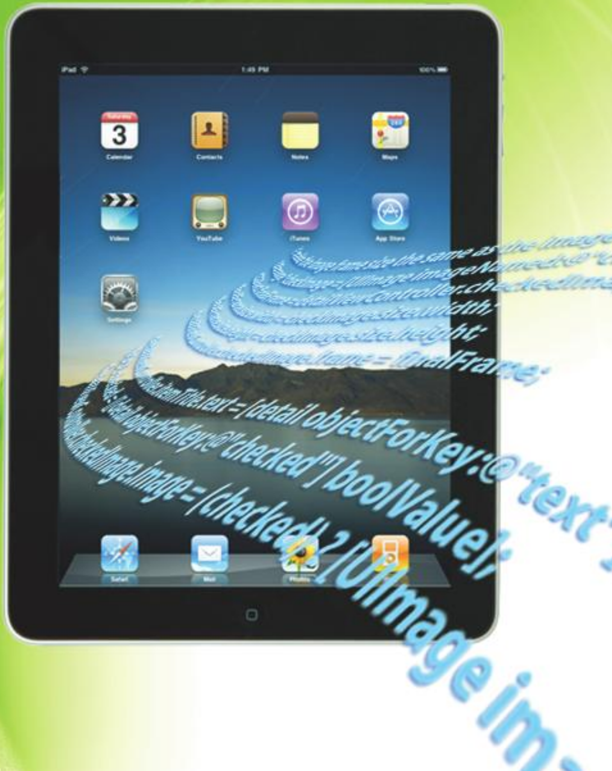
Making Everything Easier!™

2nd Edition

iPad™ Application Development For Dummies®

Learn to:

- Download the SDK and start using Apple's developer tools
- Incorporate the latest iPad and iOS features into your app designs
- Take advantage of iPad's full functionality to create a good user experience
- Print from your application using AirPrint



Neal Goldstein
Tony Bove

Authors of iPhone Application Development All-In-One For Dummies

iPad™ Application Development For Dummies®

2nd Edition

iPad™ Application Development For Dummies®

2nd Edition

by Neal Goldstein and Tony Bove

WILEY
Wiley Publishing, Inc.

iPad™ Application Development For Dummies®, 2nd Edition

Published by
Wiley Publishing, Inc.
111 River Street
Hoboken, NJ 07030-5774
`www.wiley.com`

Library of Congress Control Number: 2010943059

ISBN: 978-0-470-92050-3

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

WILEY

About the Authors

Neal Goldstein is a recognized leader in making state-of-the-art and cutting-edge technologies practical for commercial and enterprise development. He was one of the first technologists to work with commercial developers at firms such as Apple Computer, Lucasfilm, and Microsoft to develop commercial applications using object-based programming technologies. He was a pioneer in moving that approach into the corporate world for developers at Liberty Mutual Insurance, USWest (now Verizon), National Car Rental, EDS, and Continental Airlines, showing them how object-oriented programming could solve enterprise-wide problems. His book (with Jeff Alger) on object-oriented development, *Developing Object-Oriented Software for the Macintosh* (Addison Wesley, 1992), introduced the idea of scenarios and patterns to developers. He was an early advocate of the Microsoft .NET framework, and he successfully introduced it into many enterprises, including Charles Schwab. He was one of the earliest developers of Service Oriented Architecture (SOA), and as Senior Vice President of Advanced Technology and the Chief Architect at Charles Schwab, he built an integrated SOA solution that spanned the enterprise, from desktop PCs to servers to complex network mainframes. (He holds three patents as a result.) As one of IBM's largest customers, he introduced the folks at IBM to SOA at the enterprise level and encouraged them to head in that direction.

He is passionate about the real value mobile devices can provide and has eight applications in the App Store. These include a series of Travel Photo Guides (`http://travelphotoguides.com`) developed with his partners at mobile-fortytwo and a Digital Field Guides series developed in partnership with John Wiley & Sons (`http://lp.wileypub.com/DestinationDFGiPhoneApp`).

Along with those apps, he has written several books on iPhone programming, including all three editions of *iPhone Application Development For Dummies* (Wiley) and *Objective-C For Dummies* (Wiley).

Because you can never tell what he'll be up to next, check regularly at his Web site, `www.nealgoldstein.com`.

Tony Bove is crazy about the iPad, iPod, and iPhone, and he not only provides free tips on his Web site (`www.tonybove.com`), but also developed an iPhone application (*Tony's Tips for iPhone Users*) and is working on several iPad apps. Tony has written more than two dozen books on computing, desktop publishing, and multimedia, including his own iPod & iTunes For Dummies, iPod touch For Dummies, and iLife For Dummies, as well as iPhone Application Development All-in-One For Dummies with Neal; he also wrote Just Say No to Microsoft (No Starch Press) in 2005; The Art of Desktop Publishing (Bantam) in 1986; and a series of books about Macromedia Director, Adobe Illustrator, and PageMaker from 1986–1997. Tony produced a CD-ROM interactive documentary in 1996, Haight-Ashbury in the Sixties, and developed the Rockument music site, `www.rockument.com`, with commentary and podcasts focused on rock music history. Tony has also worked as a director of marketing for leading-edge software companies and as a marketing messaging consultant.

Dedication

Neal Goldstein: To my children, Sarah and Evan, and all of my personal and artist friends who have kept me centered on the (real) world outside of writing and technology. But most of all to my wife Linda who is everything that I ever hoped for and more than I deserve. Yes Sam . . . the light at the end of the tunnel is not a freight train.

Tony Bove: Tony dedicates this book to his mother, his brothers, and his sons, nieces, nephews, their cousins, and all their children . . . the iPad generation.

Authors' Acknowledgments

Neal Goldstein: Thanks to my business partners Jeff Enderwick and Jeff Elias in mobilefortytwo and for their support and picking up the slack while I was engaged in finishing this book. Maggie Canon for putting Tony and I together. Carole Jelen, for her continued work and support in putting this project together.

Acquisitions Editor Kyle Looper for keeping us on track and doing whatever he needed to do to allow us to stay focused on the writing. The Project Editor's Project Editor Paul Levesque who has been known to do even more than six impossible things before breakfast. Copy Editor Virginia Sanders did another great job in helping us make things clearer. Technical reviewer Jesse Fuller added a great second pair of eyes.

Tony Bove: Tony owes thanks and a happy hour or two to Carole Jelen at Waterside for agenting, to Maggie Canon for putting the authors together, and to Kathy Pennington for support.

Publisher's Acknowledgments

We're proud of this book; please send us your comments at `http://dummies.custhelp.com`. For other comments, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002.

Some of the people who helped bring this book to market include the following:

Acquisitions, Editorial, and Media Development

Senior Project Editor: Paul Levesque

Acquisitions Editor: Kyle Looper

Copy Editor: Virginia Sanders

Technical Editor: Jesse Feiler

Editorial Manager: Leah Cameron

Media Development Project Manager: Laura Moss-Hollister

Media Development Assistant Project Manager: Jenny Swisher

Media Development Associate Producers: Josh Frank, Marilyn Hummel, Douglas Kuhn, and Shawn Patrick

Editorial Assistant: Amanda Graham

Sr. Editorial Assistant: Cherie Case

Cartoons: Rich Tennant (`www.the5thwave.com`)

Composition Services

Project Coordinator: Patrick Redmond

Layout and Graphics: Carl Byers, Lavonne Roberts

Proofreaders: Laura Bowman, Lindsay Littrell

Indexer: BIM Indexing & Proofreading Services

Publishing and Editorial for Technology Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Acquisitions Director

Mary C. Corder, Editorial Director

Publishing for Consumer Dummies

Diane Graves Steele, Vice President and Publisher

Composition Services

Debbie Stailey, Director of Composition Services

Contents at a Glance

Table of Contents

Introduction

The world stood on its toes as Steve Jobs announced the iPad in January 2010 as "our most advanced technology in a magical and revolutionary device at an unbelievable price."

Do you believe in magic? The iPad has that magical quality of disappearing into your hands as you explore content with it. You have to hold one and use it to understand that feeling of the hardware disappearing — you have the software application itself in your hands, with no extraneous buttons and controls in the way of your experience with the content. And yes, the iPad is groovy — it's based on the iPod and iPhone.

But the iPad is more than groovy: It's a game changer for the Internet as a publishing medium, for the software industry with regard to applications, and for the mobile device industry with regard to the overall digital media experience. The form factor, portability, swift performance, and software experience change the game with all devices that access the Internet. And we're tickled pink to be writing about developing software for it at this early stage of its evolution, because we know the iPad will in fact revolutionize portable computing and Internet access.

Due to the success of the iPhone and iPod touch, the App Store has grown to become the repository of over 300,000 applications as of this writing, which collectively are driving innovation beyond the reach of other mobile devices — and all these apps already run on the iPad, along with about 30,000 iPad-specific apps. Opportunities are wide open for inventions that build on all the strengths of iPhone apps but that take advantage of the iPad's larger display.

As we continue to explore the iPad as a new platform, we keep finding more possibilities for applications that never existed before. The iPad is truly a mobile computer with a decent display. Its hardware and software make it possible to wander the world, or your own neighborhood, and stay connected to whomever and whatever you want to. It gives rise to a new class of here-and-now applications that enable you to access content-rich services and view information about what is going on around you, and to interact with those services or with others on the Internet.

One of the hallmarks of a great iPad application is that it leverages the iPad's unique hardware and operating system (iOS). The Software Development

Kit (SDK), which you use to develop iPad applications, helps you develop apps for iOS 4.2, which offers many new features, including multitasking and Apple's iAds program for displaying ads within apps. The SDK also includes tools such as MapKit, which makes it much easier to use the location-based features of the iPad in an application. MapKit makes it possible for even a beginning developer to take full advantage of knowing the location of the device, and we've included the code for an example app (called iPadTravel411) to show you how. And the frameworks supplied in the SDK are especially rich and mature. All you really have to do is add your application's user interface and functionality to the framework, and then "poof" . . . you have an instant application.

If this seems too good to be true, well, okay, it *is,* sort of. What's really hard, after you've learned the language and framework, is creating a structure for the iPad application's data and building models for the logic of how the application should work. Although there are lots of resources, the problem is exactly that: There are *lots* of resources — as in *thousands* of pages of documentation! You may get through a small fraction of the documentation before you just can't take it anymore and plunge right into coding. Naturally enough, there will be a few false starts and blind alleys until you find your way, but we predict that after reading this book, it will be (pretty much) smooth sailing.

Editor's note: Both authors (Tony and Neal) have previously published applications for the iPhone — you can find several of Neal's apps, including ReturnMeTo*, in the App Store, along with Tony's app,* Tony's Tips for iPhone Users.

About This Book

iPad Application Development For Dummies is a beginner's guide to developing applications for the iPad, which runs iOS. And not only do you *not* need any iPad (or iPhone) development experience to get started, you don't need any Macintosh development experience either. We expect you to come as a blank slate, ready to be filled with useful information and new ways to do things.

Because of the nature of the iPad, you can create content-rich, truly immersive applications that can be really powerful (as well as amazing to look at). And because you can also start small and create real applications that do something important for a user, it's relatively easy to transform yourself from "I know nothing" into a developer who, though not (yet) a superstar, can still crank out quite a respectable application.

The iPad can be home to some pretty fancy software as well — so we'll take you on a journey through building not just a simple app but also an industrial-strength app, so that you know the ropes for developing your own app.

This book distills the hundreds (or even thousands) of pages of Apple documentation, not to mention our own development experiences, into only what's necessary to start you developing real applications. But this is no recipe book that leaves it up to you to put it all together; rather, this book takes you through the frameworks and iPad architecture in a way that gives you a solid foundation in how applications really work on the iPad — and acts as a roadmap to expand your knowledge as needed.

It's a multiple-course banquet, intended to make you feel satisfied (and really full) at the end.

Conventions Used in This Book

This book guides you through the process of building iPad applications. Throughout, you use the provided iOS framework classes for the iPad (and create new ones, of course) and code them using the Objective-C programming language.

Code examples in this book appear in a monospaced font so they stand out a bit better. That means the code you'll see will look like this:

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

Objective-C is based on C, which (we want to remind you) *is* case-sensitive, so please enter the code that appears in this book *exactly* as it appears in the text. This book also uses the standard Objective-C naming conventions — for example, class names always start with a capital letter, and the names of methods and instance variables always start with a lowercase letter.

All URLs in this book appear in a monospaced font as well:

```
www.nealgoldstein.com
www.tonybove.com
```

You'll notice — starting around Chapter 14 — that I'll be asking you to delete some of the code you have in place for your project in order to make room for some new stuff. When that happens, I'll be referring to code I want you delete as BUI (*b*old, *u*nderlined, *i*talic) code, because said code will show up as bold, underlined and italic. Simple enough.

If you're ever uncertain about anything in the code, you can always look at the source code on Neal's Web site at `www.nealgoldstein.com`. From time to time, he provides updates for the code there and posts other things you might find useful. Tony offers tips about everything from developing apps and marketing them to using the iPad, iPod, iPhone, and iTunes at `www.tonybove.com`.

Foolish Assumptions

To begin programming your iPad applications, you need an Intel-based Macintosh computer with the latest version of the Mac OS on it. (No, you can't program iPad applications on the iPad.) You also need to download the Software Development Kit (SDK) — which is free — but you have to become a registered iOS developer before you can do that. (Don't worry; we show you how in Chapter 4.) And, oh yeah, you need an iPad. You won't start running your application on it right away — you'll use the Simulator that Apple provides with the SDK during the initial stages of development — but at some point, you'll want to test your application on a real, live iPad.

This book assumes that you have some programming knowledge and that you have at least a passing acquaintance with object-oriented programming, using some variant of the C language (such as C++, C#, or maybe even Objective-C). If not, we point out some resources that can help you get up to speed. The examples in this book are focused on the frameworks that come with the SDK; the code is pretty simple (usually) and straightforward. (We don't use this book as a platform to dazzle you with fancy coding techniques.)

This book also assumes that you're familiar with the iPad itself and that you've at least explored Apple's included applications to get a good working sense of the iPad's look and feel. It would also help if you browse the App Store to see the kinds of applications available there, and maybe even download a few free ones (as if we could stop you).

How This Book Is Organized

iPad Application Development For Dummies has five main parts.

Part I: Planning the Killer App

Part I introduces you to the iPad world. You find out what makes a great iPad application and how to exploit the iPad's best features to create a compelling user experience. You also discover the marketing secrets for getting the most out of Apple's App Store and distributing your app to more customers.

Part II: Becoming a Real Developer

In this part you learn how to become an "official" developer and what you need to do to in order to be able to distribute your iPad applications through Apple's App Store. You go through the process of registering as a developer and downloading the Software Development Kit (SDK) — and then you unpack all the goodies contained therein, including Xcode (the Apple development environment) and Interface Builder. Chapter 6 spells out the details of obtaining the proper certificates and submitting your app to the App Store — and the dire consequences of not following the rules.

Part III: Understanding How Apps Work

Part III is deceptively short but intensely illuminating. The two chapters in this part explain the frameworks that form the raw material of your iPad app (which you then refine with your code and user interface objects) and reveal the design patterns that you should adopt to make use of these frameworks. This part also describes in detail the lifecycle of an iPad app from launch to termination. When you finish this part, you should have enough information to get started coding your application.

Part IV: Building DeepThoughts

With the basics behind you and a good understanding of the application architecture under your belt, it's finally time to have some fun doing something useful. In this part, we show you how to create an application that's simple enough to understand and yet demonstrates enough of the building blocks for creating a sophisticated app. We show you how an app fits into the frameworks that do all of the heavy lifting for the iPad's user interface. And because you design the app the right way from the start, you can plug in user interface elements with minimal effort using Interface Builder (part of the SDK). No sweat, no bother. Putting this handy little app together gives you some practice at creating a useful iPad program that presents a view of content, responds to simple gestures, and lets users change preference settings. It's a great application to learn about iPad development — it has enough features to be useful as an example, but it's simple enough not to make your head explode.

Part V: Building an Industrial-Strength Application

Part V shows you how to create an application that contains major functionality — we take an idea that was developed for the iPhone and expand it to take advantage of the iPad's capabilities. The app (iPadTravel411) makes it easier to travel by reducing all those hassles of getting to and from a strange airport, getting around a city, getting the best exchange rate, and knowing how much you should tip in a restaurant — that sort of thing. We don't go slogging through every detail, but we demonstrate almost all the technology you need to master if you're going to create a compelling iPad application on your own.

Part VI: The Part of Tens

Part VI consists of some tips to help you avoid having to learn everything the hard way. It talks about approaching application development in an "adult" way right from the beginning (without taking the fun out of it, I assure you).

Icons Used in This Book

TIP

This icon indicates a useful pointer that you shouldn't skip.

REMEMBER

This icon represents a friendly reminder. It describes a vital point that you should keep in mind while proceeding through a particular section of the chapter.

TECHNICAL STUFF

This icon signifies that the accompanying explanation may be informative (dare we say, interesting?), but it isn't essential to understanding iPad application development. Feel free to skip past these tidbits if you'd like (though skipping while leaning may be tricky).

WARNING!

This icon alerts you to potential problems that you may encounter along the way. Read and obey these blurbs to avoid trouble.

Where to Go from Here

It's time to explore the iPad! If you're nervous, take heart: The iPad is so new, and such rich territory for developers to mine, that no company or individual has a lock on innovating with it. Your idea just might be the killer app everyone's waiting for.

So go have some fun!

Part I
Planning the Killer App

The 5th Wave   By Rich Tennant

"What I'm doing should clear your sinuses, take away your headache, and charge your iPad."

In this part . . .

Apple CEO Steve Jobs announced the iPad as "our most advanced technology in a magical and revolutionary device at an unbelievable price." Do you believe in iPad magic? I certainly do, if the music is groovy — and with the iPad, the entire experience is groovy.

You say you want a revolution? Well, here's the plan: This part lays out what you need to know to get started on the Great iPad Development Trek. After reading this part, you can evaluate your idea for an iPad application, see how it ranks, and maybe figure out what you have to do to transform it into something that knocks your users' socks off.

- Chapter 1 describes the features of the iPad and the elements that make a great iPad application. You find out how to exploit the platform's features and embrace its limitations. You also discover how to design with Apple's expectations in mind.

- Chapter 2 goes into more detail about how to create a compelling user experience with your iPad app. You find out how to design for the iPad and its entirely new set of user interaction features.

- Chapter 3 explains what motivates your potential customers to download apps, how to reach these customers and learn from them, what marketing methods you can use to drum up interest, and how to determine the right price for your app.

Chapter 1

What Makes a Killer iPad App

In This Chapter

- Figuring out what makes an insanely great iPad application
- Discovering the features of the iPad that can inspire you
- Understanding Apple's expectations for iPad applications
- Making a plan for developing iPad software

Douglas Adams, in the bestseller *The Hitchhiker's Guide to the Galaxy* (conceived in 1971 and published in 1979), introduced the idea of a handy travel guide that looked "rather like a largish electronic calculator," with a hundred tiny flat press buttons and a screen "on which any one of a million 'pages' could be summoned at a moment's notice." It looked insanely complicated, and this is one of the reasons why the snug plastic cover it fitted into had the words DON'T PANIC printed on it in large friendly letters. According to Adams, this guide was published in this form because "if it were printed in normal book form, an interstellar hitchhiker would require several inconveniently large buildings to carry it around in."

The iPad is a hitchhiker's dream come true, and its users don't even have any reason to panic. The only "insanely complicated" part of the iPad experience may be trying to develop a killer app that best exemplifies the iPad's features, but that's why I think this book should have DON'T PANIC printed on its cover — it takes you through the entire process of imagining, creating, developing, testing, and distributing your iPad app. And in this chapter, I talk about what would make that app a killer app.

As you already know, the iPad is a *tablet* — a new category of mobile device located somewhere between a Mac laptop and an iPod touch or iPhone in terms of its capabilities — that evolved from the iPhone design and uses iOS, the iPhone/iPad Operating System.

The iPad already runs the 300,000+ iPhone apps in the Apple App Store with either pixel-for-pixel accuracy in a black box in the center of the display, or scaled up to full screen (which is done on the fly by doubling the pixels). The App Store is loaded with travel and digital media apps, so you know already that the iPad as a "Hitchhiker's Guide" is not a fantasy. You may think it a fantasy that you could develop an iPad app in less than two months, starting from where you are now, with no iPad programming experience. But you *can* — the only question is whether you can make a *great* app, or even a *killer* app. To do that, you need to look at what it takes for an iPad app to be truly great.

Figuring Out What Makes a Great iPad Application

You use the same software development kit and much of the same code to develop iPad, iPhone, and iPod touch applications. The iPad runs the same operating system as the iPhone and iPod touch. However, the iPad is a bigger device with more horsepower and a larger display, as shown in Figure 1-1.

For many iPhone/iPod touch app developers, the iPad's larger display alone changes everything. Apple demonstrated exactly how *far* things have changed when the company demonstrated the iWork suite of productivity tools (Keynote for presentations, Numbers for spreadsheets, and Pages for word processing and page formatting) on the iPad, which would be unthinkable for today's iPhone or iPod touch.

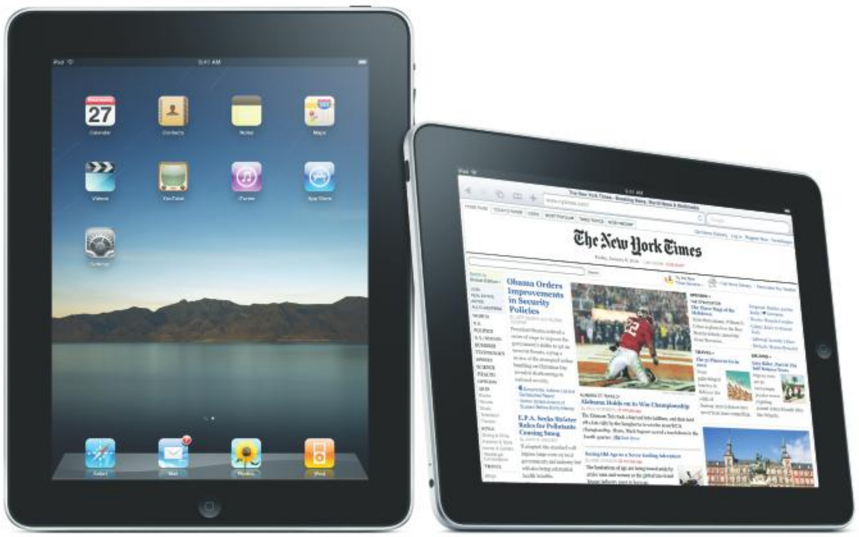


Figure 1-1: The iPad runs iOS (left) and offers a larger display to show content such as a newspaper (right).

