source-code files (which you can see in the Detail view in Figure 5-4):
    * ~~DeepThoughtsAppDelegate.h~~
    * ~~DeepThoughtsAppDelegate.m~~
    * ~~DeepThoughtsViewController.h~~
    * ~~DeepThoughtsViewController.m~~
* **Other Sources** is the group in which you typically would find the frameworks you're using @@md stuff like ~~DeepThoughts_Prefix.pch~~ as well as ~~main.m~~, your application's main function, both of which are described in Chapter 8.
* The **Resources** group contains, well, resources specifically for the target (in this case, an iPad), such as ~~.xib~~ files (which you find out about in "Using Interface Builder" in this chapter), property lists (which you will encounter in Chapter 16), images, and other media files, and even some data files.

  Whenever you choose the View-based Application template (refer to Figure 5-2) and name it DeepThoughts, Xcode creates the following files for you:
    * ~~DeepThoughts-Info.plist~~
    * ~~DeepThoughtsViewController.xib~~
    * ~~MainWindow.xib~~

  I explain ~~.xib~~ files in excruciating detail in this chapter, and you get to play with them in Chapter 9 and the rest of Part IV. Soon you'll love ~~.xib~~ files as much as I do.
* **Frameworks** are code libraries that act a lot like prefab building blocks for your code edifice. (I talk a lot about frameworks in Chapter 7.) By choosing the View-based Application template, you let Xcode know that it should add the ~~UIKit framework~~, ~~Foundation.framework~~, and ~~CoreGraphics.framework~~ to your project, because it expects that you'll need them in an app based on the View-based Application template.

<Tip>

The DeepThoughts app is limited to just these three frameworks, but I show you how to add another framework to the iPadTravel411 sample app in Chapter 14.

* The **Products** group is a bit different from the previous three items in this list: It's not a source for your app, but rather *the compiled app itself.* In it, you find

~~DeepThoughts.app~~. At the moment, this file is listed in red because the file can't be found (which makes sense because you haven't built the app yet).

<Remember>

A file's name appearing in red lets you know that Xcode can't find the underlying physical file.

<TechnicalStuff>

If you happen to open the ~~DeepThoughts~~ folder on your Mac, you won't see the "folders" that appear in the Xcode window. That's because those folders are simply groupings that help organize and find what you're looking for; this list of files can grow to be pretty large, even in a moderate-size project.

When you have a lot of files, you'll have better luck finding things if you create subgroups within the Classes, and/or Resources groups, or even whole new groups. You create subgroups (or even new groups) in the Groups & Files list by choosing New Project@@-->New Group. You then can select a file and drag it to a new group or subgroup.

Building and Running Your Application

It's really a blast to see what you get when you build and run a project that you created @@md even if all you did was choose a template from the Project window. Building and running a project is relatively simple:

LAYOUT: Please note the "|" character in the following paragraphs.

1. **If it isn't already chosen, choose Simulator - 4.2 | Debug from the Overview drop-down menu in the top-left corner of the Project window to set the active SDK and Active Build Configuration.**

   This combination (Simulator - 4.2 | Debug) may be chosen already, as you can see back in Figure 5-4. Here's what that means:

   * When you download an SDK, you may actually download *multiple* SDKs @@md a Simulator SDK and a device SDK for each of the current iOS releases.
   * The one to use for iPad development is the *iPad Simulator 4.2* SDK for iOS 4.2. Later, you can switch to the actual device SDK and download your app to a real-world iPad, as described in Chapter 6. But before you do that, there's just one catch. . . .

<Remember>

   * You have to be in the iOS Developer Program to run your app on a device, even on your very own iPad. Go to Chapter 4 and enroll in the program if you haven't done so already.

   A *build configuration* tells Xcode the purpose of the built product. You can choose between Debug, which has features to help with debugging (there's a no-brainer for you); and Release, which results in smaller and faster binaries. You use Debug most of the time as you develop an app, and I use Debug for most of this book @@md so go with Debug for now. (Choose Simulator - 4.2 | Debug from the Overview drop-down menu.)

2. **Choose Build@@-->Build and Run from the main menu to build and run the application.**

You can also press @@cmd+Return or click the Build and Run button in the Project Window toolbar. The status bar in the Project window tells you all about build progress, build errors such as compiler errors, or warnings @@md and (oh, yeah) whether the build was successful. Figure 5-6 shows that this was a successful build @@md you can tell by the ~~Succeeded~~ message in the bottom-right corner of the window.

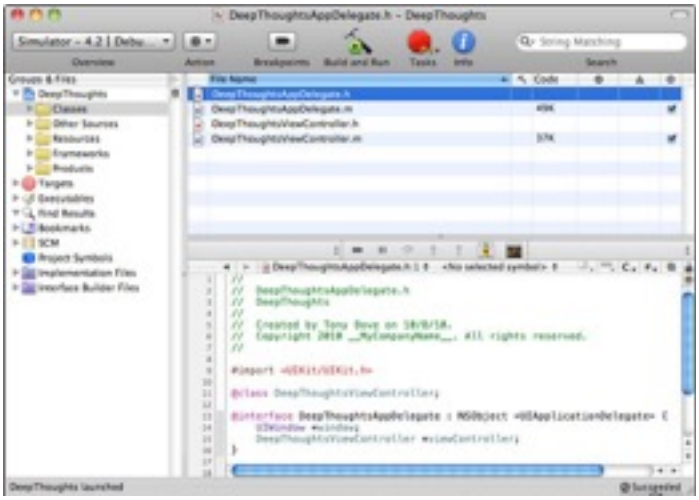

Figure 5-6: A successful build.

You can also display the Build Results window by clicking the ~~Succeeded~~ message in the Status bar (You find out more about debugging and the Build Results window in Chapter 12.)

After it's launched in the Simulator, your first app looks a lot like what you see in Figure 5-7. You should see the gray status bar and a white window, and the simulated Home button on the bottom to quit your app, but that's it. You can also choose actions in the Hardware menu (shown in Figure 5-7), which I explain next.

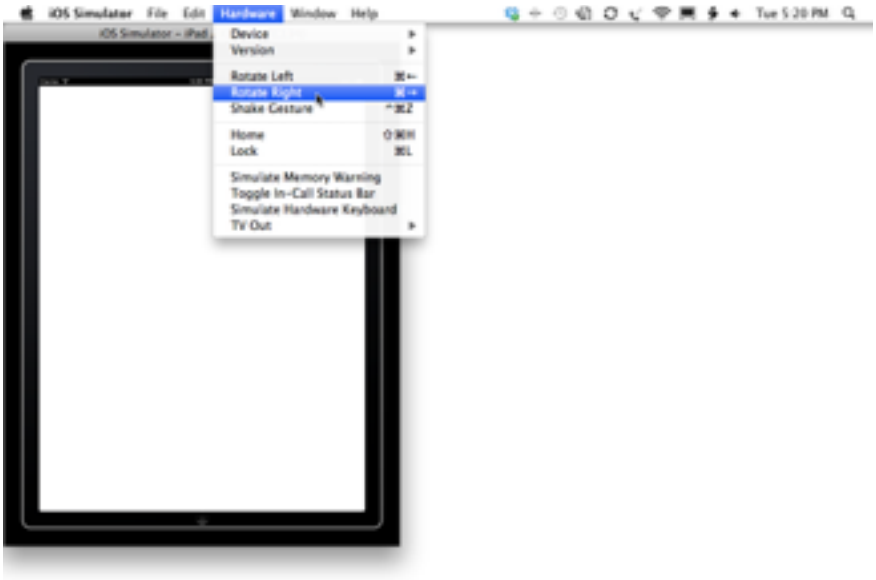

Figure 5-7: The DeepThoughts app in the Simulator.

The Simulator

When you run your app, Xcode installs it on the iOS Simulator (or a real iPad if you specified the device as the active SDK, as shown in Chapter 6) and launches it.

Using the Hardware menu and your keyboard and mouse, the Simulator mimics most of what a user can do on a real iPad, albeit with some limitations that I point out shortly.

Hardware interaction

You use the Simulator's Hardware menu (refer to Figure 5-7) when you want the Simulator to simulate the following:

* **Rotate left:** Choosing Hardware@@-->Rotate Left rotates the Simulator to the left. If the Simulator is in portrait view, it changes to landscape view; if the Simulator is already in landscape view, it changes to portrait view.
* **Rotate right:** Choosing Hardware@@-->Rotate Right rotates the Simulator to the right, with the same effect as choosing Hardware@@-->Rotate Left.
* **Use a shake gesture:** Choosing Hardware@@-->Shake Gesture simulates shaking the iPad.
* **Go to the Home screen:** Choosing Hardware@@-->Home does the expected @@md you go to the Home screen.
* **Lock the Simulator (device):** Choosing Hardware@@-->Lock locks the simulator.
* **Send the running app low-memory warnings:** Choosing Hardware@@-->Simulate Memory Warning fakes out your app by sending it a (fake) low-memory warning. I don't cover this in this book, but it's a great feature for seeing how your app may function out there in the real world.

* **Toggle the status bar between its Normal state and its In Call state:** Choose Hardware@@-->Toggle In-Call Status Bar to check out how your app functions when the device is not answering a call (Normal state) and when it supposedly *is* answering a call (In Call state) @@md these choices apply only to the iPhone as of this writing.
* **Simulate the hardware keyboard:** Choose Hardware@@-->Simulate Hardware Keyboard to check out how your app functions when the iPad is connected to the optional physical keyboard dock.
* **TV Out:** To bring up another window that acts like an external display attached to the device, choose Hardware@@-->TV Out, and then choose 640x480, 1024x768, or 1280x720 for the window's display resolution. Choose Hardware@@-->TV Out@@-->Disabled to close the external display window.

Gestures

On the real device, a gesture is something you do with your fingers to make something happen in the device, like a tap, or a drag, and so on. Table 5-1 shows you how to simulate gestures using your mouse and keyboard.

LAYOUT: Please format the numbers in the num lists within the second column of the table so that the text aligns, and the number outdents, like we do for regular Num Lists. Thanks.

Table 5-1 Gestures in the Simulator

~~*Gesture*~~	~~*iPad Action*~~
Tap	Click the mouse.
Touch and hold	Hold down the mouse button.
Double tap	Double-click the mouse.
Two-finger tap	1. Move the mouse pointer over the place where you want to start. 2. Hold down the Option key, which makes two circles appear that stand in for your fingers. 3. Press the mouse button.
Swipe	1. Click where you want to start and hold the mouse button down. 2. Move the mouse slowly in the direction of the swipe and then release the mouse button.
Flick	1. Click where you want to start and hold the mouse button down. 2. Move the mouse quickly in the direction of the flick and then release the mouse button.
Drag	1. Click where you want to start and hold the mouse button down. 2. Move the mouse slowly in the drag direction.
Pinch	1. Move the mouse pointer over the place where you want to start. 2. Hold down the Option key, which makes two circles appear that stand in for your fingers. 3. Hold down the mouse button and move the circles in (to pinch) or out (to un-pinch).

Uninstalling apps and resetting your device

You uninstall applications on the Simulator the same way you'd do it on the iPad, except you use your mouse instead of your finger.

1. **On the Home screen, place the pointer over the icon of the app you want to uninstall and hold down the mouse button until all the app icons start to wiggle.**
2. **Click the app icon's Close button @@md the little *x* that appears in the upper-left corner of the icon @@md to make the app disappear.**

3. **Click the Home button @@md the one with a little square in it, centered below the screen @@md to stop the other app icon's wiggling and finish the uninstallation.**

You can also move an app's icon around by clicking and dragging with the mouse.

You can remove an application from the background the same way you'd do it on the iPad, except you use your mouse instead of your finger.

1. **Double-click the Home button to display the applications running in background.**
2. **Place the pointer over the icon of the application you want to remove and hold down the mouse button until the icon starts to wiggle.**
3. **Click the icon's Remove button @@md the red circle with the @@ms that appears in the upper-left corner of the application's icon.**
4. **Click the Home button to stop the icon's wiggling and then once again to return to the Home screen.**

To reset the Simulator to the original factory settings @@md which also removes all the apps you've installed @@md choose iOS Simulator@@-->Reset Content and Settings.