The biggest challenge in making a killer app for the iPad is to design for the iPad *experience,* and one reason why the iPad offers such a better experience than any Windows netbook or tablet computer is its sex appeal (which for many apps can mean *more excellent content* and *finer style*). For example, according to Douglas Adams, the Encyclopedia Galactica describes *alcohol* as “a colorless volatile liquid formed by the fermentation of sugars” and also notes “its intoxicating effect on certain carbon-based life forms.” On the other hand, The Hitchhiker’s Guide to the Galaxy not only tells you what *alcohol* is, it says “the best drink in existence is the Pan Galactic Gargle Blaster,” describes its effect as “like having your brains smashed out by a slice of lemon wrapped round a large gold brick,” tells you which planets have bars that offer it and at what prices, and then shows you how to mix one yourself. As Adams points out, “The Hitchhiker’s Guide to the Galaxy *sells rather better* than the Encyclopedia Galactica.”

If the explosion of new iPhone apps since its introduction is any indication, you will want to take advantage of the iPad’s sexiness, and that means leveraging its fabulous touch-sensitive interface and other features. Because the iPad evolved from the iPhone design, the iPad has design advantages that make netbooks and laptops feel like the dull Encyclopedia Galactica. Most iPhone apps are designed to take advantage of the iPhone’s Multi-Touch display; accelerometer (which detects acceleration, rotation, motion gestures, and tilt); or location services for detecting its physical location — or all three.

However, you can create iPad apps that are not just a little bit better than their iPhone counterparts, but a *lot* better (and an order of magnitude more powerful), with an interface that’s simpler to use than a Mac.

Providing an immersive experience

An iPad app can offer a more immersive experience compared with an iPhone app by adding *more content* — full pages from the Internet or in memory, maps you can zoom into, full-screen videos and slide shows with music, and so on. People can enjoy this content while away from their desks — on living room couches, in coffee shops, on the train, in outer space — and more easily share it with others than they can by using an iPhone or iPod touch.

Whenever possible, add a realistic, physical dimension to your application. When I demonstrate the iPad to someone, my favorite game to play is Touch Hockey, because it fully immerses you in an experience that resembles the physical game so well. But it’s also a good idea to extend some physical metaphors, like the newspaper or book page, to provide a more immersive experience. *The New York Times,* for example, designed an iPad app that looks like a newspaper and also includes embedded, fully functional videos (not just videos that appear in a separate window). In the iBooks app, you swipe the page to go to the next one, just like a real book, but you can also search the entire text, add bookmarks, and change the font size.

Need for Speed Shift for the iPad from Electronic Arts feels like you're driving the display with your hands as you steer the car using the iPad like a steering wheel. The high-definition screen is just inches from your face — the field of view and the sensation of speed you get are incredible. The full-screen display is also fully touch sensitive — you can tap on a car and see inside it, flick a lifelike gear shifter to shift gears, and tap the rear-view mirror to look behind you.

Even utility apps can be rethought to be a better experience. On the iPhone, the Contacts app is a streamlined list, but on the iPad, Contacts is an address book with a beautifully tangible look and feel. The more true to life your application looks and behaves, the easier it is for people to understand how it works and enjoy using it.

Making content relevant

The iPad's large display may tempt you to consider a design for your app that would look good on a laptop. But you should not forget the first rule of iPhone design: Make its content and functions *relevant to the moment.* The iPad, like the iPhone, adds mobility to the party. This ability to run apps wherever you are and whenever you want makes it possible to have the information you need (as well as the tools you'd like to use) constantly available. But it's not just about the fact that the app you need is ready to run right there on your iPad; it's (as importantly) about how the app is designed and implemented. It needs to require as little as possible from the user in terms of effort when it comes to delivering results.

An iPad app can present information relevant to where you are, what time it is, what your next activity might be, and how you're holding the device (in portrait or landscape view, tilting and shaking it, and so on), just like an iPhone or iPod touch app.

For example, the version of Google Maps for the iPad displays a full-screen map that can show your location and immediately find commercial establishments nearby. (For example, you can search for *"sushi"* to find sushi restaurants.)

The iPad platform offers a strong foundation for pinpointing the device's current location on a map, controlling views, managing data, playing multimedia content, switching display orientations, and processing gestures. Because the iPad platform can do all that, an app can know your current location, the hotels or campgrounds you're going to stay at, and the planets you're planning to visit. It can even show videos and play the music of the stars all at the same time. While searching maps and brochures, you can know at a glance where you are, how to get to your destination, and what the weather's like so that you know what to wear.

Designing for the touch-display experience

The important design decision to make, whether you're starting from scratch with a new iPad app or evolving one from an iPhone app, is whether to use the large iPad screen and the new user interface elements to give people access to more information in one place. Although you don't want to pack too much information into one screen, you do want to prevent people from feeling that they must visit many different screens to find what they want. An iPad app can offer the primary content on the Main view and provide additional information or tools in an auxiliary view (such as a *popover* that appears above the Main view) to give users access to functions without requiring them to leave the context of the Main view.

The large iPad screen also gives you a lot more room for multifinger gestures, including gestures made by more than one person. An iPad app can react to gestures and offer touch controls and pop-up settings that are relevant to what you're actually doing in the app and where you place your fingers. With a display the size of a netbook, you have a lot more screen real estate to allow dragging and two-finger gestures with graphics and images, and depending on what you're doing, a tap or gesture on a particular part of the screen can have a particular function. For example, in the Gameloft version of the first-person shooter called *Nova* (as adapted to the iPad), the display size gives you more flexibility than the iPhone version, with more controls and objects such as mini-maps, and you can slide two fingers across the screen to throw grenades.

With all this in mind, there are at least two things that you need to consider — besides functionality, of course — when it comes to creating a great iPad app:

- Exploiting the platform and ecosystem
- Creating a compelling user experience

The rest of this chapter and Chapter 2 dig more into this Two-Part Rule of Great iPad Applications.

Exploiting the Platform

Okay, enough talk about the iPad's unique experience. Just what exactly is the iPad platform, and what are its features?

The iPad runs iOS version 4.2 as its operating system, and iPad apps use many of the same views and controls you used if you already developed an iPhone app. But the design similarities end there. The iPad's *hardware* is ground zero for conceiving the design of an iPad app — it's the place to start dreaming of what kind of experience to provide:

- A touch-sensitive display size of 1,024 x 768 pixels that supports multifinger gestures.
- The connection features of the iPhone (except phone calls): Wi-Fi and optional 3G Internet access; a compass; location services (although a hardware GPS isn't included in the first version of the iPad, so it isn't as accurate); and the ability to play audio and video with ease.
- Flexible orientation — users can tilt it, rotate it, and turn it upside down.
- The capability to plug in an external keyboard (or pair a Bluetooth wireless keyboard with the iPad) and use it in place of the onscreen keyboard for extended typing.
- The ability for users to dock the iPad and share files with a computer or other iPad users.

Exploiting advantages of the system

One of the keys to creating a great app is taking advantage of what the device offers. In the case of a new platform with new possibilities, such as the iPad, exploiting advantages is especially important. The combination of hardware and system software opens up design advantages that depart from the typical design approach for desktop and laptop applications.

For example:

- **Multifinger gestures:** Applications respond to multifinger gestures, not mouse clicks. If you design an app that simply uses a single finger tap as if it were a mouse click, you may be missing an opportunity to design a better user experience.
- **Movement and orientation:** The iPad includes an accelerometer just like the one in an iPhone and iPod touch, so you can also design apps that detect accelerated movement, as well as change the display for different orientations.

- **Split views and unique keyboards:** You can use a Split view to display more than one view onscreen at a time. You can also bring up a special keyboard unique to the task, such as the numbers-and-formulas keyboard that appears in the Numbers app for the iPad.
- **Internet access:** As with an iPhone or iPod touch, users can send and receive e-mail and browse the Web; sync contacts, calendars, and notes over the Internet; and download content from Apple stores. With quick and easy access, your app doesn't need to store lots of data on the iPad — all it really needs to do is jump on the Internet and grab what it needs from there.
- **Computer sync over USB connection or local area network:** Users can sync their photos, contacts, calendars, music, video, and other content from their computers (again, just like an iPhone or iPod touch), and with some apps (such as Bento from FileMaker), users can sync data over a local area network.
- **Television or projection system connection:** Users can connect the iPad to an HDTV or projection system in order to show content to larger audiences.
- **Consistent system environment:** The Home button quits your app, and the volume controls take care of audio, just like you'd expect them to. User preference settings can be made available in the Settings application to avoid cluttering your app's user interface. And your iPad and iPhone/iPod touch apps can coexist on an iPad with Web services and apps created in HTML5.
- **Breathtaking imagery:** Photos and video already look fantastic on this display, but the artwork you create yourself for your app should be set to 24 bits (8 bits each for red, green, and blue), plus an 8-bit alpha channel to specify how a pixel's color should be merged with another pixel when the two are overlaid one on top of the other. In general, the PNG format is recommended for graphics and artwork.

In the following sections, you get to dive into some of the major features, grouped into the following major areas:

- Accessing the Internet
- Tracking location
- Tracking motion
- Supporting multifinger gestures and touches
- Playing content
- Accessing the content of Apple's supplied apps (such as Contacts and Photos)
- Taking advantage of the iPad display

Accessing the Internet

An iPad can access Web sites and servers on the Internet through Wi-Fi or optional 3G services. This Internet access gives you the ability to create apps that can provide real-time information. An app can tell a user, for example, that the next tour at the Tate Modern in London is at 3 p.m.

This kind of access also allows you, as the developer, to go beyond the limited memory and processing power of the device and access large amounts of data stored on servers, or even offload the processing. You don't need all the information for every city in the world stored on the iPad, nor do you have to strain the iPad processor to compute the best way to get someplace on the Tube. You can send the request to a server for all that information, especially information that changes often.

TECHNICAL STUFF

This technique is called *client-server computing* — a well-established software architecture where the client provides a way to make requests to a server on a network that's just waiting for the opportunity to do something. A Web browser is an example of a client accessing information from other Web sites that act as servers.

Knowing the location of the user

You can create an app that can determine the device's current location or even be notified when that location changes, using the iPad's location services. As people move, it may make sense for your app to tailor itself to where the user is, moment by moment.

Many iPad and iPhone apps use location information to tell you where the nearest coffee house is or even where your friends are. The iPadTravel411 sample application described in Part V uses this information to tell you where *you* are and give you directions to your hotel.

When you know the user's location, you can even put it on a map, along with other places he or she may be interested in. You find out how easy it is to add a map to your app in Chapter 15.

Tracking orientation and motion

The iPad contains three *accelerometers* — devices that detect changes in movement. Each device measures change along one of the primary axes in three-dimensional space. An app can, for example, know when the user has turned the device from vertical to horizontal orientation, and it can change the view from portrait to landscape if doing so makes for a better user experience.

You can also determine other types of motion such as a sudden start or stop in movement (think of a car accident or fall) or the user shaking the device back and forth. It makes some way-cool features easy to implement — for example, the Etch A Sketch metaphor of shaking the iPad to undo an operation. You can even control a game by moving the iPad like a controller — such as the aforementioned Need for Speed Shift game for the iPad (Electronic Arts), in which you drive the car by using the iPad like a steering wheel.

Tracking user's fingers on the screen

People use their fingers to select and manipulate objects on the iPad screen. The moves that do the work, called *gestures,* give the user a heightened sense of control and intimacy with the device. Several standard gestures — tap, double-tap, pinch-close, pinch-open, flick, and drag — are used in the applications supplied with the iPad.

TIP

You may want to stick with the standard gestures in your app, just because folks are already aware of (and comfortable with) the current pool, but the iPad's multifinger gesture support lets you go beyond standard gestures when appropriate. Because you can monitor the movement of each finger to detect gestures, you can create your own.

Playing content

Your iPad app can easily play audio and video. You can play sound effects or take advantage of the multichannel audio and mixing capabilities available. You can even create your own music player that has access to all the audio synced to the iPad from the user's iTunes Library. You can also play back many standard movie file formats, configure the aspect ratio, and specify whether controls are displayed. You can put up pages that look like Web pages or book pages if you want, and you can easily mix content for an immersive experience.

Accessing information from Apple's apps

Your app can access the user's information in the Contacts app on the iPad and display that information in a different way or use it as information in your application. For example, a user could enter the name and address of a hotel, and the application would file it in the user's Contacts database. Then, when the user arrives at Paddington Station, the application can retrieve the address from the Contacts app and display directions. What's more, your app can also present standard interfaces for picking and creating contacts.

What you can do with Contacts, you can do in a similar fashion with the Calendar app. Your app can remind a user when to leave for the airport or create calendar events based on what's happening this week in London. These events show up in the Calendar app and in other apps that support that framework.

Your app can also access the Photo library in the iPad Photos app, not only to display them, but also to use or even modify them. For example, the Photos app lets you add a photo to a contact, and several applications enable you to edit your photos on the iPad itself.

Copying, cutting, and pasting between apps

iOS (the iPad and iPhone operating system) provides support for Copy, Cut, and Paste operations within and between applications. It also provides a context-sensitive Edit menu that can display the Copy, Cut, Paste, Select, Select All, and Delete system commands. That means that while each iPad application is generally expected to play only in its own sandbox, you actually do have ways to send small amounts of data between applications.

Multitasking, background processing, and notifications

Although iOS doesn't have true multitasking (in fact, devices need multiple cores or CPUs to offer true multitasking), it has instant-on task switching that reduces application startup and makes it easier to continue right where you left off. For certain kinds of applications, you can also process events in the background. Such applications include the following:

- **Audio:** The application plays audio in the background.
- **Location:** The application processes *location events* (information the iOS sends to your app about changes in location) in the background.
- **VoIP:** The application provides the ability for the user to make Voice over Internet Protocol calls — turning a standard Internet connection into a way to place phone calls.

iOS also offers *push* notifications for receiving alerts from your remote servers even when your app isn't running, and *local* notifications which you can use in your app to alert users of scheduled events and alarms in the background (no servers required). You can use local notifications to get a user's attention; for example, a driver navigation application running in the background can use local notifications to alert the user when it's time to make a turn. Applications can also schedule the delivery of local notifications for a future date and time and have those notifications delivered even if the application isn't running.

Living large on the big screen

The iPad display offers enough space to show a laptop application (which is one reason why Web pages look so great). You can organize your app with a master list and detailed list of menu choices, or in a layout for landscape orientation with a source column on the left and a view on the right — similar to the Mac OS X versions of iTunes and iPhoto and exemplified by the Contacts app on the iPad.

If you're familiar with iPhone apps and Mac OS X applications, think somewhere in-between. With the iPad touch-sensitive display, you no longer have to create different screens of menus (as you might for an iPhone app) or deploy drop-down menus and toolbars (as you might for an Mac OS X app) to offer many functions.

For example, to crop and mask out parts of an image in Apple's Keynote app for the iPad (the app that lets you create slide shows), you don't have to select a photo and then hunt for the cropping tool or select a menu item — just double-tap the image, and a mask slider appears. In Apple's Numbers app for the iPad, if you double-tap a numeric formula, the app displays a special numeric and function keyboard rather than a full text keyboard — and the app can recognize what you're doing and finish the function (such as a Sum function) for you.

These are examples of redesigning a known type of application to get rid of (or at least minimize) that modal experience of using a smartphone app — that sinking feeling of having only one path of communication to perform a task or supply a response. iPad applications should allow people to interact with them in nonlinear ways. Modality prevents this freedom by interrupting a user's workflow and forcing the user to choose a particular path.

TIP

Lists are a common way to efficiently display large amounts of information in iPhone apps. Lists are very useful in iPad apps, too, but you should take this opportunity to investigate whether you can present the same information in a richer way on the larger display.

Embracing the iPad's Limitations

Along with all those features, however, the iPad has some limitations. The key to successful app development — and to not making yourself too crazy — is to understand those limitations, live and program within them, and even learn to love them. (It can be done. Honest.) These constraints help you understand the kinds of applications that are right for this device.

Often, it's likely that if you *can't* do something (easily, anyway) because of the iPad's limitations, then maybe you shouldn't.

The iPad evolved from the iPhone and iPod touch, and there are related limitations you need to consider, as well as a few things left out. So learn to live with and embrace some facts of iPad life:

- Users have fat fingers. You may think that the iPad's larger display makes that relatively easy to deal with, but keep in mind that you may want to design a multiuser app for the iPad that takes into account multiple fingers. (Anyone for a nice game of Touch Hockey?)
- Memory and battery power are limited, just like on an iPhone or iPod touch. This limitation may or may not be a decisive factor, depending on what kind of app you want to create, but smaller apps generally perform better.
- Although users can switch from one app to another instantly, and apps can continue where a user left off, only one application actually runs at a given time — again, just like an iPhone or iPod touch — with some apps capable of running in the background to serve notifications or play music.
- A camera isn't included in the first version of the iPad, but your iPad app can access the synced Photo library as well as synced contacts.

The next sections help get you closer to a state of iPad enlightenment.

Designing for fingers

Although the Multi-Touch interface is a feature of both the iPad and the iPhone/iPod touch, it brings with it some limitations — although not as many as with the smaller iPhone/iPod touch displays.

First of all, fingers aren't as precise as a mouse pointer, which makes some operations even more difficult on an iPhone or iPod touch than on an iPad (text selection, for example). Still, due to fat fingers, user-interface elements need to be large enough and spaced far enough apart so that users' fingers can find their way around the interface comfortably. Apple recommends that anything a user has to select or manipulate with a finger be a minimum of 44 x 44 pixels in size.

Because it's so much easier to make a mistake using fingers, you also need to ensure that you implement a robust — yet unobtrusive — Undo mechanism. You don't want to have your users confirm every action (it makes using the app tedious), but on the other hand, you don't want your app to let anybody mistakenly delete a page without asking, "Are you *sure* this is what you *really* want to do?" Lost work is worse than tedious. Fortunately, the iPad supports the same shake-to-undo feature as the iPhone.

Balancing memory and battery life

As an app designer for the iPad, you have several balancing acts to keep in mind:

- Although significant by the original Macintosh's standards, the computer power and amount of memory on the iPad are limited.
- Although access to the Internet can mitigate the power and memory limitations by storing data and (sometimes) offloading processing to a server, those Internet operations eat up the battery faster.
- Although the iPad power-management system conserves power by shutting down any hardware features that are not currently being used, a developer must manage the trade-off between all those busy features and shorter battery life. Any app that takes advantage of Internet access, core location, and the accelerometer is going to eat up the batteries.

REMEMBER

The iPad is particularly unforgiving when it comes to memory usage. If you run out of memory, in order to prevent corruption of other apps and memory, the system will simply *shut down your app* (unfortunately not to the tune of "Shut Down" by the Beach Boys).

It just goes to show that not *all* limitations can be exploited as "features."

Why Develop iPad Applications?

Because you can. Because it's fun. And because the time has come (today!). iPad apps are busting out all over, and developers have been very successful. Even high-profile magazines such as *Wired* and *The New Yorker* now offer app versions.

Developing iPad apps can be the most fun you've had in years, with very little investment of time and money (compared with developing for platforms like Windows). Here's why:

- **iPad apps are usually bite-sized, which means they're small enough to get your head around.** A single developer — or one with a partner and maybe some graphics support — can do them. You don't need a 20-person project with endless procedures and processes and meetings to create something valuable.
- **The applications tend to be crisp and clean, focusing on what the user wants to do at a particular time and/or place.** They're simple but not simplistic. This makes application design (and subsequent implementation) much easier and faster.
- **The apps use the most innovative platform available for mobile computing.** The iPad is a game-changer. It's completely changing the Internet as a publishing medium, the software industry with regard to applications, and the mobile device industry with regard to the overall digital media experience.
- **The free iOS Software Development Kit (SDK) makes development as easy as possible.** This book reveals the SDK in all its splendor and glory. If you can't stand waiting, you *could* go on to Chapter 4, register as an iOS developer, and download the SDK . . . but (fair warning) jumping the gun leads to extra hassle. It's worth getting a handle on the ins and outs of iPad app development beforehand.

The iPad has three other advantages that are important to you as a developer:

- **You can distribute your app through the App Store.** Apple will list your app in the App Store in the category you specify, and the store takes care of credit-card processing (if you charge for your app), hosting, downloading, notifying users of updates, and all those things that most developers hate doing. Developers name their own prices for their creations or distribute them free; Apple gets 30 percent of the sales price of commercial apps, with the developer getting the rest. However, keep in mind that Apple must approve your app before it appears in the App Store — see Chapter 6 for details on submitting your app and jumping through the hoops to get it approved.
- **Apple has a robust yet inexpensive developer program.** To place your app in the store and manage it, you have to pay $99 per year to join the Individual or Company version of the iOS Developer Program (which includes iPad development support). (Apple also offers an Enterprise version for $299 per year to develop proprietary, in-house iOS applications that you can distribute to employees or members of your organization, and a free University version for education institutions to include

iOS development as part of a curriculum.) But that's it. There are none of the infamous hidden charges that you often encounter, especially when dealing with credit-card companies. Go to the Apple iOS Developer site (`http://developer.apple.com/programs/ios`) and click the Enroll Now button. Chapter 4 describes how to register as a developer and join the iOS Developer Program.

- **It's a business and productivity tool.** The iPad has become an acceptable business and individual productivity tool, in part because it has tight security as well as support for Microsoft Exchange and Office, but even more for its design as a hand-held mobile computer. Using an iPad with a customer to interact with information is a lot more engaging and cool, and it helps salespeople close faster. Automobile finance companies can begin the credit-application process while customers are standing near a vehicle. Doctors and nurses at hospitals are beginning to use iPads to view X-rays and CT scans and read medical records while standing next to the patient. This happy state of affairs expands the possible audience for your application.

Developing with Apple's Expectations in Mind

Just as the iPad can extend the reach of the user, the device possibilities and the development environment can extend your reach as a developer. To make sure you're reaching in the right direction, it helps to understand Apple's perspective on what iPad apps should be — the company clearly has done some serious thinking about it, far longer than anybody else out there, having taken years to bring the iPad to market under a veil of secrecy.

So what does Apple think? Spokespeople often talk about three different application styles:

- **Productivity applications use and manipulate information.** The iPadTravel411 sample app that I show in this book is an example, and so are Bento and FileMaker Go (FileMaker), and Apple's iWork apps — Keynote, Pages, and Numbers. Common to all these apps is the use and manipulation of multiple types of information. (I'm not talking about the Productivity category in the App Store — that's a marketing designation.)
- **Utility applications perform simple, highly defined tasks.** The preinstalled YouTube app is an example — it deals only with the YouTube videos. The Brushes app for painting (Steve Sprang) is considered a

utility, as it performs a simple, highly defined task. (Again, I'm not talking about the Utilities category in the App Store, although many of those apps are considered utility apps because they perform simple, highly defined tasks.)

- **Immersive applications are focused on delivering — and having the user interact with — content in a visually rich environment.** A game is a typical example of an immersive application.

Although these categories help you understand how Apple thinks about iPad apps (at least publicly), don't let them get in the way of your creativity. You've probably heard *ad nauseam* about stepping outside the box. But hold on to your lunch; the iPad "box" isn't even a box yet. So here's a more extreme metaphor: Try diving into the abyss and coming up with something really new.

An Overview of the Development Cycle

To keep from drowning in that abyss, you need a plan to guide you through it. Socrates anticipated software development when he said that there's nothing stable in human affairs. Tacitus, with more data in hand 450 years later, saw that in all things there is a law of cycles. By the late 1960s, the Jefferson Airplane singers were singing, "roll with the natural flow, like water off a spinning ball."

In plain words, your software development plan is a cycle; perhaps a vicious cycle, but it can be a cycle through the park. You may repeat procedures within the cycle iteratively until you get it right, but the key to understanding the cycle is the recognition that once you spin off version 1 of your app, you start all over again to develop an update.

In general terms, the software development cycle is the process of creating or altering a software product or service. Theorists have created models and methodologies for defining this cycle. Although there are at least half a dozen models (Neal's a recovering software development methodologist), the one I go through here is pretty simple and is well suited for the iPad to boot. Here goes:

1. Defining the problems
2. Designing the user experience
    a. Understanding the real-world context
    b. Understanding the device context
    c. Categorizing the problems and defining the solutions
3. Creating the program architecture
    a. A Main view
    b. Content views

   c. View controllers (which, as you learn in Chapter 7, display things on the screen and respond to user actions)

   d. Models (which, as you learn in Chapter 7, contain the app's data and logic)

4. Writing the code
5. Doing it until you get it right

Of course, the actual analysis, design, and programming (not to mention testing) process has a bit more to it than this — and the specification and design definitely involve more than what you see in this book. But from a process perspective, it's pretty close to the real thing. It does give you an idea of the questions you need to ask — and have answered — in order to develop an effective iPad application.

A word of caution, though. Even though iPad apps are much easier to get your head around than, say, a full-blown enterprise service-oriented architecture, they come equipped with a unique set of challenges. Between the iPad platform limitations and the high expectation of all the new iPad users, you'll have your hands full.

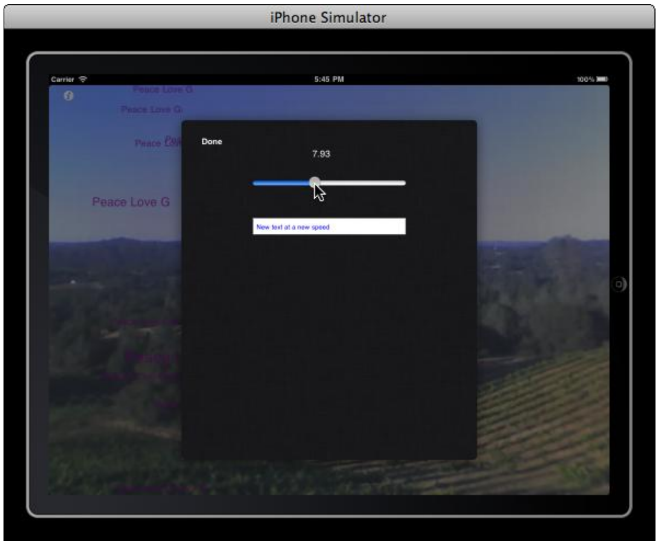
The Sample Applications

It's hard enough to understand how to develop an app, and even harder if the first example you turn to is too complex to get your head around. The first sample app, DeepThoughts (shown in Figure 1-2), which you find out how to build in Part IV, is simple enough to understand, and yet it demonstrates enough of the building blocks for creating a sophisticated iPad app that you should have no trouble following along and building it. With a little more (although not much more) work, you can use the development environment to actually create something of value.

DeepThoughts displays whatever text you enter in a flowing animation that fills the display, supposedly suggesting a meditative state (as in "peace love groovy music"). You can speed up or slow down the animation by swiping left or right, so you find out how to deal with that simple gesture, as well as with tapping an Info button or the display to change settings (such as the speed). You also learn how to accept user input — sliding a slider for speed, and entering text for the words.

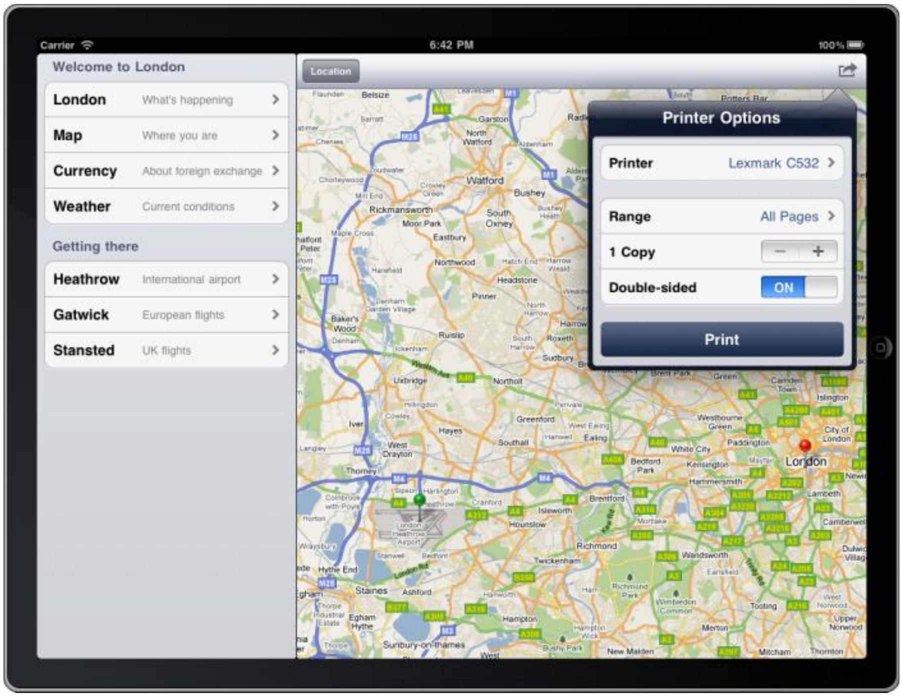
After you know a bit more about the application design cycle and what makes a good user interface, and even more (actually quite a bit more) about the iPad technologies that work behind the screen — such as frameworks, windows, views, and view controllers — and then just a few more details about getting your app ready for the App Store and the public, you're ready to do some real coding — the DeepThoughts app.

After that, you find out about the design of the iPadTravel411 app (shown in Figure 1-3), starting in Chapter 13. You find out how to use a Split view, present a map, work with Table views, add content, access data on the Web, include data with your app, and allow users to set preferences.



New text at a new speed

Figure 1-2: This book will provoke Deep-Thoughts.



Welcome to London

London	What's happening
Map	Where you are
Currency	About foreign exchange
Weather	Current conditions

Getting there

Heathrow	International airport
Gatwick	European flights
Stansted	UK flights

Printer Options

Printer	Lexmark C532
Range	All Pages
1 Copy	− +
Double-sided	ON

Print

Figure 1-3: This sample app may provoke a trip to London.

What's Next

You must be raring to go now and just can't wait to download the Software Development Kit (SDK). That's exactly what many new developers do — and later are sorry that they didn't spend more time upfront understanding the iPad user experience, how applications work in the iPad environment, and the guidelines that Apple enforces for apps to be approved for the App Store.

So be patient. The Hitchhiker's Guide to the Galaxy, that wonderful fantasy of an iPad from 1979, suggests that space is "big, really big . . . you just won't believe how vastly hugely mind-bogglingly big it is" and suggests that you bring a towel. The guide says a towel "is about the most massively useful thing an interstellar hitchhiker can have." (Again with Douglas Adams? But I promise not to get into the meaning of life, the universe, and everything, or the ultimate question — for which the answer is 42.) This book is your towel for the journey. The following chapters cover all the aspects of development you need to know before you spend time coding. Then, I promise, it's off to the stars.

Chapter 2

Creating a Compelling User Experience

In This Chapter

▶ Understanding what makes an iPad app different

▶ Making an app that's worth the cost

▶ Designing for the best user experience

▶ Avoiding the pitfalls that get apps rejected

When you have a handle on the possibilities and limitations of the iPad, your imagination is free to soar to create a compelling user experience. But what is a "compelling user experience," really?

For openers, a compelling user experience has to result from the interaction of several factors:

- Interesting, useful, and plentiful content that fills the display in an immersive experience
- Content relevant to what you're doing, where you are, what your next activity might be, and how you're holding the device
- An intuitive, well-designed user interface designed for the full Multi-Touch display that supports multifinger gestures

The iPad allows both immediacy and intimacy as it blends mobility and the power of the desktop to create a new kind of freedom. I like to use the term *user experience* because it implies more than a pretty user interface and nice graphics. A *compelling* user experience enables users to do what they need to do with a minimum of fuss and bother. It includes meeting the user's expectations based on the *context* — all the stuff going on around that user — in which the app is used. A guidebook app may have a great user interface, for example, but it may not give me the most up-to-date information or let me know that a tour of the Houses of Parliament is leaving in five minutes from the main entrance. Without those added touches, I don't consider an app compelling.

If you've developed applications for a desktop or laptop, or even for an iPhone or iPod touch, you have to rethink your design for any new app you create for the iPad, because the iPad is a singular game-changer that introduces an entirely new set of user interaction features. Albert Einstein once said that technological change is like an axe in the hands of a pathological criminal. If you've developed user interfaces before, you may want to adopt this attitude — grab an axe to chop through your previous design ideas and take a hard look at the newest apps that are just now arriving on the iPad (especially the ones from Apple), and remember Pablo Picasso's immortal words: "Bad artists copy. Good artists steal."

Apple goes out of its way to provide sample code for many of the neater tricks and features out there, all in hopes of demystifying how they work. Apple's supplied apps, including Contacts, Photos, and Calendar, are already on your iPad and ready to be examined for user interface ideas. In addition, Apple's iWork apps for the iPad — Keynote for slide presentations, Numbers for spreadsheets, and Pages for word processing and page formatting — are excellent examples to crib from, especially for productivity apps.

This chapter gently urges you to reinvent the user experience for your app to match an iPad user's expectations, from the perspective of the content you provide and the app's functionality. But first, you need to envision the totality of what your app's user experience should be.

Deep Thoughts on the User Experience

Pun intended. Creating DeepThoughts, the star of Part IV, is a fast way to get familiar with iPad software development. DeepThoughts was designed to do only one thing, so that it would be easy to understand and quick to create as you follow along with the examples. But DeepThoughts is also the skeleton of an immersive app. It's similar to any app that lets the user interact with content in a visually rich environment, except that it provides only a single view and a single piece of content. With the DeepThoughts application under your belt, you'll have a much easier time understanding and using all the resources Apple provides to help you develop iPad apps.

iPadTravel411, the star of Part V, starts in many ways where DeepThoughts leaves off — by providing a more immersive experience in a *productivity* app — a kind of app that uses and manipulates different types of information. iPadTravel411 manipulates foreign currency rates, airport transportation routes, maps, your location, weather, events, and a traveler's tasks.

Because of its ease of use and convenience, its awareness of your location, and its ability to connect seamlessly to the Internet from most places, the iPad lets you develop a totally new kind of application — one that integrates seamlessly with what the user is doing when he or she is living in the real world (what a concept). It frees the user to take advantage of technology away from the tether of the desk or coffee shop, and skips the hunt for a place to spread out the hardware. I refer to such applications as *here-and-now* apps that take advantage of technology to help you do a specific task with up-to-date information, wherever you are and whenever you'd like.

Although iPhone apps share some of these characteristics, iPad apps can offer page-like experiences due to the larger display, including the ability to provide well-organized content in a *Split view,* which displays more than one view onscreen at a time. With Split view in portrait orientation, you can present content to choose from in a master list on top, and more detailed choices in a separate view on the bottom, as demonstrated by the App Store app on the iPad. You can also use a layout for landscape orientation that offers menu choices in a source column on the left, with different views based on those choices on the right, as I do in iPadTravel411 (see Figure 2-1, right side — "Welcome to London" is the source column). The Split view is a common organizational element in iPad applications because it helps flatten the information hierarchy, and you find out how to take advantage of it in iPadTravel411 as described in Chapter 15.

Whenever possible, add a realistic, physical, true-to-life dimension to the way your app looks and behaves so that it's easy to understand and fun to use — such as the Need for Speed Shift app. Remember that a great user interface follows design principles that are based on the way people think and work. A user interface that's unattractive, convoluted, or illogical can make even a great app seem like a chore to use. But a beautiful, intuitive, compelling experience inspires a positive emotional attachment in users, and that's a good thing for an app to do.

All the features inherent in iPad apps enable you to add a depth to the user's experience that you usually don't find in laptop- or desktop-based applications — in effect, a third dimension. Not only can the use of an iPad app be part of what you're doing and where you are, but *what you're doing and where you are* can be part of the app. iPad developers can achieve a goal that's been elusive for years: the seamless integration of technology into everyday life.

The why-bother-since-I-have-my-laptop crowd still has to wrestle with this level of technology, especially those folks who haven't grown up with it. They look at an iPad as a poor substitute for a laptop or desktop — well, okay, for certain tasks, that's true. But an iPad app trumps the laptop or desktop big-time in two ways:

- **Portability:** The iPad's compact portability lets you do stuff not easily done on a laptop or desktop — on site and right now — as with the iPadTravel411 app you find out how to build in Part V.
- **Activity integration:** The iPad is integrated into the activity itself, creating a transparency that makes it as unobtrusive as possible. This advantage — even more important than portability — is the result of *context-driven design.*

The key to designing a killer iPad application is to understand that the iPad is *not* a bigger iPod touch, nor is it a more portable version of a laptop computer. It's another animal altogether, as I describe in Chapter 1, and is therefore used entirely differently. With maps that you can zoom into, full-screen videos, and slide shows with music you can play, and so on, you can provide content in your app that people can enjoy anywhere and more easily share with others, far more easily than they can with a laptop or an iPod touch. So get ready to take a closer look at how to create compelling content.

Creating Compelling Content

It's a powerful experience to hold full pages of content (from the Internet or in memory) in your hands in a device that weighs little more than a thick magazine. But keep in mind that the iPad has no default orientation — people don't pay much attention to the minimal device frame, and they're unconcerned with the location of the Home button. They can rotate from portrait to landscape orientation with ease, and your app should encourage people to interact with iPad from any side by providing a great experience in all orientations. You can see how the iPadTravel411 looks in Figure 2-1, which shows a full-screen map in portrait orientation and a split-screen map in landscape orientation.

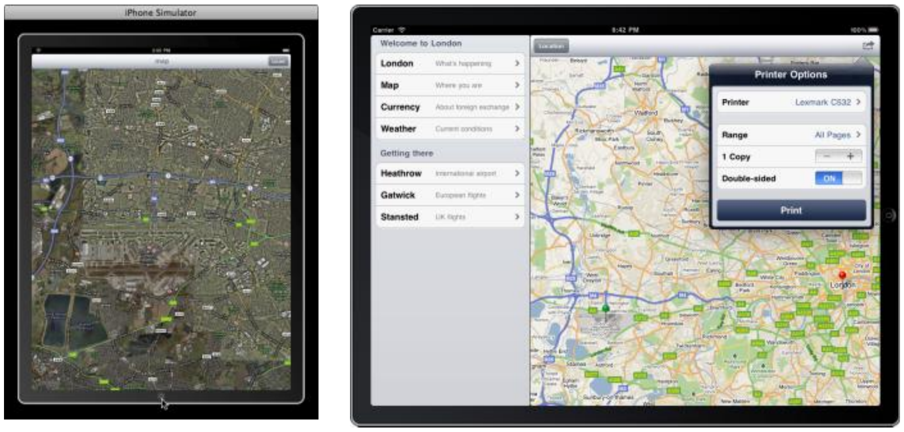


Figure 2-1: iPadTravel411 in portrait orientation (left) and landscape orientation, showing a split view and popover (right).

The large iPad display offers many ways to give people access to information all in one place, without having to switch screens (as in an iPhone) or open separate windows and modal dialogs (as in desktop applications). If your app currently provides information in a hierarchy (such as a sequence of iPhone screens), I strongly consider that you flatten the hierarchy to present more information in one place. Your app's content can appear in the Main view, and if you need to provide additional information or tools in an auxiliary view, you can employ a *popover* (a view that appears on top of the main view) so that users don't have to leave the context of the main task.

Focusing on the task at hand

What most of the really good iPad apps have in common is *focus:* They address a well-defined task. The best iPad apps give people innovative ways to interact with content while they perform a clearly defined, finite task. You should resist the temptation to fill the display with features that aren't directly related to the main task. Concentrate instead on ways to amplify the user experience, without diluting the main task with extraneous features.

For example, a book-reader app that also allows people to keep track of reading lists shouldn't make people leave the book page to view another screen to manage their reading lists. Rather, the app should put the list in a translucent popover that appears above the page and should allow people to copy favorite passages into it. In a football game app, users should be able to see information about characters without leaving the Field view.

The content itself then, especially for here-and-now apps, must be streamlined and focused on the fundamental pieces of the task. Although you *can* provide a near-infinity of detail just to get a single task done, here's a word to the wise: Don't. You need to extract the essence of each task; focus on the details that really make a difference.

When you're using a good app, every piece of the app is not merely important to the task, but also important to *where you are in the task.* For example, if you're trying to decide how to get to central London from Heathrow, the app shouldn't offer detailed information about the Tube until you need it.

Maintaining consistency with the user's world

Great apps are based on the way people — users — think and work. When you make your app a natural extension of the user's world, it makes the app much easier and more pleasant to use and to learn.

Your users already have a mental model that describes the task your software is enabling. The users also have their own mental models of how the device works. At the levels of both content and user interface, your app must be consistent with these models if you want to create a superb user experience (which in turn creates loyalty to your app).

The user interface in iPadTravel411 was based on how people divide the tasks they need to do when traveling, especially when arriving at an airport such as London's Heathrow. Here are typical categories:

- **See a map of the territory.** You'd want to see a map right away that shows your current location and lets you pin other locations on it (such as your hotel) so you can find them quickly.
- **Deal with foreign currency.** You need to know how much it really costs to convert money and buy things abroad.
- **Check the current weather and forecast.** You don't want to walk outside the airport terminal into a driving rainstorm without your coat on, and you want to know what to expect over the next few days.
- **Look up events.** You may want to check any special events happening while you're in the city so that you can avoid traffic around them or find out the schedule for an event you're attending.
- **Find transportation.** You want to know how to get to and from the airport with maximum efficiency and minimum hassle, as well as how to get around the city.

This is only a partial list, of course. Chapter 13 gets into the iPadTravel411 application design in more detail.

You can divide the tasks in other ways, but anything much different would be ignoring the user's mental model, which would mean the app wouldn't meet some of the user's expectations. It would be less pleasant to use because it would impose an unfamiliar way of looking at things instead of building on the knowledge and experiences those users already have. Basing your app on how the user interacts and thinks about the world makes designing a great user interface easier.

Modeling apps on real-world metaphors

When possible, model your application's objects and actions on objects and actions in the real world. For example, the Settings app displays on-off switches you can slide to turn things on or off. Many e-book readers let you flick the screen as if it were a paper page.

All these interface details are based on physical counterparts in the real world. You need to help people focus on the content, and one way is to design your app as a subtle frame around the information they're interested in, like the App Store app. In the App Store, the content (in this case, apps for download) appears in a carousel that reminds one of a diner jukebox at your table — and everyone who uses iTunes already knows how to navigate through the album cover art choices.

Consider creating custom controls that subtly integrate with your app's graphical style. In this way, controls are discoverable, but not too conspicuous. The car-driving metaphor in the Need for Speed Shift app (refer to Figure 2-1, right side) is so right for a touch-sensitive display that it's sexy: You can shift gears with your finger and tap other controls that appear like they belong on the dashboard while you cruise.

Engaging the user

While I'm on the subject of shifting gears, here are two more important aspects of a compelling application: direct manipulation and immediate feedback. Here's what's so great about them:

- **Direct manipulation makes people feel more in control.** On the desktop, it means a keyboard and mouse; on the iPad, the Multi-Touch interface serves the same purpose. In fact, using fingers gives a user a more immediate sense of control; there's no intermediary (such as a mouse) between the user and the object onscreen. To make this effect happen in your app, one way is to keep your onscreen objects visible while the user manipulates them.
- **Immediate feedback keeps the users engaged.** Great apps respond to every user action with some visible feedback — such as highlighting list items briefly when users tap them.

Also, consider fading controls after people have stopped interacting with them for a little while, and redisplaying them when people tap the screen. This gives even more of the screen space to the content people want to see.

Because of the limitations imposed by using fingers, apps need to be very forgiving. For example, you don't want your app to pester the user to confirm every action, but you also don't want the app to let the user perform potentially destructive, nonrecoverable actions (such as deleting all contacts or restarting a game) without asking, "Are you sure?" It should also be obvious to users how to stop a task that's taking too long to complete.

Making it obvious

Although simplicity is a definite design principle, great apps are *also* easily understandable to the target user. If you're designing a travel app, it has to be simple enough for even an inexperienced traveler to use. But if you're designing an app for foreign exchange trading, you don't have to make it simple enough for someone with no trading experience to understand.

Keep these points in mind as you plan and create your app:

- The main function of a good application is immediately apparent and accessible to the users it's intended for.
- The standard interface components also give cues to the users. Users know, for example, to touch buttons and select items from popovers.
- You can't assume that users are so excited about your app that they're willing to invest lots of time in figuring it out.