Limitations

Keep in mind that running apps in the Simulator isn't the same thing as running them in the iPad. Here's why:

* **Different frameworks:** The Simulator uses Mac OS X versions of the low-level system frameworks, instead of the actual frameworks that run on the device.
* **Different hardware and memory:** The Simulator uses the Mac hardware and memory. To really determine how your app is going to perform on an honest-to-goodness iPad, you're going to have to run it on a real iPad. (Lucky for you, I show you how to do that in Chapter 6.)
* **Different installation procedure:** Xcode installs *your* app in the Simulator automatically when you build the app using the Simulator SDK. All fine and dandy, but there's no way to get Xcode to install *other* apps from the App Store in the Simulator.
* **Lack of GPS:** You can't fake the Simulator into thinking it's laying on the beach at Waikiki. The location reported by the ~~CoreLocation~~ framework in the Simulator is fixed at
    * Latitude: 37.3317 North
    * Longitude: 122.0307 West

  Which just so happens to be 1 Infinite Loop, Cupertino, CA 95014, and guess who "lives" there?
* **Two-finger limit:** You can simulate a maximum of two fingers. If your application's user interface can respond to touch events involving more than two fingers, you'll need to test that on an actual iPad. The motion of the two fingers is limited in the Simulator @@md you can't do two-figure swipes or drags.
* **Accelerometer differences:** You can access your computer's accelerometer (if it has one) through the ~~UIKit~~ framework. Its reading, however, will differ from the accelerometer readings on an iPad (for some technical reasons I won't get into).
* **Differences in rendering:** OpenGL ES (OpenGL for Embedded Systems), one of the 3D graphics libraries that works with the iOS SDK, uses renderers on devices

that are slightly different from those it uses in Simulator. As a result, a scene on the Simulator and the same scene on a device may not be identical at the pixel level.

Customizing Xcode to Your Liking

Xcode gives you options galore; I'm guessing you won't change any of them until you have a bit more programming experience under your belt, but a few options are actually worth thinking about now.

1. **With Xcode open, choose Xcode@@-->Preferences from the main menu.**
2. **Click the Debugging tab to display the Debugging pane, as shown in Figure 5-8.**

   The Xcode Preferences window refreshes to show the Debugging pane.

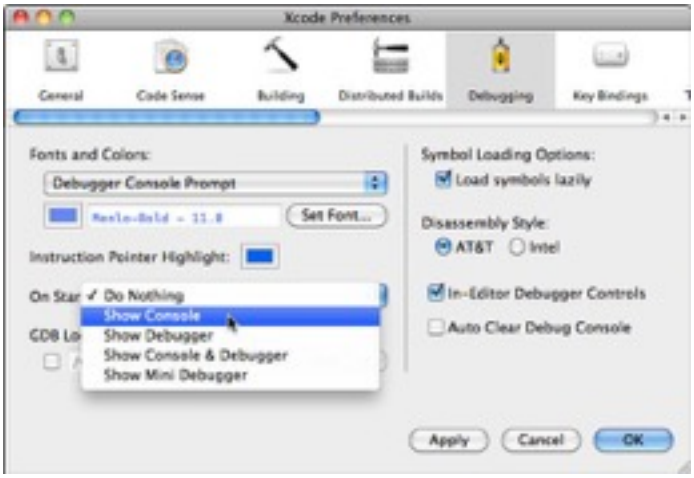

Figure 5-8: Show the console on startup.

3. **Open the On Start pop-up menu and choose Show Console (as shown in Figure 5-8). Then click Apply.**

   This step automatically opens the Console after you build your app, so you won't have to take the extra step of opening the Console to see your app's output @@md I explain the Console in Chapter 12.
4. **Click the Building tab to show the Building pane, as shown in Figure 5-9.**
5. **In the Build Results Window section of the Building pane, choose either the On Errors option or the Always option from the Open During Builds pop-up menu, as shown in Figure 5-9. Then click Apply.**

   The On Errors choice opens the Build Results window whenever an error occurs. The Always choice opens the window and keeps it open. (Some people find that having the Build Results window onscreen all the time makes it easier to find and fix errors.)
6. **Click the Documentation tab.**

   You may have to scroll the tabs horizontally to see the Documentation tab.
7. **Select the Check for and Install Updates Automatically check box and then click the Check and Install Now button.**

   This step ensures that the documentation remains up-to-date and also allows you to load and access other documentation.

8. **Click OK to close the Xcode Preferences window.**

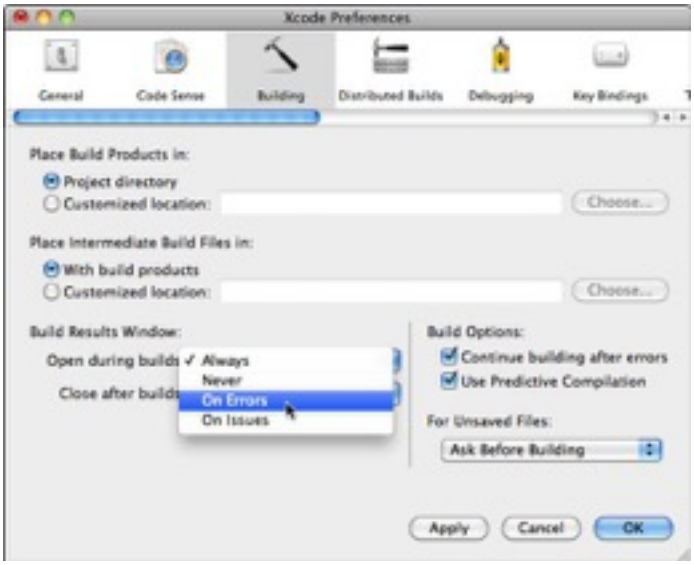

Figure 5-9: Set the option to show the Build Results window.

<Tip>

Set the tab width and other formatting options in the Indentation pane of the Preferences window.

You can also have the Text Editor show line numbers. If you click Text Editing in the Xcode Preferences toolbar to show the Text Editing pane, you can select the Show Line Numbers check box under Display Options.

Using Interface Builder

Interface Builder is a great tool for graphically laying out your user interface. You can use it to design your app's user interface and then save what you've done as a resource file, which is then loaded into your app at runtime. This resource file is then used to automatically create the single window, as well as all your views and controls, and some of your app's other objects @@md view controllers, for example. (For more on view controllers and other application objects, check out Chapter 7.)

<Tip>

If you don't want to use Interface Builder, you can also create your objects programmatically @@md creating views and view controllers and even things like buttons and labels using your very own application code. Often Interface Builder makes things easier, but sometimes just coding it is the best way.

Here's how Interface Builder works:

1. **In your Project window's Groups & Files list, select the Resources group.**

   The Detail view shows the files in the Resources group, as shown in Figure 5-10.

2. **Double-click the ~~DeepThoughtsViewController.xib~~ file in the Detail view. (Refer to Figure 5-14.)**

<Remember>

Note that ~~DeepThoughtsViewController.h~~ is still in the Editor window; that's okay because you're set to open its associated ~~DeepThoughtsViewController.xib~~ file in Interface Builder, not in the Editor window. That's because double-clicking always opens a file in a new window @@md this time, the Interface Builder window.

What you see after double-clicking are the windows as they were the last time you left them (for this project). If this is the first time you've opened Interface Builder, you see windows that look something like those in Figure 5-11.

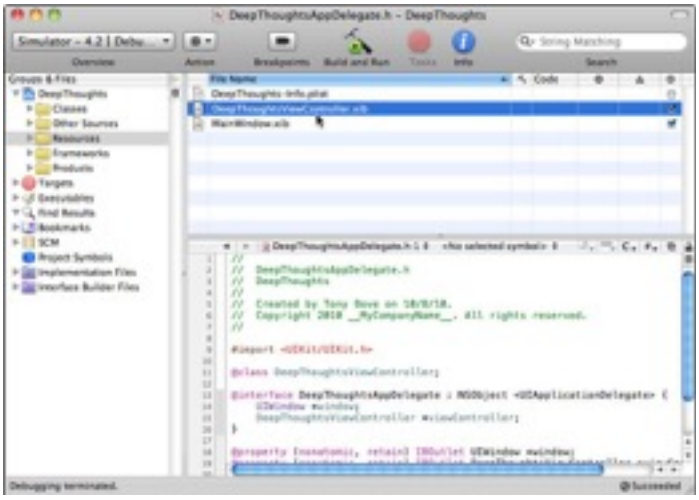

Figure 5-10: Double-click the .xib file to launch Interface Builder

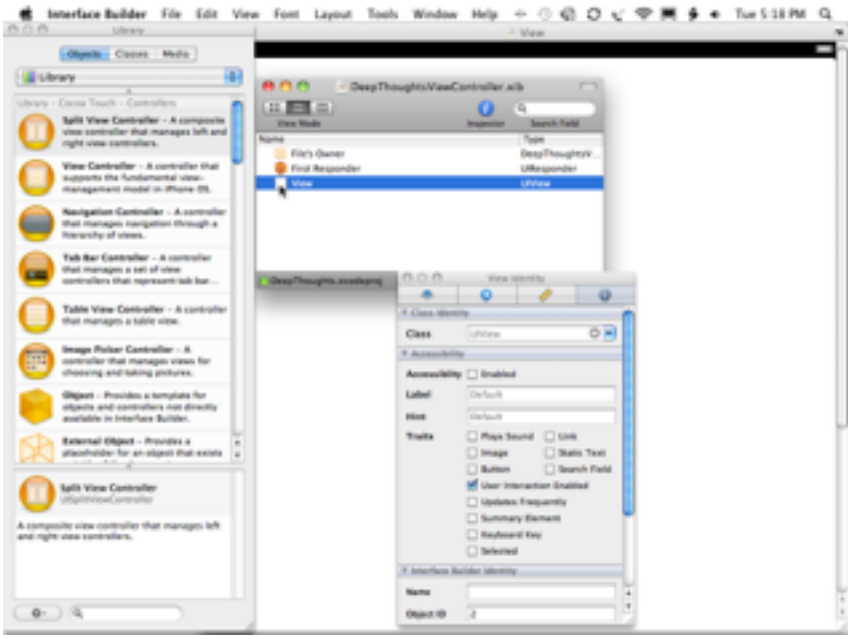

Figure 5-11: The .xib file in Interface Builder.

<TechnicalStuff>

Interface Builder supports two file types: an older format that uses the extension ~~.nib~~ and a newer format that utilizes the extension ~~.xib~~. The iPad project templates all

use ~~.xib~~ files. Although the file extension is ~~.xib~~, everyone still calls them *nib files*. The term *nib* and the corresponding file extension ~~.xib~~ are acronyms for NeXT Interface Builder. The Interface Builder application was originally developed at NeXT Computer, whose OPENSTEP operating system was used as the basis for creating Mac OS X.

The window labeled ~~DeepThoughtsViewController.xib~~ (the top center window in Figure 5-11) is the nib's main window. It acts as a table of contents for the nib file. With the exception of the first two icons (File's Owner and First Responder), every icon in this window (in this case, there's only one, View, but you'll find more as you get into nib files) represents a single instance of an Objective-C class that will be created automatically for you when this nib file is loaded, as I describe in Chapter 8

<Remember>

Interface Builder doesn't generate any code that you have to modify or even look at. Instead, it creates the ingredients for "instant" Objective-C objects that the nib loading code combines and turns into real objects at runtime.

If you were to take a closer look at the three objects in the ~~DeepThoughtsViewController.xib~~ file window (refer to Figure 5-11) @@md and if you had a pal who knew the iPad backwards and forwards @@md you'd find out the following about each object:

* **The File's Owner proxy object:** This is the controller object that's responsible for the contents of the nib file. In this case, the File's Owner object is actually the ~~DeepThoughtsViewController~~ that was created by Xcode. The File's Owner object is not created from the nib file. It's created in one of two ways: either from another (previous) nib file or by a programmer who codes it manually.
* **First Responder proxy object:** This object is the first entry in an app's dynamically constructed responder chain (a term I explain in Chapter 8) and is the object with which the user is currently interacting. If, for example, the user taps a text field to enter some data, the First Responder would then become the Text Field object.

<Remember>

Although you might use the First Responder mechanism quite a bit in your apps, there's actually nothing you have to do to manage it. It's automatically set and maintained by the ~~UIKit~~ framework.

* **View object:** The View icon represents an instance of the ~~UIView~~ class of objects. A ~~UIView~~ class of object is an area (in this case, the view) that a user can see and interact with.

If you take another look at Figure 5-11, you notice three other windows open besides the main window. Look at the View window (the one with "View" in the window's title, which appears behind and partially hidden by the other windows). In the far-right corner of the top of the View window, you would see the battery icon for the iPad in the black simulated status bar. That window is the graphical representation of the View icon in your app @@md how your new app appears on the iPad display.

<Remember>

If you close the View window and then double-click the View icon in the ~~DeepThoughtsViewController.xib~~ window, this View window opens again.

Not surprisingly (because you haven't added any data or unique code to your app yet), the View window shows the same view @@md a white screen with the black status bar and battery icon @@md as the Simulator shows when it runs your bare-bones template-

based app. (Refer to Figure 5-7.) This window is your canvas for creating your user interface: It's where you drag user-interface elements such as buttons and text fields.

These buttons, text fields, and other objects come from the Library window (the leftmost window in Figure 5-11). If the Library window isn't open, select Tools@@-->Library to open it. The Library window contains all the stock Cocoa Touch objects that Interface Builder supports. (Cocoa Touch is an application programming interface for building apps to run on the iPad, iPhone, or iPod touch.) Dragging an item from the Library to the View window adds an object of that type to the View. You start adding objects to the DeepThoughts view in Chapter 9.

<Tip>

If you happen to close the Library window, whether by accident or by design, you can get it to reappear by choosing Tools@@-->Library.

The Inspector window is also open in Figure 5-11 @@md four icons across the top from left to right correspond to the Attributes, Connections, Size, and Identity Inspectors, respectively, in the Tools menu. You learn more about these in Chapter 9.

It's Time to Get Real

Well, you still have quite a bit more to explore. But before you look behind the curtain of the iPad screen to see how iPad apps *really* run (and there's no fake Wizard of Oz back there, as I explain in Part III), and certainly before you start adding code to your first sample app in Part IV, it helps to know more about the app publishing process, how to provision your app for development, and the App Store do's and don'ts (discussed in Chapter 6).

When you've had a stroll through those adventures, you'll know everything you need to know about provisioning your app for the App Store, and designing an app that customers might actually want. (How's that for a plan?)

Chapter 7

Looking Behind the Screen

In This Chapter

* Seeing how applications actually work
* Understanding how to use the fundamental design patterns
* Doing Windows (even if you say you don't)
* Exploring an app with a view
* Manipulating view controllers
* Listing the frameworks you can use

One thing that makes iPad software development so appealing is the richness of the tools and frameworks provided in the iOS Software Development Kit (SDK). The *frameworks* are especially important; each one is a distinct body of code that actually implements your application's generic functionality @@md frameworks give the application its basic way of working, in other words. This is especially true of one framework in particular: the ~~UIKit~~ framework, which is the heart of the user interface.

In this chapter, you find out about most of the iPad's user interface architecture, which is a mostly static view that explains what the various pieces are, what each does, and how they interact with each other. This chapter lays the groundwork for developing the DeepThoughts app's user interface, which you get a chance to tackle in Chapter 9.

Using Frameworks

A *framework* offers common code providing generic functionality. iOS, the operating system for the iPad, provides a set of frameworks for incorporating technologies, services, and features into your apps. For example, the ~~UIKit~~ framework gives you event-handling support, drawing support, windows, views, and controls you can use in your app.

A framework is designed to easily integrate your code that runs your game or delivers the information that your user wants. Frameworks are similar to software libraries, but with an added twist: They also *implement* a program's flow of control (unlike a software library whose components are arranged by the programmer into a flow of control). This means that, instead of the programmer deciding the order that things should happen @@md such as which messages are sent to which objects and in what order when an application launches, or when a user touches a button on the screen @@md the order is a part of the framework and doesn't need to be specified by the programmer.

When you use a framework, you provide your app with a ready-made set of basic functions; you've told it, "Here's how to act." With the framework in place, all you need to do is *add* the specific functionality that you want in the app @@md the content as well as the controls and views that enable the user to access and use that content @@md *to* the frameworks.

The frameworks and iOS provide some pretty complex functionality, such as

* Launching the app and displaying a view
* Displaying controls and responding to a user action @@md such as tapping a toggle switch, or flicking to scroll a list
* Accessing sites on the Internet, not just through a browser, but from within your own app
* Managing user preferences
* Playing sounds and movies
* The list goes on @@md you get the picture

<Tip>

Some developers talk in terms of "using a framework" @@md but your code doesn't use frameworks so much as the frameworks *use your code.* Your code provides the functions that the framework accesses; the framework needs your code in order to become an app that does something other than start up, display a blank view, and then end. This perspective makes figuring out how to work with a framework much easier. (For one thing, it lets the programmer know where he or she is essential.)

If this seems too good to be true, well, okay, it is @@md all that complexity (and convenience) comes at a cost. It can be really difficult to get your head around the whole thing and know exactly where (and how) to add your app's functionality to that supplied by the framework. That's where *design patterns* come in. Understanding the design patterns behind the frameworks gives you a way of thinking about a framework @@md especially ~~UIKit~~ @@md that doesn't make your head explode.

Using Design Patterns

A major theme of this chapter is the fact that, when it comes to iPad app development, the ~~UIKit~~ framework does a lot of the heavy lifting for you. That's all well and good, but it's a little more complicated than that: The framework is designed around certain programming paradigms, also known as *design patterns.* The design pattern is a model that your own code must be consistent with. In programming terms, a design pattern is a commonly used template that gives you a consistent way to get a particular task done.

To understand how to take best advantage of the power of the framework @@md or (better put) how the framework objects want to use *your code* best @@md you need to understand design patterns. If you don't understand them or if you try to work around them because you're sure you have a "better" way of doing things, your job will actually be much more difficult. (Developing software can be hard enough, so making your job more difficult is definitely something you want to avoid.) Getting a handle on the basic design patterns used (and expected by) the framework helps you develop an app that makes the best use of the framework. This means the least amount of work in the shortest amount of time.

<Remember>

The design patterns can help you to understand not only how to structure your code, but also how the framework itself is structured. They describe relationships and interactions between classes or objects, as well as how responsibilities should be distributed amongst classes so the iPad does what you want it to do.

You need to be comfortable with these basic design patterns:

* Model-View-Controller (MVC)

* Delegation
* Block Objects
* Target-Action
* Managed Memory Model

Of these, the Model-View-Controller design pattern is the key to understanding how an iPad app works. I defer the discussion of the others until after you get the MVC under your belt.

<TechnicalStuff>

There's actually another basic design pattern out there: Threads and Concurrency. This pattern enables you to execute tasks concurrently (including the use of Grand Central Dispatch, that aiding and abetting feature introduced in OS X Snow Leopard for ramping up processing speed) and is way beyond the scope of this book.

The Model-View-Controller (MVC) pattern

The iOS frameworks for iPad development are *object-oriented.* The easiest way to understand what that really means is to think about a team. The work that needs to get done is divided up and assigned to individual team members (objects). Every member of a team has a job and works with other team members to get things done. What's more, a good team doesn't butt in on what other members are doing @@md just like how an object in object-oriented programming spends its time taking care of business and not caring what the object in the virtual cubicle next door is doing.

<TechnicalStuff>

Object-oriented programming was originally developed to make code more maintainable, reusable, extensible, and understandable (what a concept!) by tucking all the functionality behind well-defined interfaces. The actual details of how something works (as well as its data) are hidden, which makes modifying and extending an application much easier.

Great @@md so far @@md but a pesky question still plagues programmers:

> *Exactly how do you decide on the objects and what each one does?*

Sometimes the answer to that question is pretty easy @@md just use the real world as a model. (Eureka!) In the iPadTravel411 app that serves as an example in Part V, some of the classes of model objects are ~~Airport~~ and ~~Currency~~. But when it comes to a generic program structure, how *do* you decide what the objects should be? That may not be so obvious.

The MVC pattern is a well-established way to group application functions into objects. Variations of it have been around at least since the early days of Smalltalk, one of the very first object-oriented languages. The MVC is a high-level pattern @@md it addresses the architecture of an application and classifies objects according to the general roles they play in an application.

The MVC pattern creates, in effect, a miniature universe for the application, populated with three kinds of objects. It also specifies roles and responsibilities for all three objects and specifies the way they're supposed to interact with each other. To make things more concrete (that is, to keep your head from exploding), imagine a big, beautiful, 60-inch flat screen TV. Here's the gist:

* **Model objects:** These objects together comprise the content "engine" of your app. They contain the app's data and logic @@md making your app more than just a pretty face. In the iPadTravel411 application, for example, the model "knows" the various ways to get from Heathrow Airport to London as well as some logic to decide the best alternative based on time of day, price, and some other considerations. (You find out about adding data models in Chapter 17.)

  You can think of the *model* (which may be one object or several that interact) as a particular television program, one that, quite frankly, does not give a hoot about what TV set it is being shown on.

  In fact, the model shouldn't give a hoot. Even though it owns its data, it should have no connection at all to the user interface and should be blissfully ignorant about what is being done with its data.

* **View objects:** These objects display things on the screen and respond to user actions. Pretty much anything you can see is a kind of view object @@md the window and all the controls, for example. Your views know how to display information that they get from the model object and how to get any input from the user the model may need. But the view objects themselves should know nothing about the model. A view object may handle a request to tell the user the fastest way to London, but it doesn't bother itself with what that request means. It may display the different ways to get to London, although it doesn't care about the content options it displays for you.

  You can think of the *view* as a television screen that doesn't care about what program it's showing or what channel you just selected.

<Tip>

The ~~UIKit~~ framework provides many different kinds of views, as you'll find out later on in "Working with Windows and Views" in this chapter.

If the view knows nothing about the model, and the model knows nothing about the view, how do you get data and other notifications to pass from one to the other? To get that conversation started (Model: "I've just updated my data." View: "Hey, give me something to display," for example), you need the third element in the MVC triumvirate, the controller.

* **Controller objects:** These objects connect the application's view objects to its model objects. They supply the view objects with what they need to display (getting it from the model) and also provide the model with user input from the view.

  You can think of the *controller* as the circuitry that pulls the show off of the cable and then sends it to the screen or requests a particular pay-per-view show.

The MVC in action

Imagine that an iPad user is at Heathrow Airport, and he or she starts the handy iPadTravel411 app mentioned so often in these pages. The view will display his or her location as "Heathrow Airport." The user may tap a button (a view) that requests the weather. The controller interprets that request and tells the model what it needs to do by sending a message to the appropriate method in the model object with the necessary parameters. The model accesses the appropriate Web site (or fails to access it, due to the lack of an Internet connection), and the controller then delivers that information to the view, which promptly displays the information @@md either the appropriate page from the Web site or the ~~Weather is not available~~ offline message.

All this is illustrated in Figure 7-1.

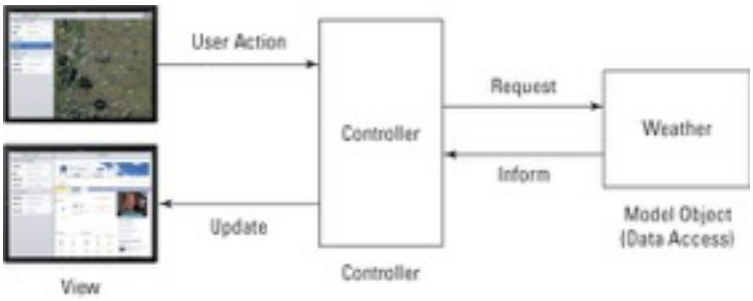

Figure 7-1: Models, controllers, and views.