Early Macintosh developers were aware of these principles. They knew that users expected that they could rip off the shrink-wrap, put a floppy disk in the machine (these were *really* early Macintosh developers), and do at least something productive immediately. The technology has changed since then; user attitudes, by and large, haven't.

Your application's text should be based on what the target user expects to see. For example, if your user isn't steeped in technical jargon, avoid it in the user interface.

Avoiding jargon doesn't mean that you have to dumb down the app. Here are some guidelines:

- If you're targeting your app toward people who already use (and expect) a certain kind of specialized language, then sure, use the jargon in your app. Just do your homework first and make sure you use those terms *correctly*.

  For example, if your app is targeted at high-powered foreign-exchange traders, it might use *pip* (price interest point — the smallest amount that a price can move, as when a stock price advances by one cent). In fact, a foreign-exchange trader expects to see price movement in pips, and not only *can* you use that term in your user interface, you *should*.
- If your app requires that the user have a certain amount of specialized knowledge about a task in order to use your application, identify what that knowledge is upfront.
- If the user is an ordinary person with generalized knowledge, use ordinary language.
- Gear your app to your user's knowledge base. In effect, meet your users where they are; don't expect them to come to you.

Don't underestimate the effect of the user interface on the people who are trying to use it. A bad user interface can make even a great app painful to use. If users can't quickly figure out how to use your app or if the user interface is cluttered or obscure, they're likely to move on and probably complain loudly about the app to anyone who will listen — or worse, give your app a lousy review and a bad rating in the App Store.

Using stunning graphics with aesthetic integrity

Appearance has a strong impact on how people perceive your app's value. As mentioned previously, an app that appears cluttered or illogical is hard to understand and use. The high-resolution display supports rich, beautiful, engaging graphics that can draw people into an application and make the simplest task rewarding.

It's a pretty safe bet that part of the appeal of the iPad to many people — especially to nontechnical users — is aesthetic: The device is sleek, compact, and fun to use. But the aesthetics of an iPad app aren't just about how beautiful your app is onscreen. Aesthetic integrity is about how well your app's appearance integrates with its function, as in the appearance of the car dashboard and the windshield view in the Need for Speed Shift app.

An immersive app like Need for Speed Shift offers what users expect — a beautiful appearance that promises a thrilling experience — but more importantly, its appearance is integrated with the task of driving a car, and the user interface elements are designed carefully so that they provide an internally consistent experience.

On the other hand, for productivity apps, you may want to keep decorative elements subtle while giving prominence to the main task. One of the early appeals of the prehistoric Macintosh and the recent iPhone was how similarly all the applications worked. Use the iPad standard behavior, gestures, and metaphors in standard ways. For example, users tap a button to make a selection and flick or drag to scroll a list. iPad users understand these gestures because the Apple-supplied apps use them *consistently*.

Fortunately, staying consistent is easy to do on the iPad; the frameworks at your disposal have that behavior built in. This is not to say that you should never extend the interface, especially if you're blazing new trails or creating a new game. For example, if you're creating a roulette wheel for the iPad, why not use a two-finger circular gesture to spin the wheel, even if it isn't a standard gesture?

Designing the User Experience

It's rare (except with sample apps) for an app's user experience to be simply a combination of some of the iPad's basic experiences. But DeepThoughts, which you build in Part IV, is simplicity itself. It displays whatever text the user enters, and the mechanism for changing the text is just like most other iPhone or iPod touch apps — touch the *i* (information) button or tap the display itself, and tap the text field to use the onscreen keyboard. As you build DeepThoughts, you discover how to use the basic building blocks of the iPad user experience.

The iPadTravel411 app in Part V presents a more complex set of problems. A traveler doesn't need a lot of information at any one time. In fact, the user wants as little info as possible (just the facts ma'am) but as current as possible. It doesn't help to have last year's train schedule.

To get the design ball of your application rolling, start thinking about what your user will want from the application — not necessarily the features, but what the experience of using the application should be like.

Understanding the real-world context

You can reach the goal of seamlessness and transparency by following some very simple principles when you design the user experience — especially with respect to the user interface.

Become the champion of relevance

There are two aspects to this directive:

- Search and destroy anything that isn't relevant to what the user is doing while he or she is using a particular part of your application.
- Include — and make easily accessible — everything a user needs when doing something supported by a particular part of your application.

You want to avoid distracting the user from what he or she is doing. The application should be integrated into the task, a natural part of the flow, and not something that causes a detour. Your goal is to supply the user with only the information that's applicable to the task at hand. If your user just wants to get from an airport into a city, he or she couldn't care less that the city has a world-renowned underground or subway system if it doesn't come out to the airport.

Seconds count

At first, the "seconds count" admonition may appear to fall into the "blinding flash of the obvious" category — of *course* a user wants to accomplish a task as quickly as possible. If the user has to scroll through lots of menus or figure out how the app works, the app's value drops exponentially with the amount of time it takes to get to where the user needs to be.

But there are also some subtleties to this issue. If the user can do things as quickly as possible, he or she is a lot less distracted from the task at hand — and *both* results are desirable. If your app's user switches to another app and then back to your app, your app should be in the same state it was in before the user quit — showing the same views and information (such as the map in iPadTravel411, which shows the same location you just viewed).

Combine these ideas and you get the principle of *simply connect:* You want to be able to connect easily whether that connection is to a network, to the information you need, or to the task you want to do. For example, a friend of mine was telling me he uses his iPad when watching TV so he can look up things in an online dictionary or Wikipedia. (He must watch a lot of public TV.)

Doing it better on the iPad

What you get by using the application has to have more value than alternative ways of doing the same thing.

The quality of information has to be better than the alternative

You can find airport transportation in a guidebook, but it's not up to date. You can get foreign exchange information from a *bureau de change,* but unless you know the bank rate, you don't know whether you're being ripped off. You can get restaurant information from a newspaper, but you don't know whether the restaurant has subsequently changed hours or is closed for vacation. If the app can consistently provide better, more up-to-date information, it's the kind of app that's tailor-made for a context-driven design.

The app has to be worth the real cost

By *real cost,* I don't mean just the amount the user actually pays out — you need to include the time and effort of using the app. The real cost includes both the cost of the application and any costs you might incur by *using* the application. This can be a real issue for an app that requires the Internet, because international roaming charges can be exorbitant for using data services to access the Internet. That's why the app must have the designed-in capability to download the information it provides and then to update the info when you find a wireless connection.

Keep things localized

With the world growing even flatter (from a communications perspective, anyway) and the iPad available in countries on every continent, the potential market for an app is considerably larger than just the folks who happen to speak English. But having to use an app in a language you may not be comfortable with doesn't make for transparency. This means that applications have to be *localized* — that is, all the information, the content, and even the text in dialogs need to be in the user's language of choice.

Playing to the iPad's Strengths

The following features of the iPad and its operating system are key to creating applications that go beyond the desktop and that take advantage of context-based design:

- Sensing multiple fingers and multifinger gestures
- Tracking orientation and motion
- Displaying stunning graphics and images (which you can even show on a connected TV or projection system)
- Knowing the location of the device and displaying a compass
- Playing digital content and recording sound (and, of course, syncing with a computer for the content)
- Accessing the Internet via Wi-Fi or optional 3G service

There are others, of course, but you can expect to find many of these features in an iPad app.

Sensing multifinger gestures

The iPad Multi-Touch display gives you lots of room for multifinger gestures, including gestures made by more than one person. You can offer the standard swipe, pinch, and rotation gestures, among others, and use other gestures to trigger additional behavior, such as triple-tap and touch-and-hold (also called *long press*). If your app offers an important task that users perform frequently and want to complete quickly, you should probably use only standard gestures, but if you're designing an app that can measure up to some of the iPad games, with realistic controls, multiplayer support, or an environment for exploring, you should think about using custom or multifinger gestures.

However, to maintain consistency with other iPad apps, use standard gestures for standard behaviors. For instance, a pinching gesture should scale a view, zooming it in and out; it should not be interpreted as, say, a selection request, for which a tap is more appropriate. Stick with real-world models, and in this case stick with the metaphors that have already appeared on the iPhone: tapping iPod playback controls, sliding on-off switches, and flicking through the data shown on picker wheels.

There are some limitations you need to be aware of. Fingers aren't as precise as a mouse pointer, and user interface elements need to be large enough and spaced far enough apart so that the user's fingers can find their way around the interface comfortably. If you design interface elements to be integrated with your graphics and images, be sure to make them large enough.

Tracking orientation and motion

When you rotate the iPad from a vertical view (portrait) to a horizontal view (landscape), the accelerometer detects the movement and changes the display accordingly. The iPad can also sense other motion using its built-in accelerometer. Even in its simplest form, motion is useful: When entering text or using the copy and paste functions, you can just shake the iPad to undo the action.

Motion detection happens so quickly that you can control a game with these movements. Although the accelerometer is used extensively in games, it also has other uses, such as enabling a user to erase a picture or make a random song selection by shaking the device.

Displaying stunning graphics and images

To rise above the inevitable swarm of new iPad apps, you'll want to offer graphics and images that truly take advantage of the display. The iPad displays 1,024 x 768 pixels, with up to 24 bits per pixel (8 bits each for red, green, and blue), plus an 8-bit alpha channel, which specifies how the pixel's colors should be merged with another pixel when the two are overlaid one on top of the other.

In most cases, you wouldn't set the alpha channel on a pixel-by-pixel basis in a drawing or painting program, but rather on an object-by-object basis, so that different parts of the object would have different levels of transparency depending on how much you wanted the background to show through. With an alpha channel, you can create rectangular objects that appear as if they are irregular in shape — you define the rectangular edges as transparent so that the background shows through.

TIP

Always create your artwork in a larger multiple of the pixel dimensions you need, so that you can add depth and details before scaling it down accurately to the iPad display size. That way your graphics and images crackle and snap with clarity and color.

Playing and recording content

The iPad evolved from the iPod, in which content is king. Not only can your app play music and videos, it can also record voice-quality (actually telephone-quality) sound with its built-in microphone, or higher quality through external microphones. A number of audio-mixing apps have already made their debut in the App Store. The iPad can sync images from your computer's Photo library, send and receive images and video clips by e-mail, and share multimedia content through the MobileMe service.

Knowing the location of the device

Because the device knows its own location (and hence, the user's location), you can further refine the context by including the actual physical location and adding that to the relevance filter. If you're in Rome, the application can ask the user whether he or she wants to use Rome as a filter for relevant information (so that when in Rome . . .).

Because the iPad knows where it is, apps can make use of this information to present content that is closer to the user. The feature isn't just for travel — apps that have nothing to do with travel, such as an app that shows you the movies playing in your area, may still use location to improve the user experience.

Accessing the Internet

Accessing the Internet allows you to provide real-time, up-to-date information. In addition, it enables you to transcend the CPU and memory limitations of the iPad by offloading processing and data storage out to a server in the clouds.

Of course, there's always a possibility that the user may be out of range, or on a plane, or has decided not to pay exorbitant roaming fees for 3G and isn't close enough to a Wi-Fi hotspot. You need to account for that possibility in your application and preserve as much functionality as possible. This usually means allowing the user to download and use the current real-time information, where applicable.

Avoiding Practices that Get Apps Rejected

Apple exerts control over the app-development and App Store ecosystem, and if you want to play ball in Apple's ballpark, you have to, well, play ball. No matter how many developers complain about Apple's rejection policies, there will always be more developers willing to follow the guidelines. All you need to do is read the documentation, steer away from the Apple trademarks and images, and stay away from content that's questionable in any legal sense. By keeping those things in mind, you can make design decisions about your app now, before developing the app, which can save you time and money later.

Some people believe Apple has not only a right, but also an obligation, to police the App Store and reject questionable apps, if only to build trust with consumers. Anacharsis, one of Greek mythology's Seven Wise Men, warned people that the market is "the place set apart where men may deceive each other." Given the way the iPad can be integrated into your everyday life and communications, a malicious app could do considerably more damage than a similar one on a desktop computer.

But Apple also wants the user experience to be a rewarding one, as well as one that's consistent with the way Apple designed its own apps and OS. And that makes perfect sense for a company that wants to expand its ecosystem and user database so that it can continue to invest in research and keep innovation on the front burner.

So what kinds of things will get your app bounced before it ever has a chance to shine in the App Store? Here are just a few:

- **Linking to private frameworks:** Apple rejects apps that call external frameworks or libraries that contain non-Apple code, unless Apple has previously approved their use. In addition, you can't download interpreted code to use in an app except for code that is interpreted and run by Apple's published APIs and built-in interpreters. Private frameworks and interpreted code may hide functions that Apple would want to know about. (Some private frameworks have been found to mine personal information from iPhone users without their knowledge.)
- **Straying too far from Apple's guidelines:** When I submitted my iPhone app (Tony's Tips for iPhone Users), it was initially rejected because the app used highlighting in a menu in a way that did not conform to Apple's guidelines. Be sure to follow the guidelines that are published in the iOS Dev Center (which you find out how to access in Chapter 4).

- **Copying Apple's existing functionality:** Although you should use the functionality provided for developers, you shouldn't simply copy something that Apple already does. Mini Web browsers — apps that essentially show Web pages and do little else — are particularly vulnerable. For example, a simple iPhone app that duplicated the functionality of Safari's bookmark button was rejected.
- **Using an inappropriate keyboard type:** If your app needs a phone number or other numeral-only input, and it presents a keyboard that also includes the possibility of entering standard alphanumeric input, it will most likely be rejected.
- **Being oblivious about whether your user lost a connection:** The iPad is all about using the Internet. If your app uses a network connection, it is *your app's responsibility* to tell the user if and when his or her iPad loses its network connection while using your app.

Now that you have some idea about what Apple expects of you — in terms of designing and developing your app — it's time for you to find out what to expect of Apple in terms of supporting your development efforts. Next, you should find out the marketing challenges for apps in the App Store and even more practices that could either enhance or inhibit your ability to effectively distribute your apps. So, onward to the next chapter, where you find out all about the App Store and your chances of success with it.

Chapter 3

The App Store Is Not Enough

In This Chapter

- ▶ Reaching potential iPad customers
- ▶ Measuring the success of marketing campaigns
- ▶ Analyzing customer activity
- ▶ Enabling in-app purchases
- ▶ Integrating iAds (interactive advertisements) in your apps
- ▶ Making a business out of iPad software development

Peter Drucker, known as the father of modern management, is also known for pointing out that business has only two functions: innovation and marketing. Because most of this book is about innovation, I need to spend at least one chapter explaining why so many developers don't make enough money from iPad, iPod touch, and iPhone apps, and what you can do to mitigate the complex issues surrounding the marketing of these apps.

Apple will list your application along with at least 20,000 iPad apps and 225,000+ iPhone apps already listed in the App Store — remember that an iPad can also run all iPhone and iPod touch apps. Yes, it's wonderful that Apple takes only 30 percent of the sales price and takes care of hosting, downloading, credit-card processing, and notifying users of updates. And if you remember the early days of developing for game machines, you may appreciate the fact that Apple lets you name your own price for your app. You can even distribute an app for free. What you can't do, and perhaps this is a good thing, is pay for preferential treatment. And as of this writing, Apple doesn't accept advertising within the store, but it does offer a version of the iAds program for developers to advertise in other apps.

The App Store lists the top paid and free apps in each category, and it lists the newest apps by release date, but unless your app is already successful and in the top paid or top free lists, your app's fleeting appearance in the list sorted by release date may provide only a short spike in sales — unless you prepare yourself to take advantage of it by applying some of the methods in this chapter.

The trouble with using any kind of technology to reach customers is the same, old or new: measuring the results. “Half the money I spend on advertising is wasted,” according to a famous remark by John Wanamaker, founder of the first department store in Philadelphia (one of the first department stores in the United States) in 1861. “The trouble is, I don’t know which half.”

Why People Buy Apps from the App Store

Why are people adopting consumer apps in ever-larger numbers? Based on a Juniper Research study from July 2010, app downloads are expected to rise to more than 25 billion in 2015. Resolve Market Research conducted a survey to find that the number one reason to own an iPad is “It’s an entertaining and cool device.” One of the most popular paid apps is Friendly – Facebook Browser — people prefer to pay for the app rather than use the iPad’s free Safari browser to go to Facebook’s site.

Choosing to download apps is a lifestyle decision. People like to adopt a new lifestyle that changes the way they live and work. While many of the older generations saw the wonder of the Internet and still think in terms of connecting to the Web and using a browser — and still pull out credit cards to type the numbers when we order products — newer generations born with the Internet assume they are already connected. They use apps more than the browser, and they find credit cards inconvenient — they are used to simply clicking to pay for things. Apps make them far more powerful in their daily lives, and they are hungry for apps that can positively affect their lives and their work. Apps that are not cool, or don’t really have much of an effect on their lives, are mostly ignored.

The decision to download iPad apps for work and play is not so much a matter of convenience. It is the positive effect apps have on people’s lives — the immediacy of sharing with others, the exhilaration of being fashionable with the coolest apps, and the real-time control they feel when they use these apps.

The App Store is, of course, the only place to download iPad apps. (I’m not getting involved in the whole issue of sites that sell apps for “jailbroken” or otherwise hacked iPads, which is another topic.) Apple has created an ecosystem around the iOS platform for the iPad, iPhone, and iPod touch that opens it up equally to all developers for true innovation.

A major factor in the developer community’s acceptance of Apple’s “one-stop shop” is the App Store’s equal treatment of all customers. This equal treatment is a fact of life today in all but the most posh stores, but it was an innovation in John Wanamaker’s store in 1861. Wanamaker created the price tag because he believed that if everyone was equal before God, then everyone should be equal before price. (He also invented the cash refund and guaranteed the quality of his merchandise in print.) Apple has also established trust with its customers by screening apps before listing them and enforcing guidelines among app developers for a “quality experience” for consumers. And, of course, the price tag is right up front.

Speaking of a quality experience, people are attracted to new technologies *just for* the experience. Wanamaker embraced innovation as early as possible to attract customers with a new experience — his was the first department store with electrical illumination (1878), the first with a telephone (1879), and the first to install pneumatic tubes to transport cash and documents (1880). Today, people are attracted to the App Store’s use of technology, its ease of use, and this highly innovative form of shopping-on-demand right from your mobile device.

There is no substitute for combination of trust, equal treatment, and a high-quality experience. The App Store is *the* place to list your iPad, iPhone, and iPod touch apps. Marketing them, however, is entirely up to you.

Finding out how to reach your potential customers

The App Store is right at your iPad customers’ fingertips. Tap the Featured button on the bottom row of buttons, and the Featured screen appears, showing highlighted apps at the top, as shown in Figure 3-1. More apps appear under the New and Noteworthy heading. Featured, Top Charts, Categories, and Updates buttons appear along the bottom, ready to entice potential customers.

The Featured screen also includes the What’s Hot button at the top, showing the most popular apps based on downloads. The Featured screen’s New and Noteworthy list and the What’s Hot list are where early adopter customers go to buy on impulse. Your app may make a brief appearance in the New and Noteworthy list when you release it, only to be crowded out almost immediately by more new apps. There are, by my rough estimates as of this writing, about nine iPhone and iPod touch apps born in the App Store every hour of every day, and the developers are just getting started with iPad apps, which double or triple this rate.

But if you’ve properly categorized your app, it should appear in the list of apps on the screen devoted to that category. Attaching your app to the appropriate category, as I describe in Chapter 6, is extremely important. Customers looking for a social networking app tap the Social Networking category to find the apps they’re looking for.

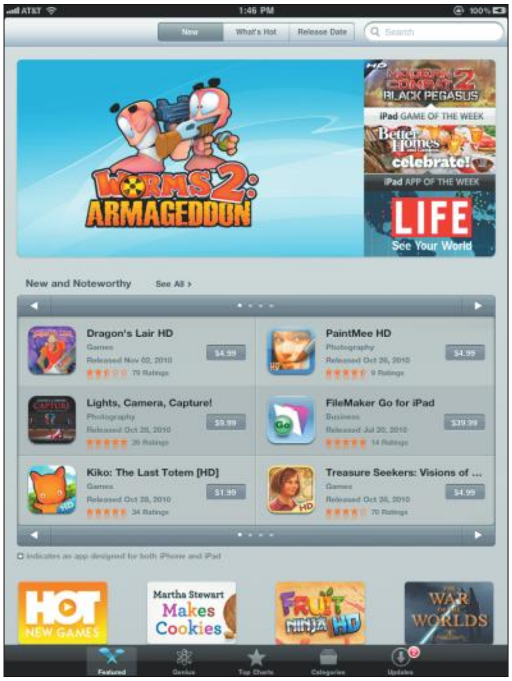


Figure 3-1: The App Store app's Featured screen.

The Top Charts screen is for those customers who need to catch up to the early adopters and only have time to look at the most popular apps of all time. Your app will not reach these lists unless you've engaged in a successful marketing strategy.

Of course, everything you do to show off your app — describing its functionality and showing screen shots of the app in action — will impact a customer's initial perception of your app. You should write your description with a focus on what makes the functionality or design of your app unique, and leave out the price information (which will differ from country to country anyway). Use engaging screen shots that demonstrate the special capabilities of your app. Don't use words like "sale," "lite," or "free" in your app icon. And make sure you have a functioning Web site with content about and support for your app, because you supply a link to it on your app's page in the App Store.

Some customers will take the time to tap the Search entry field in the upper-right corner (refer to Figure 3-1) to bring up the onscreen keyboard and search the store. As they type a keyword you assigned to your new app, or something close to its name, your app should pop up right away as a suggestion. It's therefore extremely important to use an appropriate name for your app (with terms that people might search for) and to assign appropriate keywords, as shown in Chapter 6.

TIP

Many developers choose to develop a free version of an app in order to draw attention to the paid version. Free apps are more likely to be downloaded because, well, they're free. And according to AdMob, upgrading from the free version was the top reason given when iPhone and iPod touch users were asked what drives them to purchase a paid app. However, a free version of a paid app must be a fully functional app — see "Publishing free and paid versions" in this section for details.

The In App Purchase feature offers users the opportunity to buy other apps, merchandise, game levels, premium features, e-books, and so on from right within the app. (See the "Deploying the In App Purchase Feature" section, later in this chapter, for more.) You may also want to consider offering your customers an incentive, such as free deals through the In App Purchase feature, if they tell their friends about your app. Anyone browsing the App Store can choose Tell a Friend from the Buy App pop-up on the app's information screen in the store to send the app information in an e-mail.