<Remember>

When you think about your application in terms of Model, View, and Controller objects, the ~~UIKit~~ framework starts to make sense. It also begins to lift the fog from where at least part of your application-specific behavior needs to go. Before I get more into that, however, you need to know a little more about the classes provided to you by the ~~UIKit~~ that implement the MVC design pattern @@md windows, views, and view controllers.

Working with Windows and Views

iPad apps have only a single window. When your application is running @@md even though other apps may be hibernating or running in the background @@md your app's interface takes over the entire screen.

Looking out the window

The single window you see displayed on the iPad is an instance of the ~~UIWindow~~ class. This window is created at launch time, either programmatically by you or automatically by ~~UIKit~~ loading it from a *nib* file @@md a special file that contains instant objects that are reconstituted at runtime. (You can find out more about nib files in Chapter 5.) You then add views and controls to the window. In general, after you create the Window object (that is, if you create it instead of having it done for you), you never really have to think about it again.

<Remember>

A user can't directly close or manipulate an iPad window. It's your app that manages the window.

Although your app never creates more than one window at a time, iOS can support additional windows on top of your window. The system status bar is one example. You can also display alerts on top of your window by using the supplied Alert views.

Figure 7-2 shows the window layout on the iPad for the iPadTravel411 app.

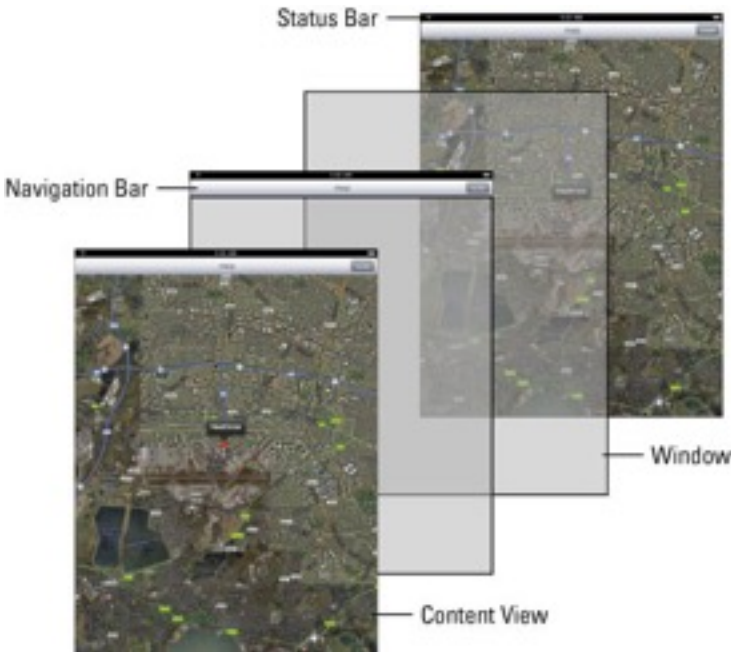

Figure 7-2: The iPadTravel411 app window layout.

Admiring the view

In an iPad app world, view objects are responsible for the view functionality in the Model-View-Controller architecture.

A view is a rectangular area on the screen (on top of a window). The *Content view* is that portion of data and controls that appear between the upper and lower bars shown in Figure 7-2.

<TechnicalStuff>

In the ~~UIKit~~ framework, windows are really a special kind of view, but for purposes of this discussion, I'm talking about views that sit on top of the window.

What views do

Views are the main way for your app to interact with a user. This interaction happens in two ways:

* **Views display content.** For example, by making drawing and animation happen onscreen.

  In essence, the view object displays the data from the model object.

* **Views handle touch events.** They respond when the user touches a button, for example.

  Handling touch events is part of a *responder chain* (a special logical sequence detailed in Chapter 8).

The view hierarchy

Views and subviews create a view hierarchy. There are two ways of looking at it (no pun intended this time): visually (how the user perceives it) and hierarchically (how you

structure it). You must be clear about the differences, or you will find yourself in a state of confusion that resembles Times Square on New Year's Eve.

Looking at it visually, the window is at the base of this hierarchy with a *Content view* on top of it (a transparent view that fills the window's Content rectangle). The Content view displays information and also allows the user to interact with the application, using (preferably standard) user-interface items such as text fields, buttons, toolbars, and tables, all of which are specialized kinds of views.

In your program, that relationship is different. The Content view is added to the window view as a *subview.*

* Views added to the Content view become *subviews* of it.
* Views added to the Content view become the *superviews* of any views added to them.
* A view can have one (and only one) superview and zero or more subviews.

<Remember>

It seems counterintuitive, but a subview is displayed *on top of* its parent view (that is, on top of its superview). Think about this relationship as containment: A superview *contains* its subviews. Figure 7-3 shows an example of a view hierarchy @@md "A Content View", with A, B, and C subviews.

Controls @@md such as buttons, text fields, and the like @@md are really view subclasses that become subviews. So are any other display areas you may specify. The view must manage its subviews, as well as resize itself with respect to its superviews. Fortunately, much of what the view must do is already coded for you. The ~~UIKit~~ framework supplies the code that defines view behavior.

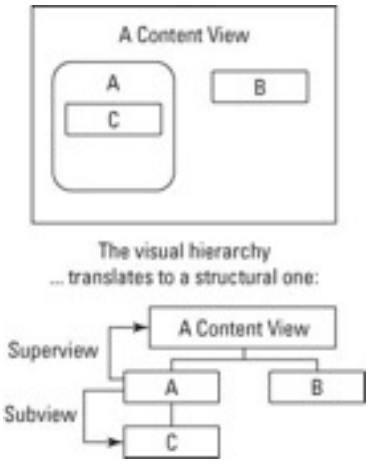

Figure 7-3: The view hierarchy is both visual and structural.

The view hierarchy plays a key role in both drawing and event handling. When a window is sent a message to display itself, the window asks its subview to render itself first. If that view has a subview, it asks *its* subview to render itself first, going down the structural hierarchy (or up the visual structure) until the last subview is reached. It then renders itself and returns to its caller, which renders itself, and so on.

You create or modify a view hierarchy whenever you add a view to another view with Interface Builder (or if you add a view programmatically). The ~~UIKit~~ framework automatically handles all the relationships associated with the view hierarchy.

<Tip>

Developers typically gloss over this visual versus hierarchical view when starting out @@md and without understanding this, it's really difficult to get a handle on what's going on.

The kinds of views you use

The ~~UIView~~ class defines the basic properties of a view, and you may be able to use it as is @@md like you do in the DeepThoughts app @@md by simply adding some controls.

The framework also provides you with a number of other views that are subclassed from ~~UIView~~. These views implement the kinds of things that you as a developer need to do on a regular basis.

<Warning>

It's important to use the view objects that are part of the ~~UIKit~~ framework. When you use an object such as a ~~UISlider~~ or ~~UIButton~~, your slider or button behaves just like a slider or button in any other iPad app. This enables the consistency in appearance and behavior across apps that users expect. (For more on how this kind of consistency in a user interface is one of the characteristics of a great app, see Chapter 2.)

Container views

Container views are a technical (Apple) term for Content views that do more than just lie there on the screen and display your controls and other content.

The ~~UIScrollView~~ class, for example, adds scrolling without you having to do any work.

~~UITableView~~ inherits this scrolling capability from ~~UIScrollView~~ and adds the ability to display lists and respond to the selections of an item in that list. Think of the Contacts application (and a host of others).

Another container view, the ~~UIToolbar~~ class, contains button-like controls @@md and you find those everywhere on the iPad. In Mail, for example, you tap an icon button in the toolbar to respond to an e-mail. Toolbars can be positioned at the top and bottom of a view. If you're familiar with iPhone apps, keep in mind that the iPad's increased screen size makes it possible to include more items on a toolbar.

Controls

Controls are the fingertip-friendly graphics you see extensively used in a typical application's user interface. Controls are actually subclasses of the ~~UIControl~~ superclass, a subclass of the ~~UIView~~ class. They include touchable items like buttons, sliders, and switches, as well as text fields in which you enter data.

Controls make heavy use of the Target-Action design pattern, which you get to see with the Done button in the DeepThoughts app in Chapter 11.

Display views

Think of *Display views* as controls that look good, but don't really do anything except, well, look good. These include ~~UIImageView~~, ~~UILabel~~, ~~UIProgressView~~, and ~~UIActivityIndicatorView~~. (You use ~~UILabel~~ in the DeepThoughts app in Chapter 10 to display the area in which the falling words appear.)

Text and Web views

Text and *Web views* provide a way to display formatted text in your application. The ~~UITextView~~ class supports the display and editing of multiple lines of text in a scrollable area. The ~~UIWebView~~ class provides a way to display HTML content. These views can be used as the Content view, or they can be used in the same way as a Display view (that is, as a subview of a Content view). You encounter ~~UIWebView~~ in the iPadTravel411 app in Chapter 17, which you use to display the Weather view. ~~UIWebView~~ also is the primary way to include graphics and formatted text in Text Display views.

Alert views and action sheets

Alert views and *action sheets* present a message to the user, along with buttons that allow the user to respond to the message. Alert views and action sheets are similar in function but look and behave differently. For example, the ~~UIAlertView~~ class displays a blue alert box that pops up on the screen, and the ~~UIActionSheet~~ class displays a box that slides in from the bottom of the screen.

Navigation views

Tab bars and *navigation bars* work in conjunction with view controllers to provide tools for navigating in your app. Normally, you don't need to create a ~~UITabBar~~ or ~~UINavigationBar~~ directly @@md it's easier to use Interface Builder or configure these views through a tab bar or navigation bar controller.

The window

A *window* provides a surface for drawing content and is the root container for all other views.

Controlling View Controllers

View controllers implement the controller component of the Model-View-Controller design pattern. These Controller objects contain the code that connects the app's view objects to its model objects. They provide the data to the view. Whenever the view needs to display something, the view controller goes out and gets what the view needs from the model. Similarly, view controllers respond to controls in your Content view and may do things like tell the model to update its data (when the user adds or changes text in a text field, for example); or compute something (the current value of, say, your U.S. dollars in British pounds); or change the view being displayed (as with choosing Weather in the iPadTravel411 app).

As shown in "The Target-Action pattern" section later in this chapter, a view controller is often the (target) object that responds to the onscreen controls. The Target-Action mechanism is what enables the view controller to be aware of any changes in the view, which can then be transmitted to the model. For example, when the user taps the Weather entry in the iPadTravel411 app to request the current weather conditions, the following occurs:

Comp Services: Please do not bold the following numbered list.

1. A message is sent to that view's view controller to handle the request.
2. The view controller's method interacts with the Weather model object.
3. The model object processes the request from the user for the current weather.
4. The model object sends the data back to the view controller.
5. The view controller creates a new view to present the information.

View controllers have other vital iPad responsibilities as well, such as the following:

* Managing a set of views @@md including creating them, or flushing them from memory during low-memory situations.
* Responding to a change in the device's orientation @@md say, landscape to portrait @@md by resizing the managed views to match the new orientation.
* Creating a *Modal* view, which is a child window that displays a dialog requiring the user to do something (tap the Yes button, for example) before returning to the application.

<Remember>

You would use a Modal view to ensure the user has paid attention to the implications of an action (for example, "Are you sure you want to delete all your contacts?").

<Remember>

Apple recommends that apps should support all iPad landscape and portrait orientations when appropriate @@md Apple's own Keynote app, for example, runs only in landscape orientations. You'll want to use a view controller just to manage a single view and auto-rotate it when the device's orientation changes. The app's window and view controllers provide the basic infrastructure needed to support rotations @@md you can use the existing infrastructure as is or customize the behavior to suit the particulars of your app, as you do with the iPadTravel411 app in Chapter 15.

What about the Model?

As this chapter shows (and as you will continue to discover), a lot of the functionality you need is already in the frameworks.

But when it comes to the model objects, for the most part, you're pretty much on your own. You need to design and create model objects to hold the data and carry out the logic. In the iPadTravel411 app in Chapter 14, for example, you create an ~~Airport~~ object that knows the different ways to get into the city that it supports.

<Remember>

You may find classes in the framework that help you get the nuts and bolts of the model working. But the actual content and specific functionality is up to you. As for actually implementing model objects, you find out how to do that in Chapter 17.

Using naming conventions

When creating your own classes, it's a good idea to follow a couple of standard framework-naming conventions:

- Class names (such as ~~View~~) should start with a capital letter.
- The names of methods (such as ~~viewDidLoad~~) should start with a lowercase letter.
- The names of instance variables (such as ~~frame~~) should start with a lowercase letter.

When you do it this way, it makes it easier to understand what something actually is just from the name.

Adding Your Own Application's Behavior

Earlier in this chapter (by now it probably seems like a million years ago), I mention other design patterns used in addition to the Model-View-Controller (MVC) pattern. Three of these patterns @@md the Delegation pattern, the Target-Action pattern, and the Block Object pattern, along with the MVC pattern and subclassing, provide the mechanisms for you to add your app-specific behavior to the ~~UIKit~~ (and any other) framework.

<Remember>

The first way to add behavior is through model objects in the MVC pattern. Model objects contain the data and logic that make, well, your application.

The second way, the way people traditionally think about adding behavior to an object-oriented program, is through *subclassing,* where you first create a new (sub) class that inherits behavior and instance variables from another (super) class and then add additional behavior, instance variables, and *properties* to the mix until you come up with just what you want. (I explain properties in Chapter 11.) The idea here is to start with something basic and then add to it @@md kind of like taking a deuce coupe (1932 Ford) and turning it into a hot rod. You'd subclass a view controller class, for example, to respond to controls.

The third way to add behavior involves using the Delegation pattern, which allows you to customize an object's behavior without subclassing by basically forcing another object to do the first object's work for it. For example, the Delegation design pattern is used at application startup to invoke a method ~~applicationDidFinishLaunching:~~ that gives you a place to do your own application-specific initialization. All you do is add your code to the method.

The fourth way to add behavior is by using Block objects. The Block Object design pattern is similar to Delegation, but it's more "event driven" in that it allows you to create methods or functions that you can pass to other methods or functions that are executed as needed. For example, you might want to have some code that scrolls the view as necessary when the keyboard appears. You would pass that to a method that's invoked when the keyboard appears.

The fifth way to add behavior involves the Target-Action design pattern, which allows your application to respond to an event. When a user taps a button, for example, you specify what method should be invoked to respond to the button tap. What is interesting about this pattern is that it also requires subclassing @@md usually a view controller @@md in order to add the code to handle the event.

The next few sections go into a little more detail about Delegation patterns and Target-Action patterns.

The Delegation pattern

Delegation is a pattern used extensively in the iOS frameworks for iPad and iPhone apps, so much so that it's very important to clearly understand it. In fact, once you understand it, your life will be much easier.

Delegation, as I mention in the previous section, is a way of customizing the behavior of an object without subclassing it. Instead, one object (a Framework object) delegates the

task of implementing one of its responsibilities to another object. You're using a behavior-rich object supplied by the framework *as is,* and putting the code for program-specific behavior in a separate (delegate) object. When a request is made of the Framework object, the method of the delegate that implements the program-specific behavior is automatically called.

For example, the ~~UIApplication~~ object handles most of the actual work needed to run the application. But, as you will see in Chapter 8, it sends your application delegate the ~~application:didFinishLaunchingWithOptions:~~ message to give you an opportunity to restore the application's window and view to where it was when the user previously left off. You can also use this method to create objects that are unique to your app.

When a Framework object has been designed to use delegates to implement certain behaviors, the behaviors it requires (or gives you the option to implement) are defined in a *protocol*.

Protocols define an interface that the delegate object implements. Protocols can be formal or informal, although I concentrate solely on the former because it includes support for things like type checking and runtime checking to see whether an object conforms to the protocol.

In a formal protocol, you usually don't have to implement all the methods; many are declared optional, meaning you only have to implement the ones relevant to your app. Before it attempts to send a message to its delegate, the host object determines whether the delegate implements the method (via a ~~respondsToSelector:~~ message) to avoid the embarrassment of branching into nowhere if the method is not implemented.

<Remember>

You can find out much more about delegation and the Delegation pattern when you develop the DeepThoughts app in Part IV and especially the iPadTravel411 app in Part V.

The Block Object pattern

Although delegation is extremely useful, it is not the only way to customize the behavior of a method or function.

Blocks are like traditional C functions in that they are small, self-contained units of code. They can be passed in as arguments of methods and functions and then used when they're needed to do some work. Like many programming topics, understanding block objects is easier when you use them, as you do in Chapter 10.

With iOS 4, a number of methods and functions of the system frameworks are starting to take blocks as parameters, including the following:

* Completion handlers
* Notification handlers
* Error handlers
* Enumeration
* View animation and transitions
* Sorting

In Chapter 10, you use the block-based animation method ~~animateWithDuration:animations:~~ to implement the animation in DeepThoughts. Block

objects also have a number of other uses, especially in Grand Central Dispatch and the ~~NSOperationQueue~~ class, the two recommended technologies for concurrent processing. But because concurrent processing is out of scope for this book (way out of scope in fact), I leave you to explore that use on your own.

The Target-Action pattern

You use the *Target-Action* pattern to let your app know that it should do something. A user may have tapped a button or entered some text, for example, and the app must do something. The control @@md a button, say @@md sends a message (the Action message) that you specify to the target you have selected to handle that particular action. The receiving object, or the Target, is usually a view controller object.

If you wanted to develop an app that could start a car from an iPad (not a bad idea for those who live in a place like Minneapolis in winter), you could display two buttons, Start and Heater. You could use Interface Builder to specify, when the user taps Start, that the target is the ~~CarController~~ object and that the method to invoke is ~~ignition~~.

<TechnicalStuff>

The Target-Action mechanism enables you to create a control object and tell it not only what object you want handling the event, but also the message to send. For example, if the user touches a Ring Bell button onscreen, you want to send a Ring Bell message to the view controller. But if the Wave Flag button on the same screen is touched, you want to be able to send the Wave Flag message to the same view controller. If you couldn't specify the message, all buttons would have to send the same message. It would then make the coding more difficult and more complex because you would have to identify which button had sent the message and what to do in response. It would also make changing the user interface more work and more error prone.

When creating your app, you can set a control's action and target through the Interface Builder. This setting allows you to specify what method in which object should respond to a control without having to write any code.

For more on the Interface Builder, check out Chapter 9.

Doing What When?

The ~~UIKit~~ framework provides a great deal of ready-made functionality, but the beauty of ~~UIKit~~ lies in the fact that @@md as this chapter explains @@md you can customize its behavior using four distinct mechanisms:

* Subclassing
* Delegation
* Target-Action
* Block Objects

One of the challenges facing a new developer is to determine which of these mechanisms to use when. (That was certainly the case for me.) To ensure that you have an overall conceptual picture of the iPad application architecture, check out the Cheat Sheet for *iPhone Application Development For Dummies,* which offers a summary of which mechanisms are used when. You can find the Cheat Sheet at ~~www.dummies.com/cheatsheet/iphoneapplicationdevelopment~~.

You still have quite a bit more background information to explore before you get started building the DeepThoughts app in Chapter 9. It helps a great deal to know more about how an app runs in iOS @@md the *runtime scenario*. Although that sounds like the title of a prison escape movie, it's really about what goes on inside the iPad when the user launches your app, and you find out all about that in the next chapter, along with how Interface Builder nib files work. What fun!

Chapter 8

Understanding How an App Runs

In This Chapter

* Watching how the template-based app works at runtime
* Following what goes on when the user launches your app
* Getting a handle on how nib files work
* Remembering memory management
* Knowing what else you should be aware of at runtime

When you create an Xcode project and select a template, as I show in Chapter 5, you get a considerable head start on the process of coding your very own iPad app. In that chapter, I choose the View-based Application template for the DeepThoughts app, and as a result, I have a working app that offers a view.

As the wise sage (and wisecracking baseball player) Yogi Berra once said, "You can observe a lot just by watching." Before you add anything more to this skeleton of an app, it helps to look at *how* it does what it already does. By uncovering the mysteries of what this template does at runtime, you can learn a bit more about where to put *your* code.

As you find out in Chapter 7, a *framework* offers common code providing generic functionality. iOS provides a set of frameworks for incorporating technologies, services, and features into your apps. The framework is designed to easily integrate your code; with the framework in place, all you need to do is *add* the specific functionality that you want in the app @@md the content as well as the controls and views that enable the user to access and use that content @@md *to* the frameworks.

App Anatomy 101 @@md The Lifecycle

The short-but-happy life of an iPad app begins when a user launches it by tapping its icon on the iPad Home screen. The system launches your app by calling its ~~main~~ function, which you can see in the Xcode Editor window in Figure 8-1.

Figure 8-1: The main function is where it all begins.

The ~~main~~ function does only three things:

* Sets up an autorelease pool:

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

* Calls the ~~UIApplicationMain~~ function to create the application object and delegate and set up the event loop. The template uses the first ~~nil~~ as the principle class name so that ~~UIApplication~~ is the assumed name, and it specifies the second ~~nil~~ to load the delegate object from the application's main nib file:

```
int retVal = UIApplicationMain(argc, argv, nil, nil);
```

* At termination, releases the autorelease pool:

```
[pool release];
return retVal;
```

(As Objective-C programmers already know, the lines beginning with // in the code shown in Figure 8-1 are comments that don't do anything.)

To be honest, this whole ~~main~~ function thing isn't something you even need to think about. What's important is what happens *when* the ~~UIApplicationMain~~ function is called. Here's the play-by-play:

1. **The main nib file is loaded.**

   A *nib file* is a resource file that contains the specifications for one or more objects. The main nib file usually contains a window object of some kind, the application delegate object, and any other key objects. When the file is loaded, the objects are reconstituted (think "instant application") in memory.

   In the DeepThoughts app you just started (with a little help from the aforementioned View-based Application template), this is the moment of truth when the ~~DeepThoughtsAppDelegate~~ and ~~DeepThoughtsViewController~~ objects are created along with the main window.

   For more on the application delegate and view controller objects and the roles they play in apps, see Chapter 7.

2. **The *application delegate* (~~DeepThoughtsAppDelegate~~) receives the ~~application:didFinishLaunchingWithOptions:~~ message.**

   You can see the ~~DeepThoughtsAppDelegate.m~~ implementation file in Figure 8-2, as provided by the template @@md see how much code is already written for you!

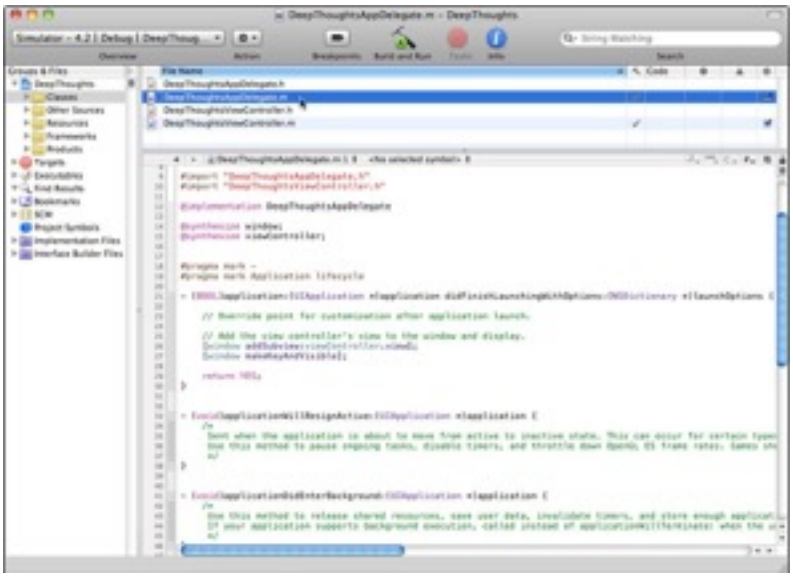

Figure 8-2: The application delegate implementation as provided by the template.