Besides getting your app listed in the App Store's lists, there's no way through Apple to reach potential customers. You need to consider all methods of reaching customers, and you need to price your app according to what your target customer expects, which is a primary topic of the section "Marketing 101: Pricing your app."

Marketing 101: Pricing your app

The literature about marketing could probably fill all the Trump Towers in the world, but if you want to learn about marketing quickly, there are at least two iPhone apps for that. Marketing Master and MarketingProfs, both free in the App Store, walk you through the basic concepts, and even though you certainly could do better by enrolling at Wharton (where the first Marketing 101 course was taught in 1909), it's a place to start.

Marketing is setting up a strong bait attraction system that generates leads, sorts those leads into qualified prospects, and then turns those prospects into customers. Besides fishing for the right prospects, you have to convince them to buy your app — in other words, "ABC, *always be closing*" (as the Alec Baldwin character so succinctly put it in the movie *Glengarry Glen Ross*).

One of the biggest lessons of Marketing 101 is to determine your target audience for your product. Assemble as much information about your target customer as possible — demographics, education, income level, and so on — because this information will influence all your marketing decisions, from the text you write in your descriptions and ads to the channels you use to distribute your message.

Another big lesson is to determine the cost of acquiring new customers. The simple math here is to divide all the dollars you spend in marketing per month by all the new dollars you receive each month in sales. When you know this, then you need to figure out how much you *should* be spending. To figure that out, you need to know how much your customers are worth to you — the *lifetime value* of your customer. The secret to increasing the lifetime value of your customer is to increase the quality of the customer experience, thereby encouraging repeat business. You're not in the app game to do just one app for the iPad; you need to develop more apps for the iPad (and possibly apps for the iPhone and iPod touch) and build a customer base that will be happy to buy them.

Although it's too early to predict iPad customer behavior, keep in mind that iPhone users download approximately ten new apps a month, according to AdMob, and those who regularly download paid apps spend approximately $9 on an average of five paid downloads per month. You need to attract the right people, not just anyone — potential customers are those who will understand the value of your app (also known as the *value proposition,* otherwise known as "what's in it for me?").

But at what price? Much has been written about iPhone and iPod touch app pricing strategies, and these theories haven't changed much with the iPad. At the beginning of the iPhone gold rush, pricing an app at $0.99 helped to get the app into the Top 100. But now, with hundreds of thousands of iPhone apps that already run on the iPad, and iPhone developers scrambling to design iPad versions, that's no longer true. All good marketers know that price is never a good selling point; anyone can come along and be cheaper. A better approach is to determine the true value of the app. People will pay for quality — and as more business apps become available, their prices will likely reflect their value.

The best approach is to check out similar apps, especially competing ones (if any). Remember how costly it is to acquire customers. Starting at a higher price gives you some room to offer discounted prices at different times, such as the Black Friday and Cyber Monday that follow Thanksgiving, or the start of the annual Apple Developer Conference.

Publishing free and paid versions

It's tempting to publish a free, slightly crippled version of your app that upsells the customer to the paid version. But in so many cases in the world of desktop computing, crippled free apps have been annoying and even misleading. It's not that Apple doesn't want you to promote your apps through your other apps — Apple just wants you to do it a certain way.

The free version of your paid app must be a fully functional app. A free version can't appear to be crippled, with visually disabled buttons or sections. You also can't display the price of the paid version inside the free app. A free app can have fewer features than the paid version, but the free version must be a complete app in its own right, and you can't badger the free app's users with reminders to upgrade to the paid version. Tricks like these will get your free app rejected. Also, you must use a different icon for the free and paid versions of your app, so that consumers can easily distinguish them.

These limitations should not prevent you from deploying a free app that gently enables users to discover the paid version. Also, consider deploying a basic free app that offers paid levels or feature sets through the In App Purchase feature. (See the section "Deploying the In App Purchase Feature," later in this chapter.) And you can always take advantage of Apple's iAds for Developers program and buy ads to promote your apps. (See the "Putting iAds in Your App" section, later in this chapter.)

Knowing Your Customers

One of the biggest problems facing the iPad, iPod touch, and iPhone app marketer is that the App Store doesn't tell you *who* your customers are. Sure, you know how many customers you have, and you also know from which countries, and how many of them have updated your app (if you provided an update). You even know how much they spent. What you don't know, however, can hurt you. How can you possibly build relationships with customers you don't know?

The vast majority of iPhone and iPod touch apps downloaded from the App Store are in use by less than 5 percent of users a month after downloading, according to Pinch Media (now merged with Flurry Analytics at `www.flurry.com`). Just 20 percent of users return to run a free application one day after downloading. As time goes by, that decline in usage continues, eventually settling below 5 percent after one month and nearing 0 percent after three months.

TIP

Category matters, too — games are used for longer periods than any other genre. Pinch Media found the long-term audience for the average app is just 1 percent of the total number of downloads.

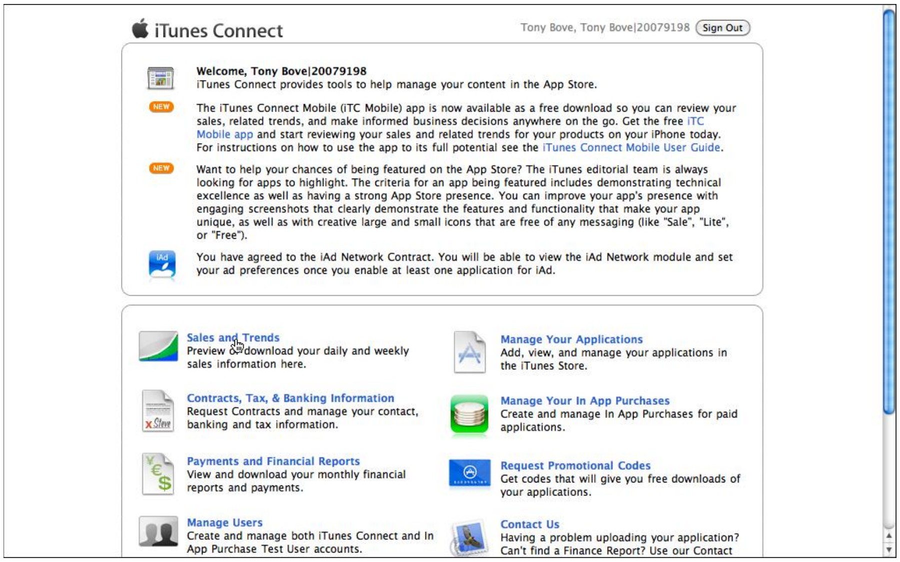
So customer loyalty is hard to build. It's difficult to determine whether a user's positive experience with your app will translate into sales of your next app or your more expensive desktop app. There are no guarantees. You need to get as much data about your customers as you can find.

You may want to add a link to a Web page that offers an optional customer registration process. You could then ask questions during this process to get more information about your customer. You probably need to offer some kind of incentive to get your customers to register, such as credit toward an in-app purchase, or an exclusive service — for example, in my app Tony's Tips for iPhone Users, I offer registered customers access to a support forum in which they can ask me specific questions about using the iPhone.

Tracking downloads

You use iTunes Connect, described in Chapter 6, to submit apps to the App Store and manage apps in the store. Apple releases daily sales data about your app in iTunes Connect, which you can view online (or download as reports), with details on how many were sold and in which country, and your profit.

To find the information, first follow the instructions in Chapter 4 to register as an Apple developer and join the iOS Developer Program. After you've built and submitted your app and it is approved and selling in the store, log in to iTunes Connect (as explained in Chapter 6) and click the Sales and Trends link, as shown in Figure 3-2.



Welcome, Tony Bove|20079198
iTunes Connect provides tools to help manage your content in the App Store.

The iTunes Connect Mobile (iTC Mobile) app is now available as a free download so you can review your sales, related trends, and make informed business decisions anywhere on the go. Get the free iTC Mobile app and start reviewing your sales and related trends for your products on your iPhone today. For instructions on how to use the app to its full potential see the iTunes Connect Mobile User Guide.

Want to help your chances of being featured on the App Store? The iTunes editorial team is always looking for apps to highlight. The criteria for an app being featured includes demonstrating technical excellence as well as having a strong App Store presence. You can improve your app's presence with engaging screenshots that clearly demonstrate the features and functionality that make your app unique, as well as with creative large and small icons that are free of any messaging (like "Sale", "Lite", or "Free").

You have agreed to the iAd Network Contract. You will be able to view the iAd Network module and set your ad preferences once you enable at least one application for iAd.

Sales and Trends
Preview & download your daily and weekly sales information here.

Manage Your Applications
Add, view, and manage your applications in the iTunes Store.

Contracts, Tax, & Banking Information
Request Contracts and manage your contact, banking and tax information.

Manage Your In App Purchases
Create and manage In App Purchases for paid applications.

Payments and Financial Reports
View and download your monthly financial reports and payments.

Request Promotional Codes
Get codes that will give you free downloads of your applications.

Manage Users
Create and manage both iTunes Connect and In App Purchase Test User accounts.

Contact Us
Having a problem uploading your application? Can't find a Finance Report? Use our Contact

Figure 3-2: Visit iTunes Connect and click Sales and Trends.

The Sales and Trends page appears, which should look a lot like what you see in Figure 3-3 (only with better sales figures, I hope) — if not, click the Dashboard button. You can then click Daily to see a bar chart of today's sales. Click Weekly and then choose a week in the Week pop-up menu to see sales for that week. If you offer free apps or use In App Purchase, click those options on the right side to see the sales trend for those (or Updates to see app updates). The Sales Trend page also lists your top paid apps and the top markets where each of your apps sells.

To download reports, click the Sales button (next to the Dashboard button). You can then choose Daily or Weekly (and choose a week in the Week pop-up menu) and click the Download button, as shown in Figure 3-4. You can import these reports into any spreadsheet program, like Excel or iWorks Numbers.

Want to check your iTunes Connect sales on the go? Download the iTunes Connect app for your iPhone or iPod touch. (You don't need an app for the iPad because the iPad can already display the full iTunes Connect screen in Safari.) Tap Summary in the row of buttons along the bottom of the screen to see a summary of all paid and free apps, In App Purchases, and updates. Tap Markets, and then tap Sales to see sales charts, or tap Updates to see progress with your app updates, segmented into different markets.

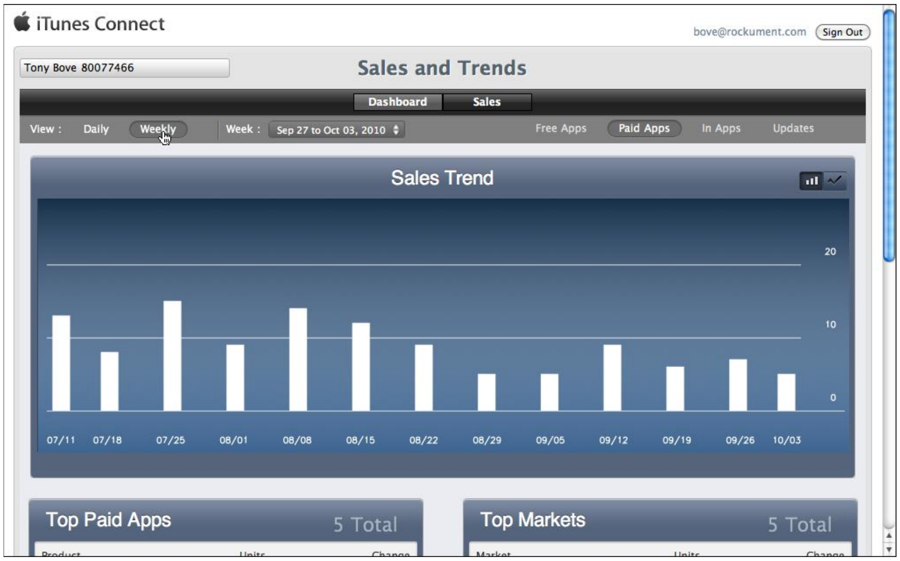
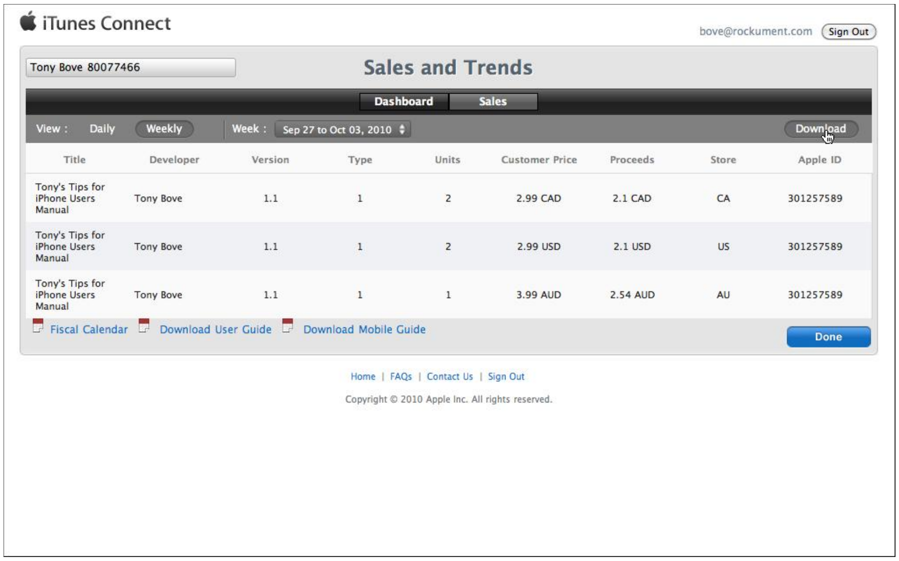


Figure 3-3: Click Dashboard to see bar charts of your sales for the day or week.



Title	Developer	Version	Type	Units	Customer Price	Proceeds	Store	Apple ID
Tony's Tips for iPhone Users Manual	Tony Bove	1.1	1	2	2.99 CAD	2.1 CAD	CA	301257589
Tony's Tips for iPhone Users Manual	Tony Bove	1.1	1	2	2.99 USD	2.1 USD	US	301257589
Tony's Tips for iPhone Users Manual	Tony Bove	1.1	1	1	3.99 AUD	2.54 AUD	AU	301257589

Fiscal Calendar
Download User Guide
Download Mobile Guide
Done
Home | FAQs | Contact Us | Sign Out
Copyright © 2010 Apple Inc. All rights reserved.

Figure 3-4: Choose Daily or Weekly reports to download.

TIP

Some savvy developers out there have come up with a number of desktop applications that have been designed to download and graph the iTunes Connect sales data for you. For example, AppViz (`www.ideaswarm.com/products/appviz`) is a Mac application that can import the reports from the Web or from a downloaded file, and it displays charts of your daily, weekly, and monthly sales. appFigures (`www.appfigures.com`) is a Web-based solution for tracking app sales, and it can download and graph your reports from iTunes Connect.

Adding analytical code to your app

There are several analytics options for iPad, iPod touch, and iPhone apps if you're willing to compile the necessary code into your app.

For example, Flurry Analytics (`www.flurry.com`), formerly Pinch Analytics, is used in thousands of popular apps because it can track any action anywhere in your app. Armed with this information, you can fine-tune the user experience in your updates and offer new features to try to catch usage drop-off as early as possible and retain more customers. You can also measure all types of revenue, from paid downloads and subscriptions to advertising and in-app purchases.

AdMob, now part of Google, offers AdMob Analytics (`http://analytics.admob.com`), a service that works with your Web site to track customers that access pages on the site through your app. All you have to do is install a code snippet onto each page you want to analyze, and AdMob does the rest. When your app requests a page from your site, your server passes analytics-related data to AdMob, which processes your data and makes it available on its site. AdMob can track the number of unique visitors and pages consumed on your site, and it can monitor user engagement metrics such as the length and depth of each visit.

Deploying the In App Purchase Feature

Apple offers the In App Purchase feature, which developers can deploy in their apps to give their users the ability to purchase virtual items directly from inside the app. If you're developing a game app with multiple levels or environments, or virtual property, consider adding In App Purchase to your app to sell more levels, environments, or property — the Eliminate app from ngmoco:) (yes, that's the developer's username) is a good example of an iPhone app that does this.

If you're developing a specialized e-book reader, use In App Purchase to sell your specialized e-books. Even if you're developing a productivity or travel app, you can deploy In App Purchase to sell additional premium features — Magellan RoadMate for the iPhone, for example, offers spoken street names and directions, and SkyVoyager Expansion Pack (Carina Software) is available for sale within SkyVoyager as an In App Purchase.

It's important to note that In App Purchase collects only payment. It doesn't download the e-book, add the game level, or hand over the virtual property. You need to provide the additional functionality, including unlocking built-in features or downloading content from your servers.

You put the In App Purchase store directly in your app using the Store Kit framework. (For more about frameworks, see Chapter 7.) The Store Kit framework connects to the App Store on your app's behalf to securely process the user's payments.

You use iTunes Connect to set up your products the same way you set up new apps. In App Purchase supports four types of products:

- **Content:** You can offer game levels, virtual property, and characters; digital books and magazines; photos and artwork; in short, any content that can be delivered *within* your app.
- **Functionality:** You can unlock or expand features you've already delivered in your app, such as a game that offers multiple smaller games for purchase. This is one alternative to publishing a free app in order to promote a paid app — publish a free app with paid extra features.

- **Services:** You can charge users for a one-time service, such as voice transcription — each time the service is used, In App Purchase processes it as a separate purchase.
- **Subscriptions:** You can provide access to content or services on a subscription basis, such as a finance magazine or an online game portal. You're responsible for tracking subscription expirations and renewal billing — the App Store doesn't send out renewal notices for you.

Although the In App Purchase feature provides a general mechanism for creating products, everything else is up to you. You can't sell real-world goods and services; you can sell only digital content, functionality, services, or subscriptions that work *within* your app. No intermediary currency is allowed (such as a virtual world's currency), and you can't include real gambling (although simulated gambling is okay). And it goes without saying that pornography, hate speech, and defamation are not allowed.

In App Purchase divides the responsibilities of selling products between your app and the App Store, handling only the payment portion. Here's how it works: Your app retrieves the list of product identifiers (set up with iTunes Connect) from its *bundle.* (You find out more about adding a bundle to your app in Chapter 16.) The app sends a request to the App Store for localized information about the products. Your app then displays this information in a store format, so that users can purchase items. When a user elects to purchase an item, your app calls Store Kit to collect payment. Store Kit prompts the user to authorize the payment and then notifies your app to provide the items the user purchased.

You can provide the content within your app binary (the file you submit to the App Store that contains your app, as I show in Chapter 6), and enable it when the user makes a purchase. Or, you can download the content from your servers for use by your app when the user makes a purchase.

Be sure the purchased items are available in all instances of your app running on all the devices the user owns (iPads, iPhones, and iPod touch models), even after your app is deleted from a device, reinstalled, or downloaded to a new device. To restore purchased items on a new device or after your app is reinstalled using the Store Kit framework, your app calls the payment queue's method for restoring completed transactions. A transaction will be created and delivered for each already purchased item — your app can process them as if they were new payment requests. The only exceptions are consumable items that are used up or disappear after use (and can never be reused). Examples of consumable items include virtual poker chips, in-game ammunition, or virtual supplies such as construction materials. If you offer such items, you need to mark them as consumable when you submit them via iTunes Connect. It is "vitally important" (in Apple's words) that you describe the transient nature of these items in your item's description. The Store Kit does not restore consumable items.

The details of the In App Purchase process is spelled out (in more detail than I can go into here) in the In App Purchase Programming Guide, which you can find in the iOS Dev Center — see Chapter 4 for instructions on registering as a developer and exploring the iOS Dev Center.

Putting iAds in Your App

Free apps can still generate revenue. To put your free app to work, you need to *monetize* the app with advertisements. The iAd Network offers you a source of revenue — Apple sells and serves the ads that appear in your app, and you receive 60 percent of the advertising revenue generated. You can also exclude ads from competitors or other unwanted advertisers based on specific keywords, URLs, and application Apple IDs.

The iAd Network provides an automated ad exchange for you to easily incorporate iAd rich media ads from advertisers into your app, monitor their performance, and track revenue. The iAd Network is not the only one; you can also place ads from AdMob (`www.admob.com`, now part of Google) or Mobclix (`www.mobclix.com`) in your apps.

Ad exchanges act as online marketplaces for buying and selling advertising impressions. Developers can earn income by "renting" space in their apps (known as *inventory*) in an auction for advertisers, ad networks, and agencies. The latter can maximize their click-through rates by bidding on precisely targeted audience segments. Thus, the more you know about your own customers, the more ads you can get for your app that are precisely targeted for more clicks (and therefore, more income).

To deploy iAds in your app, you first have to join the iAd Network by logging into iTunes Connect (as described in Chapter 6). Click through the Developer Advertising Services Agreement in the Contracts, Tax, and Banking Information section. If you don't already publish a paid app in the App Store, you will be asked to set up your banking and tax information. (See Chapter 6 for details on that as well.)

You can then enable your app for iAd rich media ads in the Manage Your Applications section within iTunes Connect, and set up your preferences in the iAd Network section to exclude competitors or advertisers based on keywords, URLs, and application Apple IDs.

iAd rich media ads are launched when a user taps on a dedicated section within your app. The iAd Programming Guide, available from the iOS Dev Center, describes how to dedicate a portion of your screen to display an ad and how to change the banner size and orientation, and it explains how your app should respond when a banner ad is touched. The iAd Framework

Reference provides a list of tasks, methods, and protocols to use when developing your app to deliver iAd rich media ads.

You can also find more technical information about configuring your app to offer iAd rich media ads in the iOS Application Programming Guide and the View Controller Programming Guide for iOS.

Links Are Not Enough

It goes without saying that you have a Web page (or an entire site) devoted to your app, and you've outfitted your site with keywords for search engine optimization so that searches in Google result in your Web page appearing on or near the first search page. You also use Google Analytics to measure traffic. Reams have been written on this topic. (See Pedro Sostre and Jennifer LeClaire's *Web Analytics For Dummies* for one particularly good use of such paper reams.)

When promoting an app, use well-written copy, good screen shots, quotes from user reviews, and third-party recommendations. If you have the skills or the budget, develop a quick video, upload it to YouTube, and put that on your page.