The ~~application:didFinishLaunchingWithOptions:~~ message tells the delegate when the application has launched. In this step, you initialize and set up your application. You have a choice here: You may want to display your main application window as if the user were starting from scratch, or you may want the window to look the way it did when the user last exited the application. The application delegate object is a custom object that you code. It's responsible for some of the application-level behavior of your application. (Delegation is an extensively used design pattern that I introduce in Chapter 7.)

3. **The ~~UIKit~~ framework sets up the event loop.**

   The *event loop* is the code responsible for polling input sources @@md the screen, for example. Events, such as touches on the screen, are sent to the object @@md say, a controller @@md that you have specified to handle that kind of event, as shown in Figure 8-3. These handling objects contain the code that implements what you want your app to do in response to that particular event. A touch on a control may result in a change in what the user sees in a view, a switch to a new view, or even the playing of the song "Don't Touch Me."

4. **The normal processing of your application is interrupted.**

   In iOS 4.2, your application must be able to handle situations where its normal processing is interrupted. The interruption may be temporary @@md for example, an incoming FaceTime call, calendar alerts, or the user pressing the Sleep/Wake button @@md or it may be a permanent one, such as when the user switches out of your app and your app begins the transition to its *background state,* where it's suspended to conserve power but remains in memory. iOS sends you a number of messages to let you know exactly what's happening and also give you the opportunity to do things such as save user data and *state information* @@md saving where the user was in the application. (I cover all this in "Responding to interruptions" in this chapter.)

<Remember>

Saving is important, because whether the application is interrupted, sent to its background state, or terminated, when the time comes for it to launch again (refer to Step 2) and the ~~UIApplicationMain~~ sends the application delegate the ~~applicationDidFinishLaunchingWithOptions~~ message, you can restore the application to where the user left off.

5. **When the user performs an action that would cause your app to quit, ~~UIKit~~ notifies your app and begins the termination process.**

   With iOS 4.2, the Terminator doesn't seek and destroy your app @@md it's simply moved to the inactive state and then the background state. (See "Responding to interruptions" in this chapter.) But under some circumstances, your application can in fact be terminated. (I take the time to explain what those circumstances are in "Termination" in this chapter.)

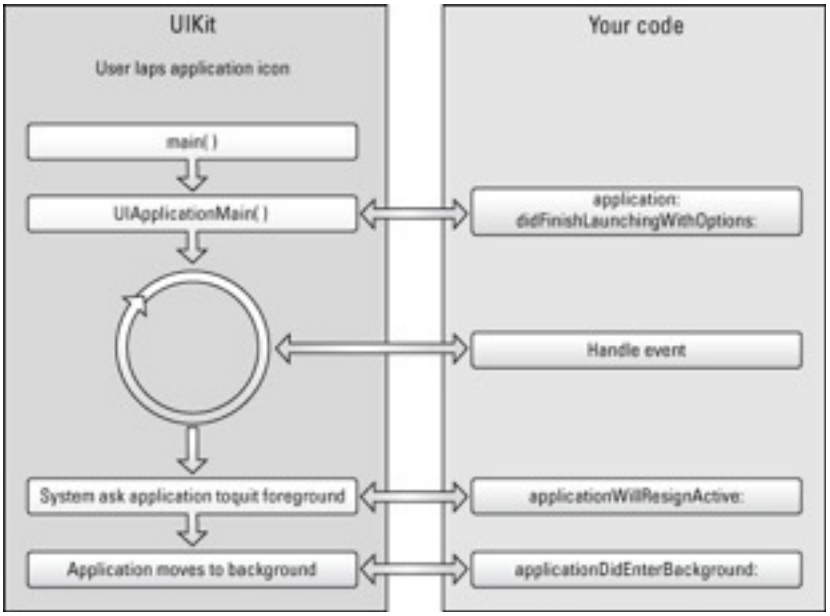

Figure 8-3: A simplified lifecycle view of an iPad application.

It all starts with the main nib file

When you create a new project using a template @@md quite the normal state of affairs, as I show in Chapter 5 @@md the basic application environment is already included. That means when you launch your app, an application object is created and connected to the window object, the run loop is established, and so on @@md despite the fact that you haven't done a lick of coding.

Most of this work is done by the ~~UIApplicationMain~~ function, as illustrated back in Figure 8-3. To take advantage of this once-in-a-lifetime opportunity to see how all this works, go back to your project window in Xcode, which you started by choosing the View-based Application template in Chapter 5, and select the Resources folder in the Groups & Files list on the left.

Here's a blow-by-blow description of what the ~~UIApplicationMain~~ function actually does:

1. **An instance of ~~UIApplication~~ is created.**
2. **~~UIApplication~~ looks in the ~~info.plist~~ file, trying to find the main nib file.**

   To see the ~~info.plist~~ file, select ~~DeepThoughts-Info.plist~~ in the Detail view of the Xcode window, as shown in Figure 8-4. The contents of the file appear in the Editor view below the Detail view.

   ~~UIApplication~~ makes its way down the Key column of the ~~info.plist~~ file until it finds the Main Nib File Base Name entry. Eureka! It peeks over at the Value

column and sees that the value for the Main Nib File Base Name entry is ~~MainWindow~~. (See Figure 8-4.)

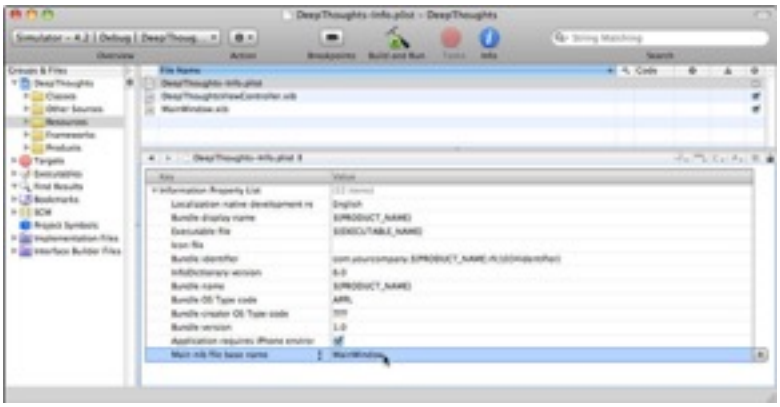

Figure 8-4: The info.plist file holds the key to the Main Nib File entry.

3. **~~UIApplication~~ loads ~~MainWindow.xib~~.**

Figure 8-5 illustrates this process of loading the main window's nib file.

The nib file ~~MainWindow.xib~~ is what causes your application's delegate, window, and view controller instances to get created at runtime. Remember, this file is provided as part of the project template. You don't need to change or do anything here. This is just a chance to see what's going on behind the scenes.

To see the nib file (~~MainWindow.xib~~) in Interface Builder, select the Resources group if it is not already selected, and double-click ~~MainWindow.xib~~ in the Xcode project window's Detail view (you can see the file in Figure 8-4.

When Interface Builder opens, take a look at the nib file's main window @@md the one labeled ~~MainWindow.xib~~ (as shown in Figure 8-6). Select File's Owner in the window, click the *i* Inspector button at the top of the window, and then click the *i* Identity tab of the Inspector window if it's not already selected (or choose Tools@@-->Identity Inspector to show the Identity tab of the Inspector window).

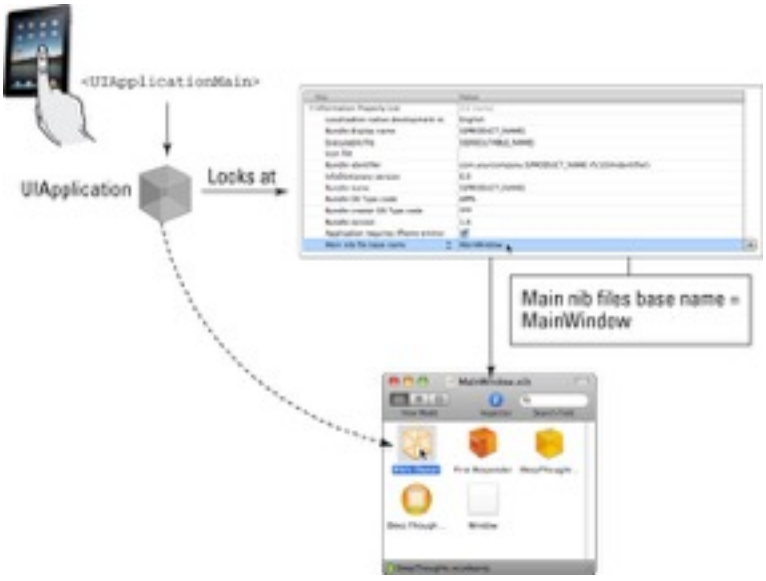

Figure 8-5: The application is launched.

The ~~MainWindow.xib~~ window shows five icons, but you can view them in a list by clicking the center View Mode button in the upper-left corner of the window, as shown in Figure 8-7. The interface objects are as follows:

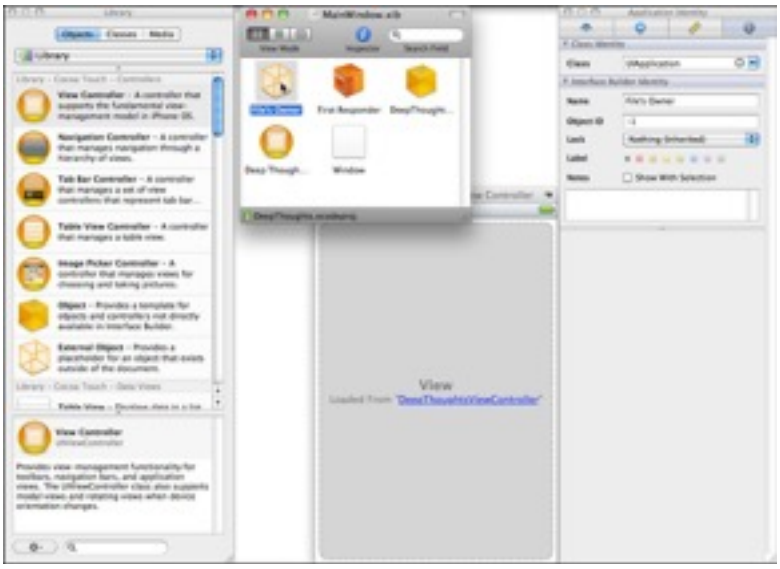

Figure 8-6: The MainWindow.xib file in Interface Builder.

* **File's Owner (proxy object):** The File's Owner @@md the object that's going to use (or *own*) this file @@md is of the class ~~UIApplication~~. This object isn't created when the file is loaded, as are the window and views @@md it's already created by the ~~UIApplicationMain~~ object before the nib file is loaded.

  ~~UIApplication~~ objects have a delegate object that implements the ~~UIApplicationDelegate~~ protocol. Specifying the delegate object can be done from Interface Builder by setting the ~~delegate~~ outlet of a ~~UIApplication~~ object. To see that this has already been done for you in the template, click File's Owner and then click the Connections tab of the Inspector window (or choose Tools@@-->Connections Inspector). The ~~delegate~~ outlet is set to "DeepThoughts App Delegate," as shown in Figure 8-7 @@md click the outlet connection in the Inspector window and DeepThoughts App Delegate is highlighted in the ~~MainWindow.xib~~ window.

* **First Responder (proxy object):** This object is the first entry in an application's responder chain, which is constantly updated while the application is running @@md usually to point to the object that the user is currently interacting with. If, for example, the user were to tap a text field to enter some data, the first responder would become the text field object.