TIP

Don't forget to display prominently on your Web page the Apple-legal App Store badge that links visitors to the App Store on iTunes. You can find the App Store Badging and Artwork page by clicking the Marketing Resources link in the iOS Dev Center under App Store Resource Center in the right column — see Chapter 4 for instructions on registering as a developer and exploring the iOS Dev Center.

But Web page links are not enough. This ecosystem (of iTunes, the App Store, the iPad, the iPhone, and the iPod touch) offers more than a few methods of reaching potential customers, as discussed in the following sections.

Using iTunes affiliate links

Your App Store links should make you some spare change as well as tell you a few things about your customers. The iTunes affiliate program gives you links to put on your Web pages. When a visitor clicks this link and then buys something in the iTunes Store (including the App Store), you get 5 percent. Although that's not much, it doesn't hurt. You can add affiliate links to *any* apps (or songs or videos) in the store, not just your apps.

You can put an affiliate link on your blog, on your friends' Web pages, and even in the signature of your e-mails. Anywhere that you would normally link to your app in the App Store, replace it with your affiliate link.

Another good reason to do this is to obtain more data. You can find out how often visitors see your link, what percentage actually clicks on your link, and where they come from. Apple uses LinkShare (`www.linkshare.com`), a fairly popular affiliate manager. LinkShare also manages affiliate programs for AT&T, LEGO, Macys.com, TigerDirect.com, and hundreds of other companies.

Making use of user reviews

Users are your friends, even when they're bashing you in public.

The App Store customer review is one of the most valuable tools you have to convince potential customers to buy your app. Only people who have purchased your app can write a review. If you offer your users an optional registration on a Web site or by e-mail (using incentives such as insider news, discounts, or free stuff), you can use that opportunity to remind them to write a review of the app in the App Store.

Even harsh reviews can be helpful, pointing out bugs that you may have not previously uncovered or offering ideas for additional features and functions you didn't think of. You should use this information to prioritize your development activities for future updates, and you can add information about fixed bugs in the app's description when you submit the update.

Going social

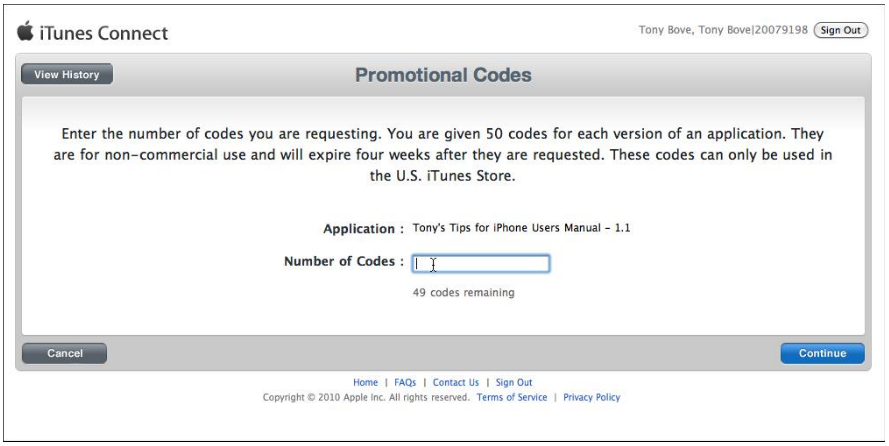
Social networking spreads the buzz about your app. One of the most popular techniques is to publicize your app on dozens of forums, including the iPhone Blog Forum (`http://forum.tipb.com`), MacRumors Forums (`http://forums.macrumors.com`), or iPhone Owners (`www.iphoneowners.com`), most of which cover iPad apps as well as iPhone and iPod touch apps. New forums are springing up every week.

Spreading buzz is a time-consuming job. Developers often turn to professional PR agencies that can put out press releases and work the blogs and forums for you. A good PR blast can drive thousands of sales within a few days. But beware: Sales can fall off a cliff as new stories replace the old ones.

You should submit a press release about your app to the blogs and publications that directly serve your customers. You may not get attention for a paid app without also including a promotional code so that the reviewer can download the app for free. As of this writing, Apple gives you 50 promotional codes for each version of an app; use them wisely because there are far more than 50 general review blogs for iPad apps, and there may be thousands of other blogs that serve your potential customers, such as travel blogs for customers of a travel app.

Remember that each promotional code you request expires four weeks after you requested it, so request only the number of codes you need at the moment. After you've submitted your app's information and promotional code to a few blogs, go back and request more. These codes can be used only in the U.S. iTunes Store.

To get your promotional codes, visit iTunes Connect and click the Request Promotional Codes link. (Refer to Figure 3-2.) Then type the number of codes you need, as shown in Figure 3-5, and click Continue. iTunes Connect then provides the promotional codes to send in your e-mail or blog request. Reviewers already know how to enter promotional codes into the iTunes Store before buying an app.


iTunes Connect
Tony Bove, Tony Bove|20079198
Sign Out
View History
Promotional Codes
Cancel
Continue
Home | FAQs | Contact Us | Sign Out
Copyright © 2010 Apple Inc. All rights reserved. Terms of Service | Privacy Policy

Enter the number of codes you are requesting. You are given 50 codes for each version of an application. They are for non-commercial use and will expire four weeks after they are requested. These codes can only be used in the U.S. iTunes Store.

Application : Tony's Tips for iPhone Users Manual – 1.1

Number of Codes :

49 codes remaining

Figure 3-5: Request promotional codes to give your app away to reviewers.

Buying advertising

Generating buzz through advertising is a time-honored tradition in marketing, dating back to ancient times when Egyptians used papyrus to make sales messages and wall posters and when Roman emperors advertised military victories and public works on coins.

The coins are a good example: They were mobile, the image appeared often (at every transaction) to establish the "brand" of the emperor, and they cross-promoted other victories and public works.

Branding is a topic covered in grandiose detail in enough books to fill at least one Trump Tower. (Yup, there's even *Branding For Dummies.*) Companies with very recognizable brands tend to make free apps to promote the brand.

You may want to consider creating a version of your app that you could license or sell to a client company that then puts its recognizable brand on it. Such an arrangement is called a *white label* deal because the client company supplies the brand on the label.

If you're publishing more than one paid app, the first place to advertise your newest app is in your older apps — add links to cross-promote your other paid app. It costs nothing and helps to build customer loyalty, just because the customer can see that you've developed other apps. You would typically use an "About" section in your app for these links, so that they are not obtrusive. Don't display inactive or crippled functions in a free app that promotes a paid version of the app — the free app won't make it through the App Store approval process.

Consider *buying* ads through Apple's iAd for Developers Program, which was set up for developers to advertise their apps in other apps using rich media iAds. With iAd for Developers, users can download advertised apps from the App Store without leaving the app they're in. As the advertiser, you don't have many choices for customizing the targeting of your apps (as of this writing) — but iAd for Developers will optimize your campaign to ensure the right audience is viewing and interacting with your ads.

The iAds are rich media experiences within the app that act like mini Web applications. You can use any standard Web technology to create an ad that works in Safari on iOS, but you should use Apple's iAd JS and the Native Bindings libraries that are specifically designed for iOS. iAd JS is a JavaScript library based on WebKit that provides the foundation to create ads that use audio, the Multi-Touch interface, WebKit animations, and HTML5 video. iAd JS provides a declarative interface (the interface can be described using an XML document, rather than built using code), as well as other mechanisms to load your iAds quickly on iOS. The Native Bindings library gives you access to some Apple application functions that enable your iAd to integrate with apps. For more details, consult the iAd JS Ad Creation Guide in the iAd JS Reference Library. (Click the iAd JS Reference Library link in the iOS Dev Center to get there.)

You can also buy ads on other mobile networks that offer ads in apps. AdMob and Mobclix offer different ways to precisely target your ads. AdMob, for example, offers a video ad unit that runs a dedicated video player inside the app. The app's users can engage with interactive campaigns without leaving the video player. As the advertiser, you can also set up action buttons that let the app's users share video content with friends and connect to social networking sites — again, without ever leaving the video player. As an advertiser, you have a choice of auto-play or click-to-play: The former plays your video ads as soon as the app loads, whereas the latter requires the app's users to tap your banner in order to engage with the campaign.

Another popular choice is Google AdWords. You can reach anyone that searches on Google or on partner networks using any browser. There are close to a google of books available on this topic. (Well, almost 100; try *Google AdWords For Dummies* by Howie Jacobson.)

Getting publicity

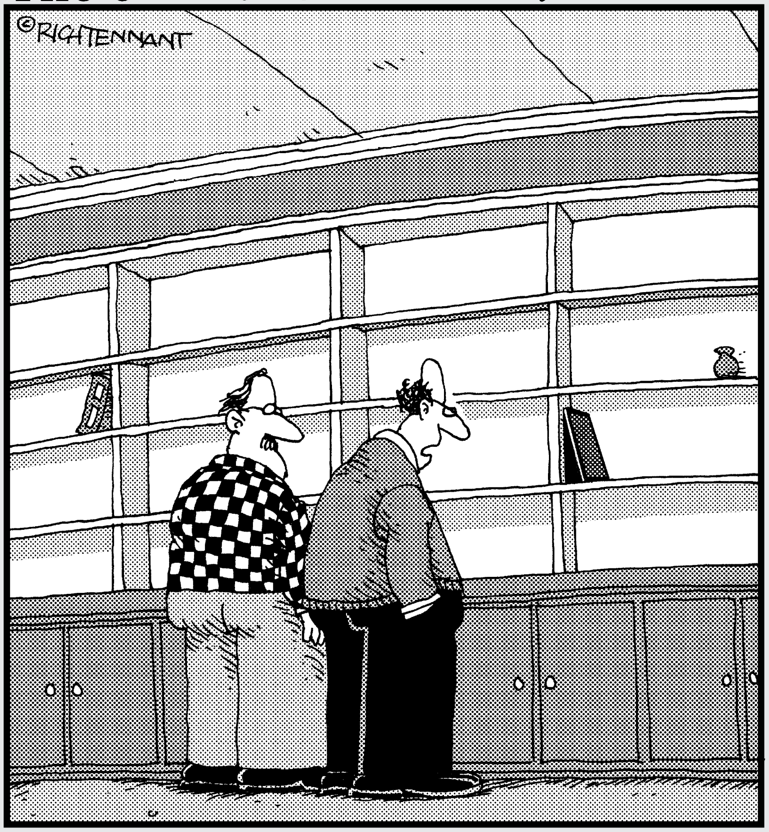
Publicity offers the biggest payoff in the short term, and the best way to get it is to pay an excellent PR firm. Good publicity can create a spike in sales that could be misleading, but if you've implemented other marketing campaigns to take advantage of it, sales could level out at a much higher rate than before the publicity hit. The best of the PR firms can help you with your entire marketing strategy.

But if you can't afford that . . . publicity stunts work well if received well by the public. Some of world's most beloved annual events began their existence as cheap publicity stunts. In 1903, publisher Henri Desgrange started a bicycle road race as a publicity stunt to promote his newspaper, never imagining that the *Tour de France* would be going strong more than 100 years later. The Rose Bowl grew out of an 1890 stunt designed to promote Pasadena, California; the Miss America pageant began in 1921 as a publicity stunt to lure tourists to Atlantic City after Labor Day; and the Academy Awards began in 1929 as a cheap publicity stunt for the movie industry. As Lenny Bruce put it, "Publicity is stronger than sanity: Given the right PR, armpit hair on female singers could become a national fetish." (It did, about 15 years later.)

If you can generate publicity, be sure to have a demo on hand — something to titillate people whether they have their iPads in hand or not. Create a video on YouTube and link it to your press release. Offer your app at a lower price for a limited time period at the start of a publicity campaign. Leave no stone unturned when looking for promotional opportunities as part of the campaign. And make sure your demo works — a sacrifice to the demo gods can't hurt. Or just keep repeating the mantra from the patron prophet of demos, Demosthenes: "Small opportunities are often the beginning of great enterprises."

Part II
Becoming a Real Developer

The 5th Wave By Rich Tennant



"I build bookshelves and Bernice buys an iPad."

In this part . . .

You can work at home alone, but it takes a village to develop an iPad app — the Apple developer village.

You have to register as an Apple developer if you want to get the Software Development Kit (SDK) and all the other goodies that Apple provides for developers — and of course, that means agreeing a confidentiality agreement. And if you actually want to run your application on a real iPad, you have to join the iOS developer program. This part gets you through these processes and introduces you to the SDK.

- Chapter 4 gets you into the Apple developer village. You find out how to register as a developer, join the program, explore the developer center on the Web, and download the SDK.
- Chapter 5 goes into more detail about the SDK itself. You learn all about Xcode and Interface Builder, how to start a project from a template, how to build and run an iPad app in the Simulator, and how to customize Xcode to your liking.
- Chapter 6 takes you by the hand through the excruciating process of provisioning your iPad to run your app during development, and the even more inscrutable process of setting up your iPad app for development and for submission to the App Store. I put all this murky stuff into one chapter so that you don't have to hunt all over the developer center and portal looking for it.

Chapter 4

Enlisting in the Developer Corps

In This Chapter

- Registering as a developer
- Exploring the iPhone Dev Center
- Installing the Software Development Kit (SDK)
- Looking at the whys and hows of joining the Developer Program

Benjamin Franklin's famous *Join, or Die* political cartoon of the 1760s could well be applied to Apple's role in today's mobile software industry. You can't gain independence on your own; you need the powerful movement of a large group. Apple needs developers, and developers need Apple.

For sure, you can develop your applications independently, and you can even develop for other platforms (which is the topic of other books), but many of those platforms offer immature Software Development Kits and little or no support. What's more, you could develop for a number of platforms and then watch your product die in a diffused marketplace.

Apple is clearly on a mission with the iPad, iPhone, and App Store ecosystem to change the user experience, and you *have* to join (or die). No, you won't automatically turn into an Apple fanboy (but it doesn't hurt to be one, either). You *will* be supported with a robust Software Development Kit, comprehensive information, and reliable support.

Most importantly, you *must* join if you want to develop apps for the iPad, iPod touch, or iPhone (or any combination of these). You have to follow Apple's policies and procedures. Although the developer kit you use to develop apps for the iPad, iPod touch, and iPhone — the iOS Software Development Kit (SDK) — is *free,* you have to register as an iPhone developer first. And don't forget — to run the SDK, you need an Intel-based Mac running Mac OS X Snow Leopard version 10.6.4 or a newer version.

Registering also gives you access to all the documentation and other resources in the iOS Dev Center. This whole ritual transforms you into a *Registered iOS Developer* capable of developing for the iPad, iPod touch, and iPhone.

By the time you read this, Apple may have changed the titles of the Web pages and centers. Go to `http://developer.apple.com` for a complete overview of Apple's developer centers.

Becoming a registered developer is free, but there's a catch: If you actually want to run your application on a real iPad as opposed to only on the Simulator that comes with the SDK, you have to *join the iOS Developer Program*. Fortunately, an individual membership costs only $99 as of this writing. (I should mention as well that an individual membership is required of anyone who wants to distribute his or her app using the App Store.)

Although you can register as a developer and join the iOS Developer Program all in one step (as I show in "Joining the Developer Program" in this chapter), you may want to register as an Apple developer first (after all, it's free). Then, after you get your feet wet with the SDK, you can pay the fee and join the iOS Developer Program. That's why I provide separate sections in this chapter for "Becoming a Registered Developer" and "Joining the Developer Program."

What you see when you go through this process yourself may be slightly different from what you see here. Don't panic. It's because Apple changes the site from time to time.

Becoming a Registered Developer

Although just having to register is annoying to some people, it doesn't help that the process itself can be a bit confusing. Fear not! Follow the steps, and you can safely reach the end of the road.

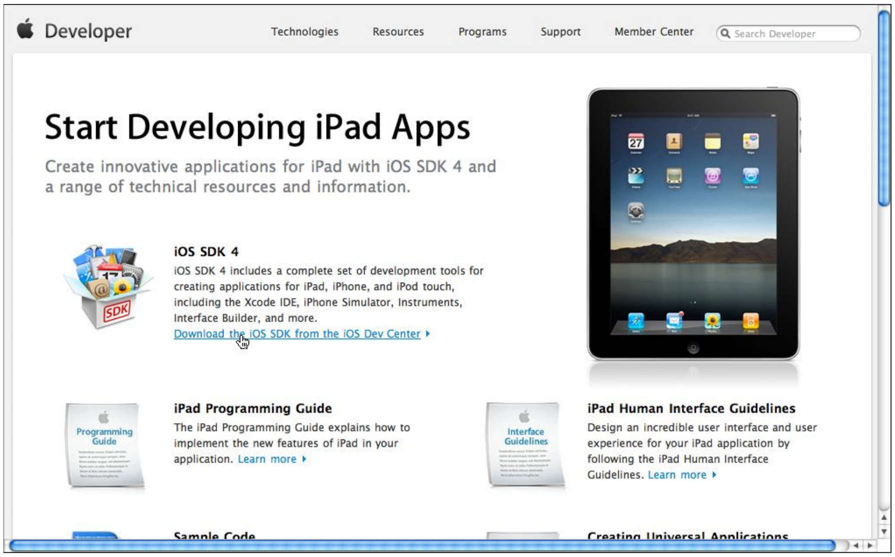
Your first stop is to become an Apple Registered Developer — if you're not already registered — and obtain your Apple ID. If you don't want to join the iOS Developer Program right away, you can register first for free, and then join later when you're ready to pay for the Program.

If you've already registered, you can skip to the next section to join the iOS Developer Program with your registered developer Apple ID and password, or you can skip ahead to "Downloading the SDK" and join the iOS Developer Program later. (You can develop software using the SDK without joining, but as you find out later, you can't run this software on your iPad until you join.)

So, without further ado, here's how to quickly become a Registered Developer:

1. **Go to the iPad section of Apple's Web site (`www.apple.com/ipad`) and click the Learn More link in the SDK section in the lower-left corner, or just point your browser to `http://developer.apple.com/ipad/sdk`.**

   Doing so brings you to a page similar to the one shown in Figure 4-1, with the SDK logo. Apple does change this site occasionally, so when you get there, it may be a little different.



Start Developing iPad Apps

Create innovative applications for iPad with iOS SDK 4 and a range of technical resources and information.

iOS SDK 4

iOS SDK 4 includes a complete set of development tools for creating applications for iPad, iPhone, and iPod touch, including the Xcode IDE, iPhone Simulator, Instruments, Interface Builder, and more.
Download the iOS SDK from the iOS Dev Center ▸

iPad Programming Guide

The iPad Programming Guide explains how to implement the new features of iPad in your application. Learn more ▸

iPad Human Interface Guidelines

Design an incredible user interface and user experience for your iPad application by following the iPad Human Interface Guidelines. Learn more ▸

Sample Code

Creating Universal Applications

Figure 4-1: Start developing iPad apps here.

2. **Click the Download the iOS SDK from the iOS Dev Center link. (See Figure 4-1.)**

   The iOS Dev Center main page appears, as shown in Figure 4-2. You may be tempted by some of the links, but they get you only so far until you log in as a registered developer.

   If you're registered already, click Log In and supply your Apple ID and password; you can then skip to the next section to join the iOS Developer Program, or skip ahead to "Downloading the SDK" and join the iOS Developer Program later.

3. **Click the Register link in the top-right corner of the screen. (See Figure 4-2.)**

   You see a page explaining why you should become a registered developer and what Apple has to offer registered developers, as shown in Figure 4-3.

4. **Click Get Started. (See Figure 4-3.)**

   A new page appears, asking whether you want to create a new Apple ID or use an existing one.

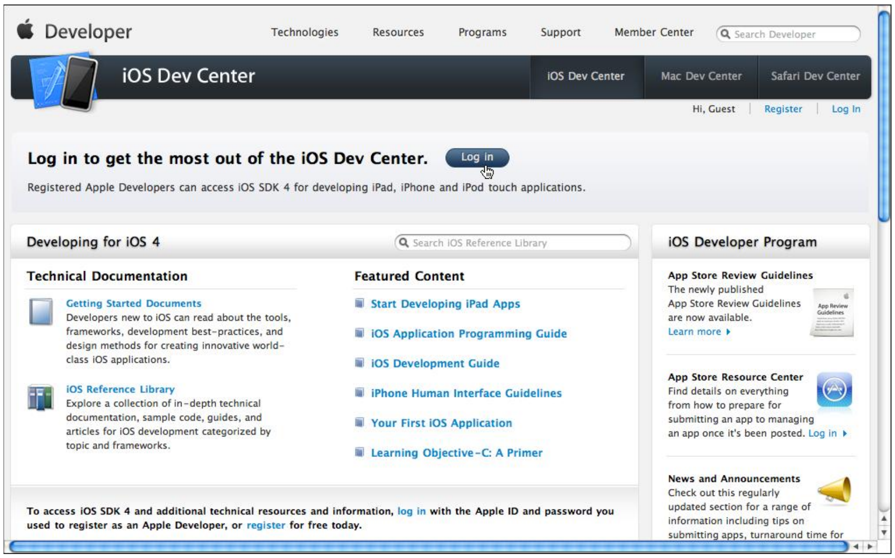
   You can use your current Apple ID (the same one you use for iTunes, MobileMe, or the Apple Store) or create a new Apple ID and then log in.

   - *If you don't have an Apple ID,* select Create an Apple ID and click Continue. You find yourself at the Complete Your Personal Profile page, where you can enter your desired Apple ID and password, and proceed to Step 5.
   - *If you already have an Apple ID,* select the Use an Existing Apple ID option and then click Continue. You're taken to a screen where you can log in with your Apple ID and password. That takes you to the Complete Your Personal Profile page with some of your information already filled out.

5. **Continue filling out the personal profile form and then click Continue.**

   If you have an Apple ID, most of the form is already filled out.

   You must fill in the country code in the phone number field. If you're living in the United States, the country code is 1.



Developer | Technologies | Resources | Programs | Support | Member Center | Search Developer

iOS Dev Center | iOS Dev Center | Mac Dev Center | Safari Dev Center

Hi, Guest | Register | Log In

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

Registered Apple Developers can access iOS SDK 4 for developing iPad, iPhone and iPod touch applications.

Developing for iOS 4 — Search iOS Reference Library

Technical Documentation

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

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

Featured Content

- Start Developing iPad Apps
- iOS Application Programming Guide
- iOS Development Guide
- iPhone Human Interface Guidelines
- Your First iOS Application
- Learning Objective-C: A Primer

To access iOS SDK 4 and additional technical resources and information, log in with the Apple ID and password you used to register as an Apple Developer, or register for free today.

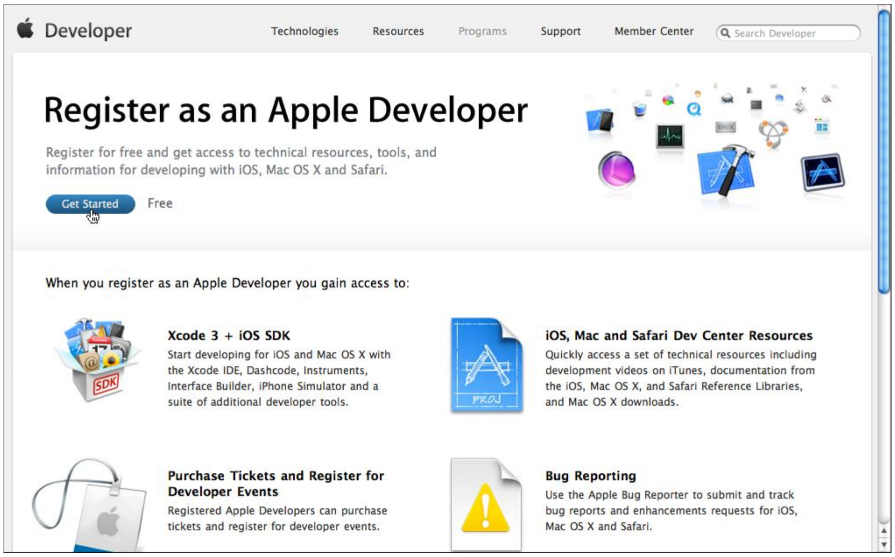
iOS Developer Program

App Store Review Guidelines
The newly published App Store Review Guidelines are now available.
Learn more

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

News and Announcements
Check out this regularly updated section for a range of information including tips on submitting apps, turnaround time for

Figure 4-2: The iOS Dev Center for developing iPad apps.



Register as an Apple Developer

Register for free and get access to technical resources, tools, and information for developing with iOS, Mac OS X and Safari.

When you register as an Apple Developer you gain access to:

Xcode 3 + iOS SDK
Start developing for iOS and Mac OS X with the Xcode IDE, Dashcode, Instruments, Interface Builder, iPhone Simulator and a suite of additional developer tools.

iOS, Mac and Safari Dev Center Resources
Quickly access a set of technical resources including development videos on iTunes, documentation from the iOS, Mac OS X, and Safari Reference Libraries, and Mac OS X downloads.

Purchase Tickets and Register for Developer Events
Registered Apple Developers can purchase tickets and register for developer events.

Bug Reporting
Use the Apple Bug Reporter to submit and track bug reports and enhancements requests for iOS, Mac OS X and Safari.

Figure 4-3: Register as an Apple Developer.

6. **Complete the next part of the form to finish your professional profile.**

   You're asked some basic business questions. After you've filled everything in and clicked the Continue button, you're taken to yet another new page, which asks you to agree to the Registered iOS Developer Agreement.

7. **Click I Agree.**

   TIP

   Don't forget to select the confirmation check box that you have read and agree to be bound by the agreement and that you're of legal age.

   If you just created your Apple ID, you're asked for the verification code sent to the e-mail address you supplied when you created your Apple ID. If you used your existing Apple ID, you're taken to Step 9.

8. **Open the e-mail from Apple, enter the verification code, and click Continue.**

   Clicking Continue takes you to a thank-you page.

9. **On the thank-you page, click the Visit iOS Dev Center button, and you're automatically logged in to the iOS Dev Center, which I describe in the "Exploring the Dev Center" section in this chapter.**

So, you're now an officially registered iPad, iPod touch, and iPhone developer, which enables you to explore the iOS Dev Center and download the SDK (as I show in "Exploring the Dev Center" in this chapter — you can jump to that section if you're not ready to join the iOS Developer Program).

However, simply registering as a developer doesn't give you the status you need to actually run your app on your own (or anyone else's) iPad, iPod touch, or iPhone, or to distribute your app through the App Store. The next section shows you how to get with the program — the iOS Developer Program.

Joining the Developer Program

The Simulator application for the Mac that comes standard with the iOS SDK is a great tool for learning to program the iPad, but it does have some limitations. It doesn't support some hardware-dependent features, and when it comes to testing, it can't really emulate such everyday iPad realities as CPU speed, memory throughput, or your actual location.

"Minor annoyances," you might say, and you might be right. But the real issue is that *just registering* as a developer doesn't get you two very important things: the ability to actually run your app on your own iPad and the right to distribute your app through the App Store. (Remember that the App Store is the only way for commercial developers to distribute their apps — even free apps — to more than a few people.)

To run your app on a real iPad or get a chance to profile your app in the App Store, you have to enroll in either the Individual or Company version of the iOS Developer Program for $99 per year. The only difference between these two versions is that the Individual program is for — you guessed it — individual developers, while the Company program is for development teams and enables team members to share development code (even if they're not in the same location). (The Enterprise version is $299 per year and gives your organization access to resources to help develop proprietary, in-house iOS applications that your organization can distribute to employees or members.)

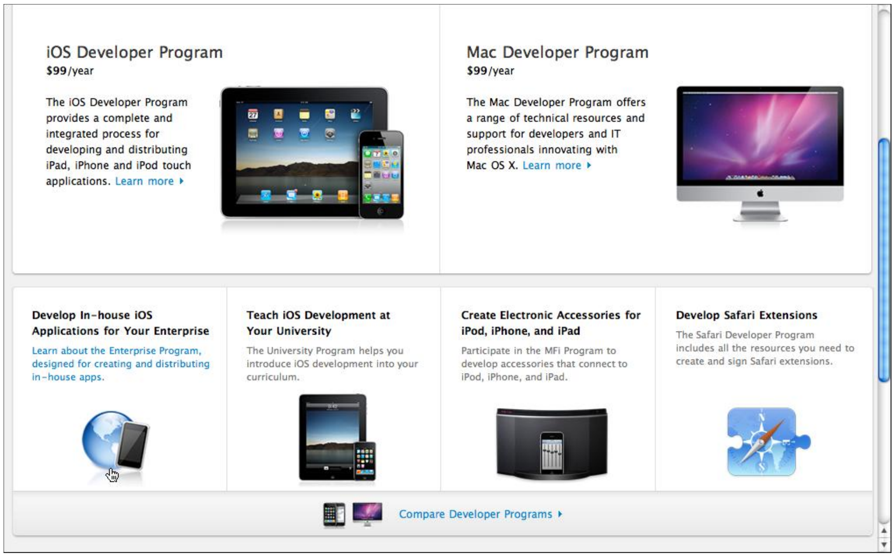
To find out more about these programs and to compare the development programs for iOS, Mac, and Safari, point your browser to `http://developer.apple.com/programs` to see the Programs page shown in Figure 4-4. To compare developer programs, click the Compare Developer Programs link at the bottom of the page to see the comparison summaries shown in Figure 4-5.

TIP

To enroll in the Enterprise version, click the Develop In-house iOS Applications for Your Enterprise link in the bottom-left corner of the page (shown in Figure 4-4).

TIP

It used to be that the membership approval process could take a while, and although the process does seem quicker these days, it's still true that you can't run your apps on your iPad until you're approved for the program. (Of course you can't submit apps to the App Store until each app is approved, but I talk about that in Chapter 6.) You should enroll as early as possible.



iOS Developer Program
$99/year

The iOS Developer Program provides a complete and integrated process for developing and distributing iPad, iPhone and iPod touch applications. Learn more ▸

Mac Developer Program
$99/year

The Mac Developer Program offers a range of technical resources and support for developers and IT professionals innovating with Mac OS X. Learn more ▸

Develop In-house iOS Applications for Your Enterprise
Learn about the Enterprise Program, designed for creating and distributing in-house apps.

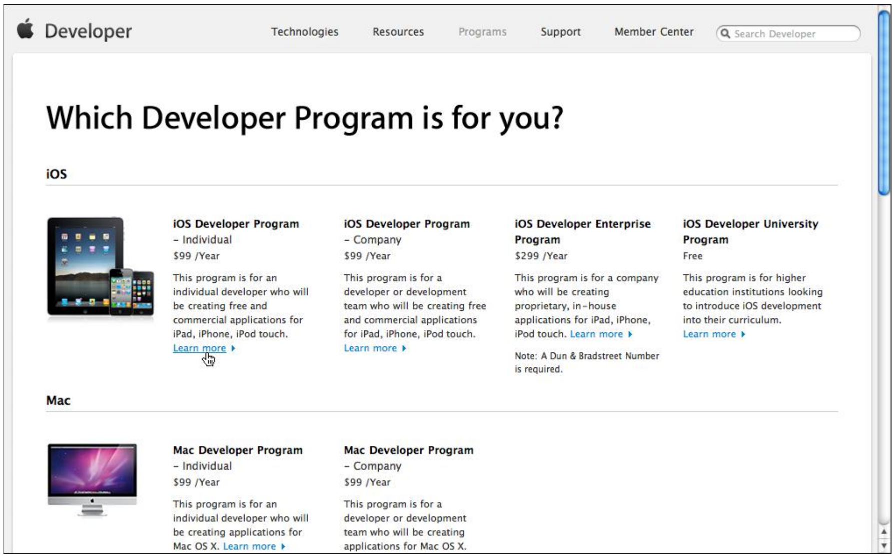
Teach iOS Development at Your University
The University Program helps you introduce iOS development into your curriculum.

Create Electronic Accessories for iPod, iPhone, and iPad
Participate in the MFi Program to develop accessories that connect to iPod, iPhone, and iPad.

Develop Safari Extensions
The Safari Developer Program includes all the resources you need to create and sign Safari extensions.

Compare Developer Programs ▸

Figure 4-4: Choosing an Apple developer programs.



Which Developer Program is for you?

iOS

iOS Developer Program – Individual
$99 /Year
This program is for an individual developer who will be creating free and commercial applications for iPad, iPhone, iPod touch.
Learn more ▸

iOS Developer Program – Company
$99 /Year
This program is for a developer or development team who will be creating free and commercial applications for iPad, iPhone, iPod touch.
Learn more ▸

iOS Developer Enterprise Program
$299 /Year
This program is for a company who will be creating proprietary, in-house applications for iPad, iPhone, iPod touch. Learn more ▸
Note: A Dun & Bradstreet Number is required.

iOS Developer University Program
Free
This program is for higher education institutions looking to introduce iOS development into their curriculum.
Learn more ▸

Mac

Mac Developer Program – Individual
$99 /Year
This program is for an individual developer who will be creating applications for Mac OS X. Learn more ▸

Mac Developer Program – Company
$99 /Year
This program is for a developer or development team who will be creating applications for Mac OS X.

Figure 4-5: Comparing Apple developer programs.

To join the iOS Developer Program (Individual or Company), follow these steps:

1. **Click the Learn More link from the comparison page shown earlier in Figure 4-5 for the Individual or Company program, or click the iOS Developer Program link on the Programs page shown earlier in Figure 4-4.**

Either link takes you to the iOS Developer Program page. (If you prefer, simply point your browser to `http://developer.apple.com/programs/ios`.)

2. **On the left side of the screen, click the Enroll Now button.**

   A new page appears with an overview of the process of joining the program, along with the technical requirement (an Intel-based Mac running Snow Leopard or newer version of OS X).

3. **Click Continue to enroll.**

   After clicking Continue, a screen appears with the option to either create a new Apple account or use an existing one.

4. **Choose an option to either create a new Apple account or use an existing one, and then click Continue.**

   Here's how to pick your option:

   - *If you already registered* (as I describe in the previous section), select the "I'm registered as a developer with Apple . . ." option from the Existing Apple Developer options on the right. Select the "I'm currently an ADC Select, Premier, or Student Member . . ." option if you are a student, ADC Select, or Premier member. (For more information about these programs, see `http://developer.apple.com/programs/adcbenefits`.) If you already joined the Mac or Safari Developer Programs, select the "I'm currently enrolled . . ." option to add the iOS Developer Program to your account.
   - *If you haven't registered yet,* choose one of the New Apple Developer options: If you have an Apple ID already (from iTunes Store or Apple Store purchases), select the "I currently have an Apple ID . . ." option. If not, select the "I need to create a new account . . ." option.

   After clicking Continue, a screen appears asking if you're enrolling as an Individual or a Company, and providing information about the Individual and Company enrollment options.

5. **Click Individual to enroll as an Individual, or click Company to enroll as a Company.**

   REMEMBER

   As of this writing, the Individual and Company programs cost $99 per year each for developing for iOS devices — you can choose one or the other. To be sure you're selecting the option that meets your needs, give the program details a once-over.

   After clicking Individual or Company, the Apple Developer Program Enrollment Personal Profile page appears if you need to continue adding personal information for an Apple account and to register as a developer — follow the steps in the previous section to register and agree to the developer agreement, and you're taken to the page for entering payment information. If you're already registered and have already agreed to the developer agreement, you go directly to the payment page.

6. **Enter your payment information and click Continue.**

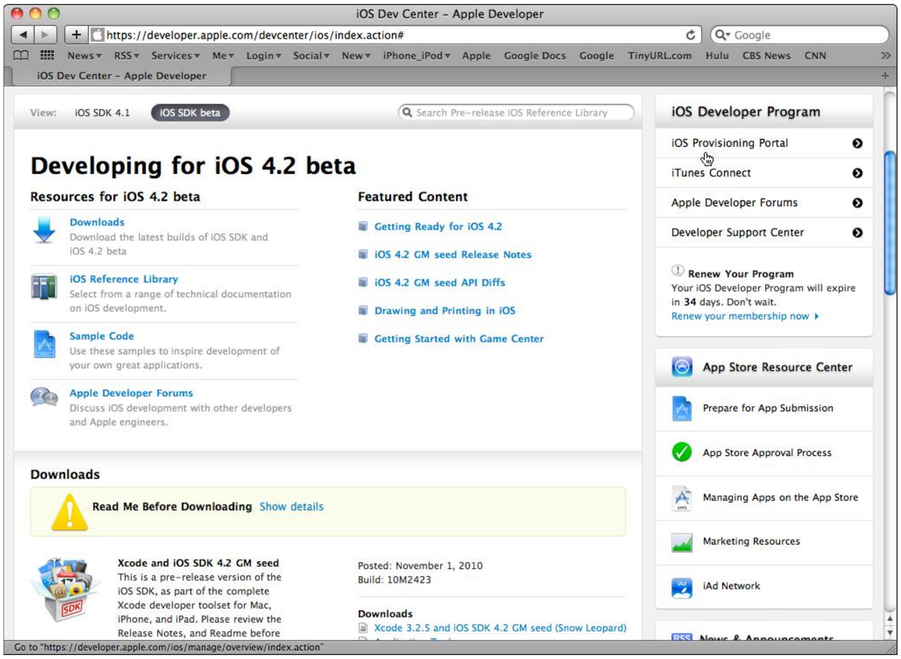
   Depending on the option you selected, you're either given the opportunity to pay (if you selected Individual) or you're asked for some more company information and then given the ability to pay. (But pay you will.)

   Although joining as an Individual is easier than joining as a Company, there are clearly some advantages to enrolling as a Company. For example, you can add team members (which I discuss in connection with the Provisioning Portal in Chapter 6), and your company name appears in your listing in the App Store.

   When you join as an Individual, your real name shows up when the user buys (or downloads for free) your app in the App Store. If you're concerned about privacy, or if you want to seem "bigger," the extra work involved in signing up as a Company may be worthwhile for you.

7. **Continue through the process, and eventually you will be accepted in the Developer Program of your choice.**

After acceptance, you can log in to the iOS Dev Center as an Official iOS Developer and see the page shown in Figure 4-6.


iOS Dev Center – Apple Developer

https://developer.apple.com/devcenter/ios/index.action#

View: iOS SDK 4.1 | iOS SDK beta

Developing for iOS 4.2 beta

Resources for iOS 4.2 beta

Downloads
Download the latest builds of iOS SDK and iOS 4.2 beta

iOS Reference Library
Select from a range of technical documentation on iOS development.

Sample Code
Use these samples to inspire development of your own great applications.

Apple Developer Forums
Discuss iOS development with other developers and Apple engineers.

Featured Content
- Getting Ready for iOS 4.2
- iOS 4.2 GM seed Release Notes
- iOS 4.2 GM seed API Diffs
- Drawing and Printing in iOS
- Getting Started with Game Center

iOS Developer Program
- iOS Provisioning Portal
- iTunes Connect
- Apple Developer Forums
- Developer Support Center

Renew Your Program
Your iOS Developer Program will expire in 34 days. Don't wait.
Renew your membership now

App Store Resource Center
- Prepare for App Submission
- App Store Approval Process
- Managing Apps on the App Store
- Marketing Resources
- iAd Network

Downloads

Read Me Before Downloading Show details

Xcode and iOS SDK 4.2 GM seed
This is a pre-release version of the iOS SDK, as part of the complete Xcode developer toolset for Mac, iPhone, and iPad. Please review the Release Notes, and Readme before

Posted: November 1, 2010
Build: 10M2423

Downloads
Xcode 3.2.5 and iOS SDK 4.2 GM seed (Snow Leopard)

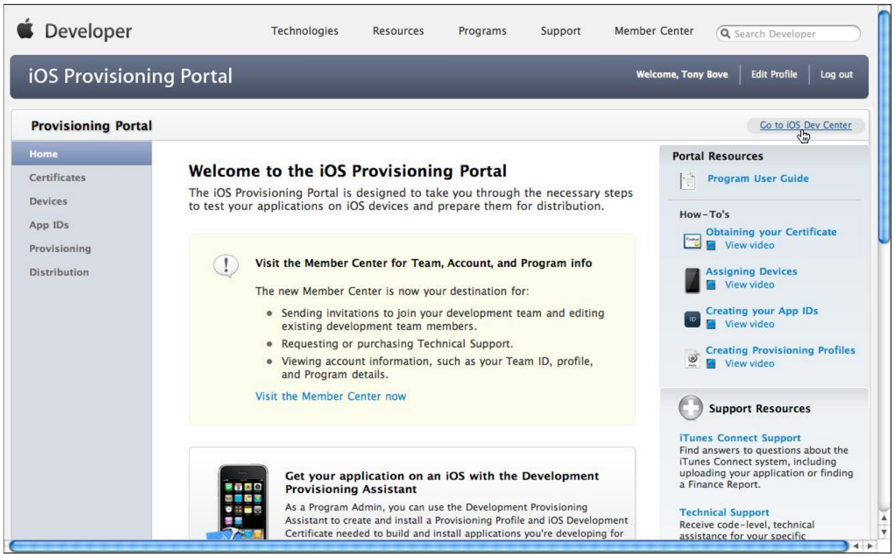
Go to "https://developer.apple.com/ios/manage/overview/index.action"

Figure 4-6: The iOS Dev Center with resources and downloads.

If you click the iOS Provisioning Portal link in the right column (refer to Figure 4-6), you see all sorts of things you can do as a developer in the portal, which is shown in Figure 4-7.

TIP

You shouldn't linger too long in the iOS Provisioning Portal, simply because it can be really confusing unless you understand the process. Click the Go To iOS Dev Center link in the upper-right corner of the page (refer to Figure 4-7) to go back to the iOS Dev Center. In Chapter 6, I explain the iOS Provisioning Portal, which lets you provision your device, run your application on it, and prepare your creation for distribution to the App Store.



Welcome to the iOS Provisioning Portal

The iOS Provisioning Portal is designed to take you through the necessary steps to test your applications on iOS devices and prepare them for distribution.

Visit the Member Center for Team, Account, and Program info

The new Member Center is now your destination for:

- Sending invitations to join your development team and editing existing development team members.
- Requesting or purchasing Technical Support.
- Viewing account information, such as your Team ID, profile, and Program details.

Visit the Member Center now

Get your application on an iOS with the Development Provisioning Assistant

As a Program Admin, you can use the Development Provisioning Assistant to create and install a Provisioning Profile and iOS Development Certificate needed to build and install applications you're developing for

iTunes Connect Support
Find answers to questions about the iTunes Connect system, including uploading your application or finding a Finance Report.

Technical Support
Receive code-level, technical assistance for your specific

Figure 4-7: The iOS Provisioning Portal.

Exploring the Dev Center

You can find out more about the resources available to you in the iOS Dev Center later in the section entitled "Resources in the Dev Center." However, for the moment, I want you to get prepared for what you're *really* after: the iOS SDK, which enables you to develop apps for the iPad.

The SDK offers tools for developing iPad, iPod touch, and iPhone apps. Here's a handy list of what's inside:

- **Xcode:** This refers to Apple's complete integrated development environment (IDE), which integrates all the SDK's features: the code editor, the build system, the graphical debugger, and project management. (I introduce you to the code editor's features in more detail in Chapter 5.)

- **Frameworks:** The SDK's multiple *frameworks* (code libraries that act a lot like prefab building blocks for building your app) help make it easy to develop apps for the Mac as well as for the iPad, iPhone, and iPod touch. Every iPad, iPhone, and iPod touch application is built using the `UIKit` framework and therefore has essentially the same core architecture; our sample app DeepThoughts in Part IV also uses the `Foundation` and `CoreGraphics` frameworks. Creating an app can be thought of as simply adding your application-specific behavior to the frameworks. The frameworks do all the rest. The frameworks provide fundamental code for building your iPad app: the required application behavior, classes for windows, views (including those that display text and Web content), controls, and view controllers. (I cover all these things in Chapter 7.) The `UIKit` framework even provides standard interfaces to core location data, the user's contacts and Photo library, and accelerometer data.
- **Interface Builder:** You find out about Interface Builder in Chapter 5 and use it to build the user interface for the DeepThoughts application in Part IV. But Interface Builder is more than your run-of-the-mill program for building graphical user interfaces. In Chapter 11, you see how Xcode and Interface Builder work together to give you ways to build (and automatically create at runtime) the user interface, as well as helping to create the infrastructure for your application.
- **iPad/iPhone Simulator:** The Simulator enables you to debug your app and do some other testing on your Mac by simulating the iPad or iPhone. The Simulator runs most iPad and iPhone apps, but it doesn't support some hardware-dependent features. I give you a rundown on the Simulator in Chapter 5.
- **Instruments:** The Instruments application lets you measure how your app performs while it's running on an iPad. It gives you a number of performance metrics, including those for testing memory and network use. It also works (in a limited way) on the Simulator, and you can test some aspects of your design there.