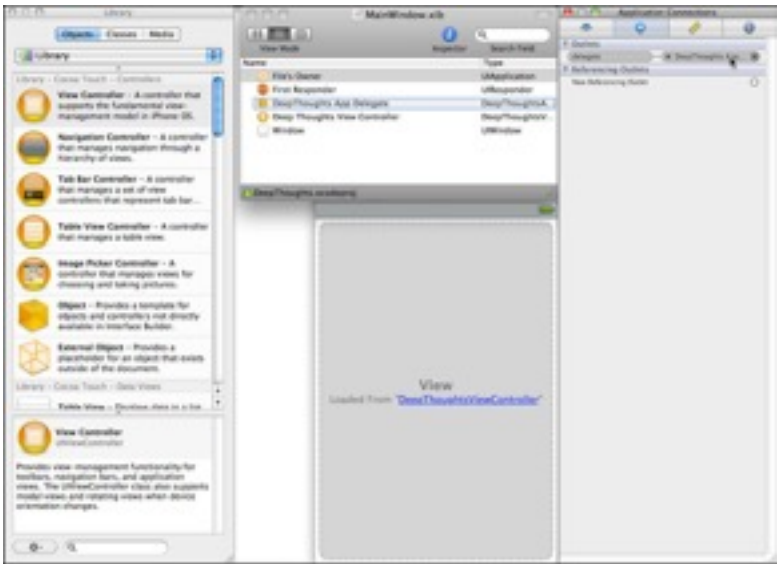

Figure 8-7: The MainWindow.xib in Interface Builder with File's Owner selected and Connections displayed.

* **Window:** The window has its background set to white and status bar set to gray, as shown in Figure 8-8, and is *not* set to be visible at launch. To see the window's attributes, click Window in the ~~MainWindow.xib~~ window and click the Attributes tab in the Inspector window.

* **An instance of ~~DeepThoughtsAppDelegate~~ set to be the application's delegate.**

  You can see the header and implementation of ~~DeepThoughtsAppDelegate~~ in Listing 1-1 and 1-2 in the next section. This is where you can put code that restores the app after launch to its previous *state* (see "Responding to interruptions" in this chapter for the meaning of this), or performs any other custom application initialization.

* **An instance of ~~DeepThoughtsViewController~~ set to be the application's view controller.**

  The view controller is where you put your code to control the views of your app, as you find out in Chapter 10.

The following section spells out what happens to these objects when ~~UIApplication~~ loads ~~MainWindow.xib~~.

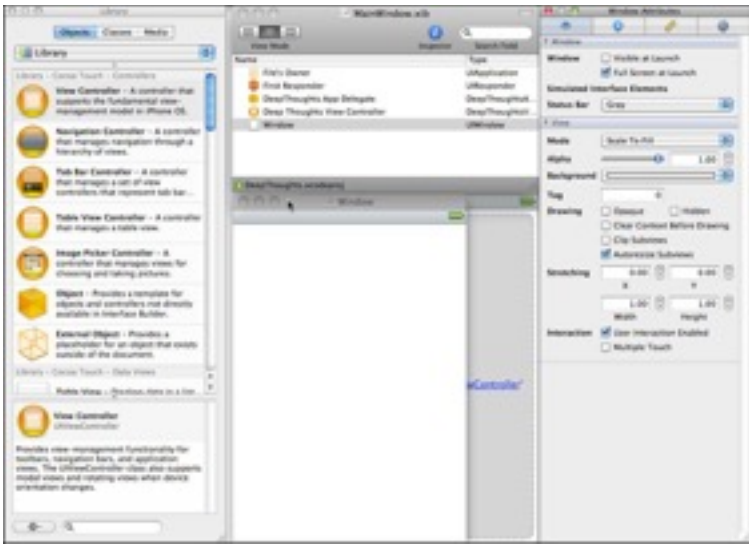

Figure 8-8: The MainWindow.xib in Interface Builder with Window selected and Attributes displayed.

Initialization

~~UIApplication~~ loads the parts of the ~~MainWindow.xib~~ file as follows:

1. **Creates ~~DeepThoughtsAppDelegate~~.**
2. **Creates ~~Window~~.**
3. **Sends the ~~DeepThoughtsAppDelegate~~ the ~~application:didFinishLaunchingWithOptions:~~ message.**
4. **~~DeepThoughtsAppDelegate~~ initializes the window.**

I show the header and implementation of ~~DeepThoughtsAppDelegate~~ in Listings 1-1 and 1-2. All this is done for you as part of the Xcode template.

Listing 1-1: DeepThoughtsAppDelegate.h

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

@class DeepThoughtsViewController;


@interface DeepThoughtsAppDelegate : NSObject
          <UIApplicationDelegate> {

    UIWindow *window;

    DeepThoughtsViewController *viewController;

}

@property (nonatomic, retain) IBOutlet UIWindow *window;

@property (nonatomic, retain) IBOutlet DeepThoughtsViewController
          *viewController;

@end
```

Listing 1-2: DeepThoughtsAppDelegate.m

```
#import "DeepThoughtsAppDelegate.h"

#import "DeepThoughtsViewController.h"

@implementation DeepThoughtsAppDelegate


@synthesize window;

@synthesize viewController;


#pragma mark -

#pragma mark Application lifecycle


- (BOOL)application:(UIApplication *)application
          didFinishLaunchingWithOptions:(NSDictionary
          *)launchOptions {


    // Override point for customization after app launch

    // Add the view controller's view to window and display

    [window addSubview:viewController.view];
```

```
    [window makeKeyAndVisible];


        return YES;

}

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

    /*

     Sent when the application is about to move from active to
        inactive state. This can occur for certain types of
        temporary interruptions (such as an incoming call) or
        when the user quits the application and it begins the
        transition to the background state.

     Use this method to pause ongoing tasks, disable timers, and
        throttle down OpenGL ES frame rates. Games should use
        this method to pause the game.

     */

}

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

    /*

     Use this method to release shared resources, save user data,
        invalidate timers, and store enough application state
        information to restore your application to its current
        state in case it is terminated later.

     If your application supports background execution, called
        instead of applicationWillTerminate: when the user
        quits.

     */

}

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

    /*

     Called as part of transition from the background to the
        inactive state: here you can undo many of the changes
        made on entering the background.

     */

}

- (void)applicationDidBecomeActive:(UIApplication *)application {
```

```
    /*
     Restart any tasks that were paused (or not yet started)
          while the application was inactive. If the application
          was previously in the background, optionally refresh
          the user interface.
     */
}
- (void)applicationWillTerminate:(UIApplication *)application {
    /*
     Called when the application is about to terminate.
     See also applicationDidEnterBackground:.
     */
}


#pragma mark -
#pragma mark Memory management


- (void)applicationDidReceiveMemoryWarning:(UIApplication
          *)application {
    /*
     Free up as much memory as possible by purging cached data
          objects that can be recreated (or reloaded from disk)
          later.
     */
}
- (void)dealloc {
    [viewController release];
    [window release];
    [super dealloc];
}
@end
```

In Listing 1-2, the view controller is initialized with the ~~applicationDidFinishLaunchingWithOptions~~ method, which you can use to do any other

application initialization as well, such as returning everything to what it was like when the user last used the application.

<Remember>

Your goal during startup should be to present your application's user interface as quickly as possible @@md quick initialization = happy users. Don't load large data structures that your application won't use right away. If your application requires time to load data from the network (or perform other tasks that take noticeable time), get your interface up and running first and then launch the slow task on a background thread. Then you can display a progress indicator or other feedback to the user to indicate that your application is loading the necessary data or doing something important.

When the ~~application:didFinishLaunchingWithOptions:~~ method is invoked, your application is in the inactive state. Unless your application does some kind of background processing, when your application becomes active, it will receive the ~~applicationDidBecomeActive:~~ message when it *enters the foreground* (becomes the application the user sees on the screen), which I explain in "Responding to interruptions" in this chapter.

<Remember>

With iOS 4.2, an application can also be launched into the background, but because the DeepThoughts application at the heart of this section doesn't do any background processing, this is the sequence I work with. And because the application does no background processing, there's also nothing it has to do in response to the ~~applicationDidBecomeActive:~~ message.

The application delegate object (refer to Listing 1-1) is usually derived from ~~NSObject~~, the root class (the very base class from which all iPad application objects are derived), although it can be an instance of any class you like, as long as it adopts the ~~UIApplicationDelegate~~ protocol. The methods of this protocol correspond to behaviors that are needed during the application lifecycle and are your way of implementing this custom behavior. Although you aren't required to implement all the methods of the ~~UIApplicationDelegate~~ protocol, every application should implement the following critical application tasks:

* Initialization, which I've just covered
* Handling events, which I cover in the next section
* Responding to interruptions, which I cover in the section following the next section
* Responding to termination, which I cover in "Termination" in this chapter
* Responding to low memory warnings, which I cover in "Observing low-memory warnings" in this chapter

Event processing

What actually happens when the user taps something to cause an event? The event is processed. The functionality provided in the ~~UIKit~~ framework manages most of the application's infrastructure. Part of the initialization process mentioned in the previous section involves setting up the main run loop and event handling code, which is the responsibility of the ~~UIApplication~~ object. Here's a rundown of how such events drive a process inside the app:

1. **You have an event @@md the user taps a button, for example.**

The touch of a finger (or lifting it from the screen) adds a touch event to the application's event queue, where it's *encapsulated* in @@md placed into, in other words @@md a ~~UIEvent~~ object. There's a ~~UITouch~~ object for each finger touching the screen, so you can track individual touches. As the user manipulates the screen with his or her fingers, the system reports the changes for each finger in the corresponding ~~UITouch~~ object.

<Tip>

My advice to you: Don't let your eyes glaze over here. This ~~UIEvent~~ and ~~UITouch~~ stuff is important, as you discover when I show you how to handle touch events in Chapter 11.

2. **The run loop monitor dispatches the event.**

   When there's something to process, the event-handling code of the ~~UIApplication~~ processes touch events by dispatching them to the appropriate *responder* object @@md the object that has signed up to take responsibility for doing something when an event happens (when the user touches the screen, for example). Responder objects can include instances of ~~UIApplication~~, ~~UIWindow~~, ~~UIView~~, and its subclasses (all which inherit from ~~UIResponder~~).

3. **A responder object decides how to handle the event.**

   For example, a touch event occurring with a button in a view is delivered to the button object. The button object handles the event by sending an action message to another object @@md in this case, the ~~UIViewController~~ object. Setting it up this way enables you to use standard button objects without having to muck about in their innards @@md just tell the button what method you want invoked in your view controller, and you're basically set.

   Processing the message may result in changes to a view, or a new view altogether, or some other kind of change in the user interface. When this happens, the view and graphics infrastructure takes over and processes the required drawing events.

4. **You're sent back to the event loop.**

   After an event is handled or discarded, control passes back to the run loop. The run loop then processes the next event or puts the thread to sleep if there's nothing more for it to do.

Responding to interruptions

On an iPad, various events besides termination can interrupt your app to allow the user to respond @@md for example, calendar alerts or the user pressing the Sleep/Wake button. Such interruptions may only be temporary. If the user chooses to ignore an interruption, your app continues running as before. If the user decides to tap the alert to deal with it, your app first moves into the *inactive state.*

When the user quits an app, its process is not terminated (as was the case with iOS 3.2 and earlier versions) @@md the app is moved to the background, where it is suspended to conserve power. (If an app needs to continue running, it can request execution time from the system.) Because the app is in the background (running or suspended) and still in memory, relaunching is nearly instantaneous. An app's objects (including its windows and views) remain in memory, so they don't need to be recreated when the app relaunches. If memory becomes constrained, iOS may purge background apps to make more room for the foreground app.

<Remember>

Because these interruptions cause a temporary loss of control by your app @@md meaning that touch events, as I describe in Chapter 11, are no longer sent to your app

@@md it's up to you to prevent what's known in the trade as a "negative user experience." For example, if your app is a game, you should pause the game. In general, your app should store information about its current state when it moves to the inactive state and be able to restore itself to the current state upon a subsequent relaunch.

In all cases, the sequence of events starts the same way @@md with the ~~applicationWillResignActive:~~ message sent to your application delegate. Using this method, you should pause ongoing tasks, disable timers, and generally put things on hold.

What happens after this depends on the nature of the interruption and/or how the user responds to the interruption. Your application may be

* Reactivated
* Moved to the background

The next two sections take a look at each scenario.

Your application is reactivated

If the user ignores the FaceTime call or calendar notification (or similar interruption), the system sends your application delegate the ~~applicationDidBecomeActive:~~ message and resumes the delivery of touch events to your application.

If the user pressed the Sleep/Wake button, the system then puts the device to sleep. When the user wakes the device later, the system sends your application delegate the ~~applicationDidBecomeActive:~~ message and your application receives events again. While the device is asleep, foreground and background applications do continue to run, but should do as little work as possible in order to preserve battery life.

In both cases, you can use the ~~applicationDidBecomeActive:~~ method to restore the application to the state it was in before the interruption. What you do depends on your application. In some applications, it makes sense to resume normal processing. In others @@md if you've paused a game, for example @@md you could leave the game paused until the user decides to resume play.

Your application moves to the background

When the user accepts the notification or interruption, or presses the Home button @@md or the system launches another application @@md your application then moves into the background state, where it is suspended to conserve power. (If an app needs to continue running, it can request execution time from the system.)

When your app enters the background state, it will be sent the ~~applicationDidEnterBackground:~~ message. In this method, you should save any unsaved data or *state* (where the user is in the app @@md the current view, options selected, and stuff like that) to a temporary cache file or to the preferences database "on disk." (Okay, I know, Apple calls the iPad storage system a *disk* even though it is a solid-state drive, but if Apple calls it that, I probably should, too, just so I don't confuse too many people.) Although I don't do this in DeepThoughts, your app's delegate can implement the delegate method ~~applicationWillTerminate:~~ to save the current state and unsaved data.

The next time the user launches your app, your code can use that information to restore your app to its previous state. You need to do everything necessary to restore your application in case it's subsequently purged from memory. You also have to do addition cleanup operations, such as deleting temporary files. Even though your application enters the background state, there's no guarantee that it will remain there indefinitely. If

memory becomes constrained, iOS will purge background apps to make more room for the foreground app.

<Warning>

If your application is suspended when your application is purged, *it receives no notice that it is removed from memory.* You need to save any data beforehand! The state information you save should be as minimal as possible but still let you accurately restore your app to an appropriate point. You don't have to display the exact same screen used previously @@md for example, if a user edits a contact and then leaves the Contacts app, upon returning, the Contacts app displays the top-level list of contacts, rather than the editing screen for the contact.

If your application requests more execution time or it has declared that it does background execution, it's allowed to continue running after the ~~applicationDidEnterBackground:~~ method returns. If not, your (now) background application is moved to the *suspended* state shortly after returning from the ~~applicationDidEnterBackground:~~ method.

When your delegate is sent the ~~applicationDidEnterBackground:~~ method, your app has approximately five seconds to finish things up. If the method doesn't return before time runs out (or if your app doesn't request more execution time from iOS), your app is terminated and purged from memory.

Your application resumes processing

At some point, it's likely that the user will once again want to use your app, which has been patiently sitting in background waiting for this opportunity to respond to the user's tap. When the user returns to your app by tapping it on a Home screen or in the switching pane below the dock, your application delegate is sent the ~~applicationWillEnterForeground:~~ and ~~applicationDidBecomeActive:~~ messages.

I've already explained in the previous section what you'll need to do in the ~~applicationDidBecomeActive:~~ method @@md restart anything it stopped doing and get ready to handle events again.

While an application is suspended, the world still moves on, and iOS tracks all of the things the user is doing that may impact your application. For example, the user may change the device orientation from landscape to portrait, and then to landscape, and then finally back to portrait. While iOS will keep track of all these events, it will send you only a single event that reflects the final change you'll need to process @@md in this case, the change to portrait.

Termination

Apps are generally moved to the background when interrupted or when the user quits. But if the app was compiled with an earlier version of the SDK or is running on an earlier version of the operating system that doesn't support multitasking @@md or you decide you don't want your app to run in the background and you set the ~~UIApplicationExitsOnSuspend~~ key in its ~~Info.plist~~ file @@md the Terminator stomps on your app.

What happens is this: Your app delegate is sent the ~~applicationWillTerminate:~~ message, and you have the opportunity to add code to do whatever you want to do before termination, including saving the state the user was in. Saving is important, because then, when the app is launched again (refer to Step 2 in "App Anatomy 101 @@md The Lifecycle," earlier in this chapter) and the ~~UIApplicationMain~~ sends the app delegate the ~~applicationDidFinishLaunching~~ message, you can restore the app to the state the

user left it (such as a certain view). This is the same thing you would have done in the application method ~~applicationDidEnterBackground:~~ @@md I cover this in "Responding to interruptions" in this chapter.

<Tip>

It's worth noting that before the application is terminated and the ~~applicationWillTerminate:~~ message is sent, the ~~applicationDidEnterBackground:~~ message is also sent. This means that for applications you compile and run only under iOS 4.2 and beyond, you only have to do all that cleanup and file saving in ~~applicationDidEnterBackground:~~.

Your ~~applicationWillTerminate:~~ method implementation has about five seconds to do what it needs to do and return. Any longer than that and your application is terminated and purged from memory. (The Terminator doesn't kid around.)

<Remember>

Even if you develop your application using the iOS 4.2 SDK and newer versions @@md as you will be doing, if you stay as up-to-date as me @@md you must still be prepared for your application to be terminated. If memory becomes an issue (and it inevitably will if there are enough apps in the background), the system might remove your application from memory in order to make more room. If it does remove your *suspended* application, *it does not give you any warning, much less notice!* However, if your application is currently running in the background, the system does call the ~~applicationWillTerminate:~~ method of the application delegate.

The Managed Memory Model Design Pattern

Launch, initialize, process, respond, terminate, launch, initialize, process, respond, terminate . . . it has a nice rhythm to it, doesn't it? Those are the five major stages of the application's lifecycle. But life isn't simple @@md and neither is runtime. To mix things up a bit, your application will also have to come to terms with memory management.

You may remember that I mention in Chapter 7 that there's one other design pattern: the Managed Memory Model. One of the main responsibilities of all good little applications is to deal with low memory. So, the first line of defense is (obviously) to understand how you as a programmer can help them *avoid* getting into that state.

In iOS, each program uses the virtual-memory mechanism found in all modern operating systems. But virtual memory is limited to the amount of physical memory available. This is because iOS doesn't store changeable memory (such as object data) on the "disk" to free up space and then read it in later when it's needed. Instead, iOS tries to give the running application the memory it needs, freeing memory pages that contain read-only contents (such as code), where all it has to do is load the "originals" back into memory when they're needed. Of course, this may be only a temporary fix if those resources are needed again a short time later.

If memory continues to be limited, the system may also send notifications to the running application, asking it to free up additional memory. This is one of the critical events that all applications must respond to.

Observing low-memory warnings

When the system dispatches a low-memory notification to your application, it's something you must pay attention to. If you don't, it's a reliable recipe for disaster. (Think of your low-fuel light going on as you approach a sign that says, "Next services 100 miles.") ~~UIKit~~ provides several ways of setting up your application so that you receive timely low-memory notifications:

* Implement the ~~applicationDidReceiveMemoryWarning:~~ method of your application delegate. Your application delegate could then release any data structure or objects it owns @@md or notify the objects to release memory they own. Apple recommends this approach.
* Override the ~~didReceiveMemoryWarning:~~ method in your custom ~~UIViewController~~ subclass. The view controller could then release views @@md or even other view controllers @@md that are off-screen. For example, in your new project (DeepThoughts) created with the View-based Application template, the template already supplies the following in ~~DeepThoughtsViewController.m~~, ready for you to customize:

```
- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview.
     [super didReceiveMemoryWarning];
// Release any cached data, images, etc that aren't in use.
}
- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}
```

* Register to receive the ~~UIApplicationDidReceiveMemoryWarningNotification:~~ notification. Such notifications are sent to the notification center, where all notifications are centralized. An object that wants to get informed about this notification registers itself to the notification center by telling which notification it wants to get informed and which method should be invoked when the notification is raised. A model object could then release data structures or objects it owns that it doesn't need immediately and can re-create later. However, this approach is beyond the scope of this book.

Each of these strategies gives a different part of your application a chance to free up the memory it no longer needs (or doesn't need right now). As for how you actually get these strategies working for you, that's dependent on your application's architecture. That means you need to explore it on your own.

Not freeing up enough memory will result in iOS sending your iPad application the ~~applicationWillTerminate:~~ message and shutting you down. For many apps, though, the best defense is a good offense, and you need to manage your memory effectively and eliminate any memory leaks in your code. A memory leak is how programmers describe a situation in which an object is unable to release the memory it has acquired @@md it can diminish performance by reducing the amount of available memory.

Avoiding the warnings

When you create an object @@md a window or button for example @@md memory is allocated to hold that object's data. The more objects you create, the more memory you use and the less there is available for additional objects you might need. Obviously, it's important to make available (that is, *de-allocate*) the memory that an object was using when the object is no longer needed. This task is called *memory management.*

Objective-C uses reference counting to figure out when to release the memory allocated to an object. It's your responsibility as a programmer to keep the memory-management system informed when an object is no longer needed.

<Remember>

Reference counting is a pretty simple concept. When you create the object, it's given a reference count of 1. As other objects use this object, they use methods to increase the reference count, and to decrease it when they're done. When the reference count reaches 0, the object is no longer needed, and the memory is de-allocated.

Some basic memory-management rules you shouldn't forget

Here are the fundamental rules when it comes to memory management:

* Any object you create using ~~alloc~~ or ~~new~~, any method that contains ~~copy~~, and any object you send a ~~retain~~ message to is *yours* @@md you own it. That means you're responsible for telling the memory-management system when you no longer need the object and that its memory can now be used elsewhere.
* Within a given block of code, the number of times you use ~~new~~, ~~copy~~, ~~alloc~~, and ~~retain~~ should equal the number of times you use ~~release~~ and ~~autorelease~~. You should think of memory management as consisting of pairs of messages. If you balance every ~~alloc~~ and every ~~retain~~ with a ~~release~~, your object will eventually be freed up when you're done with it.
* When you assign an instance variable using an accessor with a property attribute of ~~retain~~, ~~retain~~ is automatically invoked @@md that is, you now own the object. Implement a ~~dealloc~~ method to release the instance variables you own.
* Objects created any other way (through convenience constructors or other accessor methods) are not your problem.

If you have a solid background in Objective-C memory management (all three of you out there), following those rules should be straightforward or even obvious. If you don't have that background, no sweat: See *Objective-C For Dummies* for some background.

<Remember>

A direct correlation exists between the amount of free memory available and your application's performance. If the memory available to your application dwindles far enough, the system will be forced to terminate your application. To avoid such a fate, keep a few words of wisdom in mind:

* Minimize the amount of memory you use @@md make that a high priority of your implementation design.
* Be sure to use the memory-management functions.
* In other words, be sure to clean up after yourself, or the system will do it for you, and it won't be a pretty picture.

Whew!

Congratulations @@md in the last chapter and this chapter, you've just gone through the "Classic Comics" version of another several hundred pages of Apple documentation, reference manuals, and how-to guides.

The details in the previous chapter and this chapter provide a frame of reference on which you can hang the concepts I throw around with abandon in upcoming chapters

@@md as well as the groundwork for a deep enough understanding of the application lifecycle to give you a handle on the detailed documentation. Now it's time to move on to the really fun stuff: building DeepThoughts into an app that actually does something.