REMEMBER

The Simulator doesn't emulate such real-life iPad characteristics as CPU speed or memory throughput. If you want to understand how your app performs on the iPad from a user's perspective, you have to use the actual iPad and the Instruments application.

Looking forward to using the SDK

The tools in the SDK support a development process that most people find comfortable. They allow you to rapidly get a user interface up and running to see what it actually looks like. You can add code a little at a time and then run it after each new addition to see how it works. I take you through this incremental process as you develop the DeepThoughts app; for now, here's a bird's-eye view of iPad app development, one step at a time:

1. **Start with Xcode.**

   Xcode provides several project templates that you can use to get off to a fast start. (In Chapter 5, you do just that, and then you add code and more interface objects in Part IV.)

2. **Design and create the user interface.**

   Interface Builder has graphic-design tools you can use to create your app's user interface. These tools save you a great deal of time and effort. They also reduce the amount of code you have to write by creating resource files that your app can then upload automatically.

   REMEMBER

   If you don't want to use Interface Builder, you can always build your user interface from scratch, creating each individual piece and linking them all together within your app. Sometimes Interface Builder is the best way to create onscreen elements; sometimes the hands-on approach works better.

3. **Write the code.**

   The Xcode editor provides several features that help you write code. You can find out more about these features in Chapter 10.

4. **Build and run your app.**

   You build your app on your Mac and run it in the iPad/iPhone Simulator application or (provided you've joined the iOS Development Program) on your iPad.

5. **Test your app.**

   You'll want to test the functionality of your app as well as response time.

6. **Measure and tune your app's performance.**

   After you have a running app, make sure that it makes optimal use of resources such as memory and CPU cycles.

7. **Do it all again until you're done.**

Resources in the Dev Center

You're not left on your own when it comes to the Seven-Step Plan for Creating Great iPad Apps in the preceding section. After all, you have this book to help you on the way — as well as a heap of information squirreled away in various corners of the iOS Dev Center. (Refer to Figure 4-6 for the links.) The following resources are especially helpful:

- **Getting Started Videos:** These videos are relatively light on content.
- **The iOS Reference Library:** This library includes all the documentation you could ever want for developing for the iPad, iPod touch, and iPhone

(except, of course, the answer to that one question you really need answered at 3 a.m., but that's the way it goes). To be honest, most of this stuff turns out to be really useful only *after* you have a good handle on what you're doing. As you go through this book, however, you'll discover that an easier way to access some of this documentation will be through the Xcode Documentation window, described in Chapter 10. The iOS Reference Library includes the following documents:

- *Getting Started Documents (in the Library, and also a link on the iOS Dev Center page):* Think of them as an introduction to the materials in the iOS Reference Library, which is the essential library for learning about developing for the iPad, iPod touch, and iPhone. These give you an overview of development and best practices. Included is "Getting Started with iOS," which includes the "Learning Objective-C: A Primer," an overview of Objective-C, the programming language you'll use to code your apps. In the Guides section, you can find "iPad Human Interface Guidelines" and many other useful documents.

  TIP

  If you've never programmed in the Objective-C language, you should check out the "The Objective-C Programming Language" reference document in the Guides section of the iOS Reference Library. If you want to get a handle on Objective-C as quickly (and painlessly) as possible, go get yourself a copy of *Objective-C For Dummies* by co-author Neal. (Neal does a great job explaining everything you need to know in order to program in Objective-C, and he assumes you have little or no knowledge of programming.)

- *Coding How-To's (in the Library, and also a link on the iOS Dev Center page):* These tend to be a lot more valuable when you already have something of a knowledge base.

- *Sample Code (in the Library, and also a link on the iOS Dev Center page):* On the one hand, sample code of any kind is always valuable. Most good developers look over sample apps before they get started building their own. They take something that closely approximates what they want to do and then modify it until it does exactly what they want it to do. When I started iPad development, there were no books like this one; so much of what I learned came from looking at the samples and then making some changes to see how things worked. On the other hand, perusing the sample apps can give you hours of (misguided) pleasure and can be quite the time waster.

✓ **Apple Developer Forums:** I'm not the first to say that developer forums can be very helpful, and I'm also not the first to admit that they're a great way to procrastinate. As you scroll through the questions people have, be careful about some of the answers you see. No one is validating the information people are giving out. But take heart: Pretty soon you'll be able to answer some of those questions better yourself.

Downloading the SDK

Enough prep work. Time to do some downloading.

As of this writing, Apple offers version 4.2 of the SDK for both iPad and iPhone development. You use SDK version 4.2 to develop apps that are compatible with iOS 4.2 (the iPad's operating system).

To install the SDK, click the Downloads link near the top of the iOS Dev Center page under "Resources for iOS 4.2" (refer to Figure 4-6) to automatically scroll the page down to the Downloads section at the bottom (or scroll down the page until you find the Downloads section).

Version 4.2 requires an Intel-based Mac running Mac OS X Snow Leopard version 10.6.4 or a newer version.

WARNING!

By the time you read this book, it may no longer be version 4.2. You should download the latest SDK. That way, you get the most recent version to start with.

TIP

In the Downloads section is a link to a Read Me file (Xcode 3.2.5 Read Me). Click this link to read the file, which describes what Xcode can do (most of which I explain in Chapter 5).

After perusing the Read Me file, download the SDK by clicking the Xcode 3.2.5 and iOS SDK 4.2 link. You can watch the download in Safari's download window (which is only a little better than watching paint dry).

When it's done downloading, the iOS SDK window appears onscreen, complete with an installer and various packages tied to the install process. All you then have to do is double-click the iOS SDK installer and follow the (really simple) installation instructions. After you do all that, you have your very own iOS Software Development Kit on your hard drive, ready to create iPad apps.

Getting Yourself Ready for the SDK

Don't despair. The preceding process was tedious, but as the song goes, "It's all over now." Going through the process of registering and joining the program is probably the *second* most annoying part of your journey toward developing software for the iPad. The most annoying part is figuring out what Apple calls *provisioning* your iPad — the hoops you have to jump through to actually run your app on a real, tangible, existing iPad. You go through the provisioning process in Chapter 6, and frankly, getting *that* process explained is worth the price of this book.

In the next chapter, you get started using the SDK you just downloaded, and you'll become intimately acquainted with the SDK during the course of your project. I assume that you have some programming knowledge and that you also have some acquaintance with object-oriented programming and with some variant of C, such as C++, C#, and maybe even Objective-C. If those assumptions miss the mark, help me out, okay? Take another look at the "Resources in the Dev Center" section, earlier in this chapter, for an overview of some of the resources that can help you get up to speed on some programming basics. Or, better yet, get yourself a copy of *Objective-C For Dummies.*

I also assume that you're familiar with the iPad itself and that you've explored at least Apple's preinstalled apps to become familiar with the iPad's look and feel.

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 — step by step — 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 — which is certainly the fastest way to get started. The Seven Development Steps to iPad App Heaven should look something like this:

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 3–6) 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 — modify its code just a bit — 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 — 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 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, it's 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.)

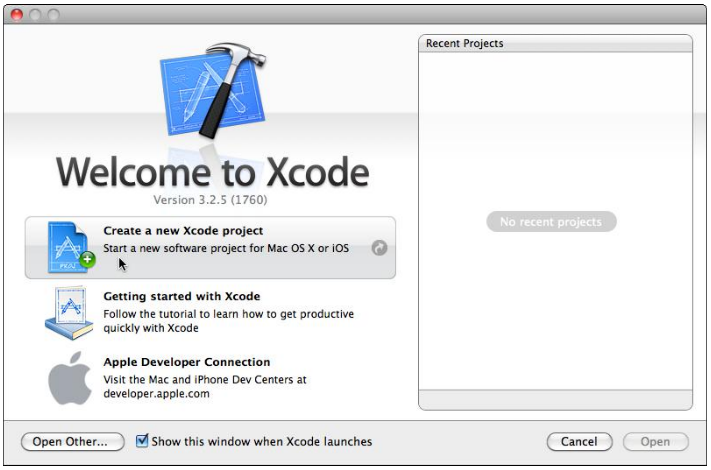

Welcome to Xcode
Version 3.2.5 (1760)
Create a new Xcode project
Start a new software project for Mac OS X or iOS
Getting started with Xcode
Follow the tutorial to learn how to get productive quickly with Xcode
Apple Developer Connection
Visit the Mac and iPhone Dev Centers at developer.apple.com
Recent Projects
No recent projects
Open Other...
Show this window when Xcode launches
Cancel
Open

Figure 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 ⌘+Shift+N.

   No matter what you do to start a new project, you're greeted by the New Project window.

   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. (Look ahead to 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.**

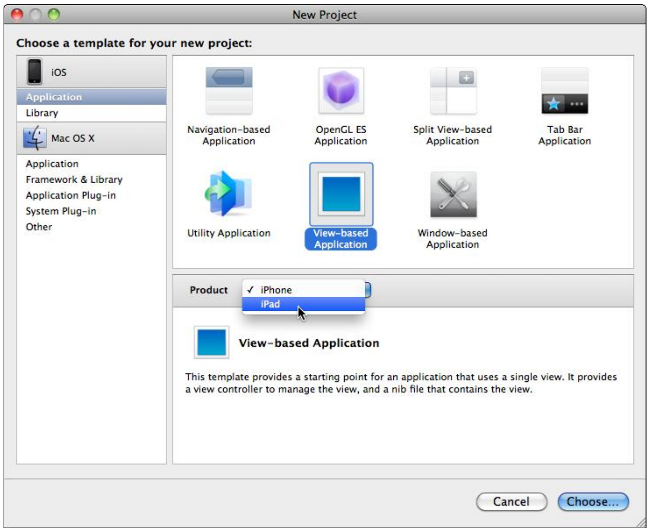
   You'll use the View-based Application option to start the DeepThoughts app, the first sample app, which you develop in Part IV.

   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 — 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.**



This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a nib file that contains the view.

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

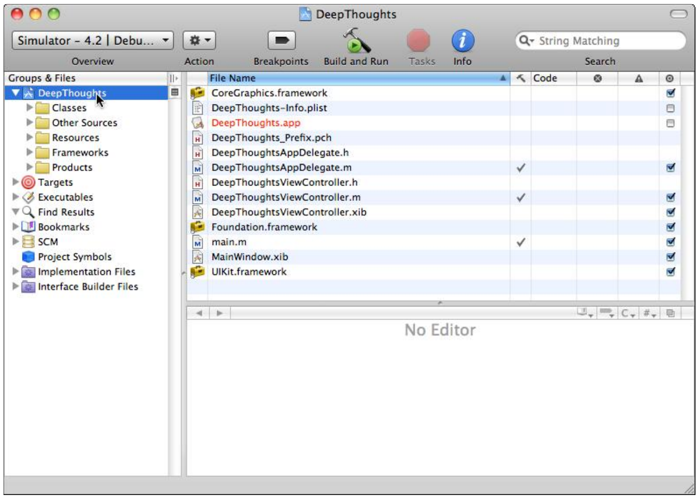


Figure 5-3: The Deep Thoughts Project window.

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.

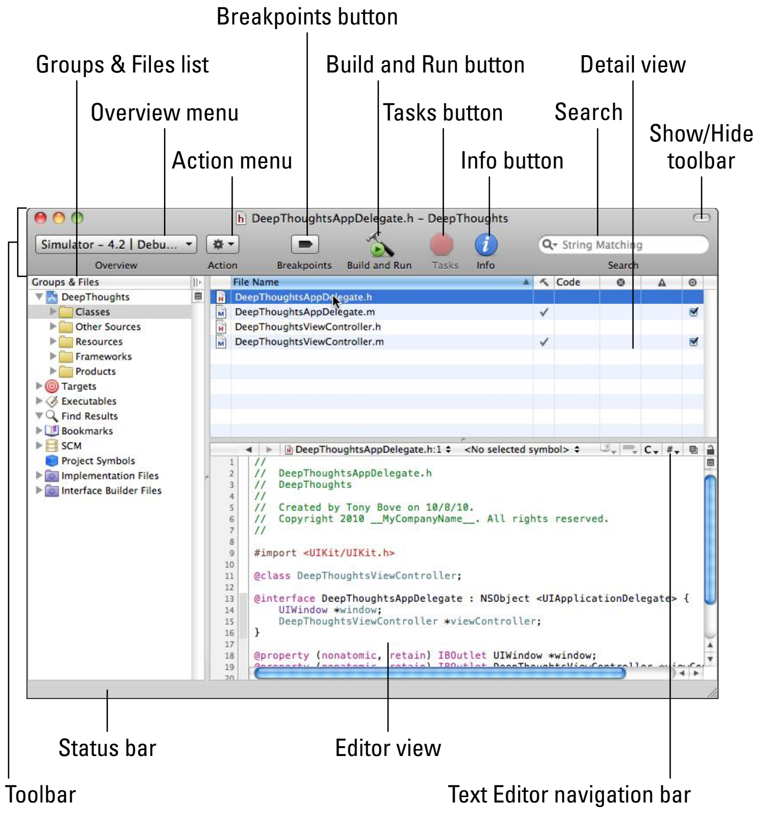
You have control over Command Central — you can organize your source files and resources as you see fit. The Groups & Files list on the left is an outline view of all of your project's files — source code, frameworks, and graphics, as well as some settings files. You can move files and folders around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

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 5-4:

- **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 — 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.)
    - 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 — and whether or not the process completed successfully.



```
//
//  DeepThoughtsAppDelegate.h
//  DeepThoughts
//
//  Created by Tony Bove on 10/8/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <UIKit/UIKit.h>

@class DeepThoughtsViewController;

@interface DeepThoughtsAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    DeepThoughtsViewController *viewController;
}

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

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

- **The favorites bar:** The favorites bar appears under the Toolbar and works like other favorites bars — 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 — 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 — 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).

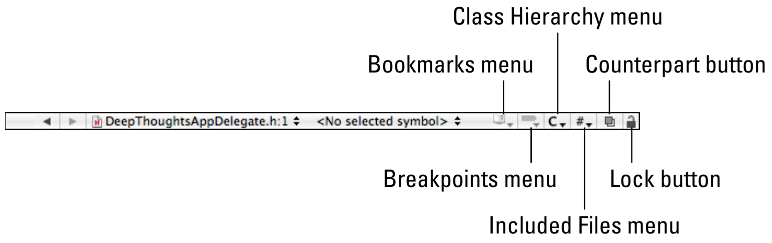


Figure 5-5: The Text Editor navigation bar.

- **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 — after all, that's what you'd expect from the Editor view — although some folks prefer to double-click a file in the 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.

  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 — selected and thus highly visible back in Figure 5-3 — 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 — 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 — 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 encounter in Chapter 16), images, 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 `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.

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 13.

- 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).

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

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 — even if all you did was choose a template from the Project window. Building and running a project is relatively simple:

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 — 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.

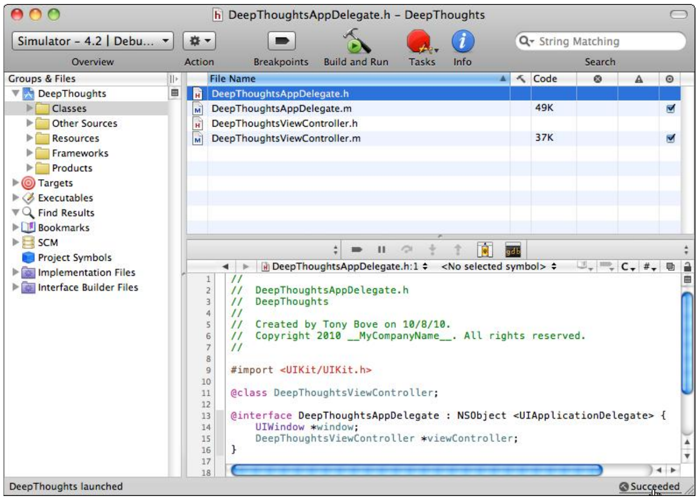
A *build configuration* tells Xcode the purpose of the built product. You can choose between Debug, which has features to help with debugging (there's a no-brainer for you); and Release, which results in smaller and faster binaries. You use Debug most of the time as you develop an app, and I use Debug for most of this book — 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 ⌘+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 — and (oh, yeah) whether the build was successful. Figure 5-6 shows that this was a successful build — you can tell by the `Succeeded` message in the bottom-right corner of the window.

   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.



File Name	Code
DeepThoughtsAppDelegate.h	
DeepThoughtsAppDelegate.m	49K
DeepThoughtsViewController.h	
DeepThoughtsViewController.m	37K

```
//
//  DeepThoughtsAppDelegate.h
//  DeepThoughts
//
//  Created by Tony Bove on 10/8/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <UIKit/UIKit.h>

@class DeepThoughtsViewController;

@interface DeepThoughtsAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    DeepThoughtsViewController *viewController;
}
```

Figure 5-6: A successful build.

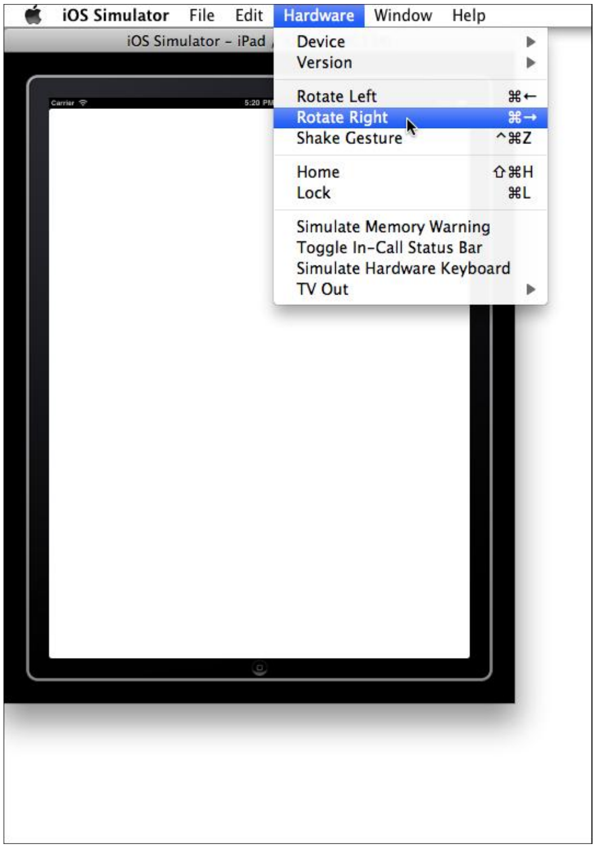


Figure 5-7: The Deep Thoughts app in the Simulator.

The Simulator

When you run your app, Xcode installs it on the iOS Simulator (or on 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 — 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) — 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, a drag, and so on. Table 5-1 shows you how to simulate gestures using your mouse and keyboard.

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.

Gesture	*iPad Action*
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 — the little *x* that appears in the upper-left corner of the icon — to make the app disappear.**
3. **Click the Home button — the one with a little square in it, centered below the screen — 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 the 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 — the red circle with the – 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 — which also removes all the apps you've installed — 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?

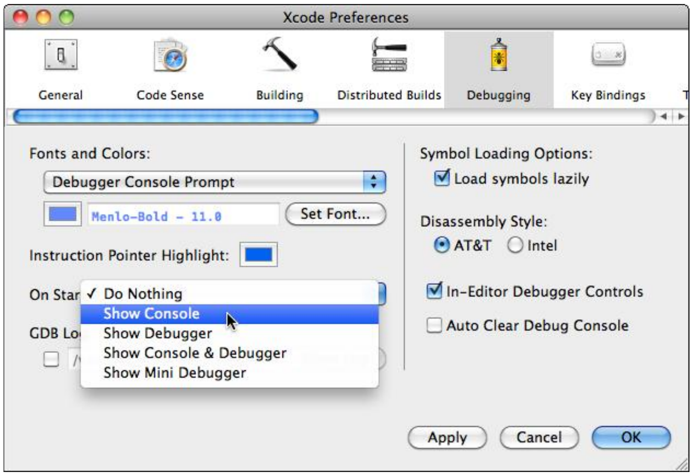
- **Two-finger limit:** You can simulate a maximum of two fingers. If your application's user interface can respond to touch events involving more than two fingers, you'll need to test that on an actual iPad. The motion of the two fingers is limited in the Simulator — you can't do two-figure swipes or 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.



Fonts and Colors:
Debugger Console Prompt
Menlo-Bold - 11.0 Set Font...
Instruction Pointer Highlight:
On Star ✓ Do Nothing
Show Console
Show Debugger
Show Console & Debugger
Show Mini Debugger
GDB Lo
Symbol Loading Options:
✓ Load symbols lazily
Disassembly Style:
[✓ AT&T] Intel
✓ In-Editor Debugger Controls
☐ Auto Clear Debug Console
Apply Cancel OK

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. (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.)

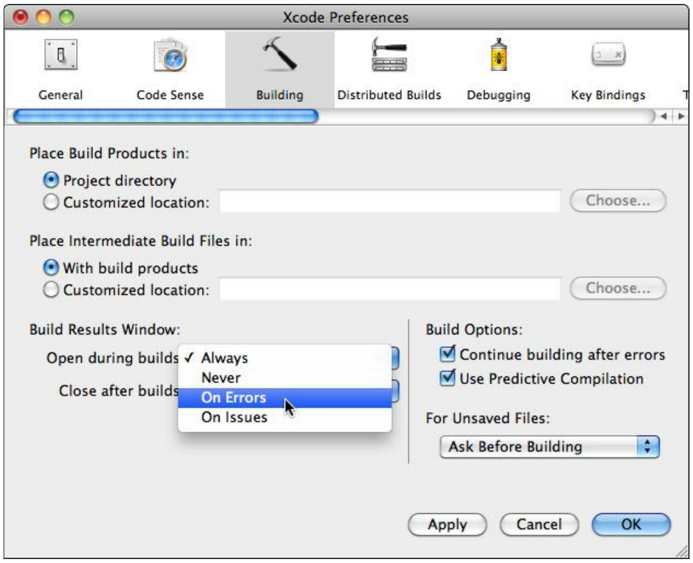
6. **Click the Documentation tab.**

   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.**



Place Build Products in:
[✓ Project directory] Customized location:

Place Intermediate Build Files in:
[✓ With build products] Customized location:

Build Results Window:
Open during builds: ✓ Always / Never / On Errors / On Issues
Close after builds

Build Options:
[✓] Continue building after errors
[✓] Use Predictive Compilation

For Unsaved Files: Ask Before Building

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 — 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 — 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.**

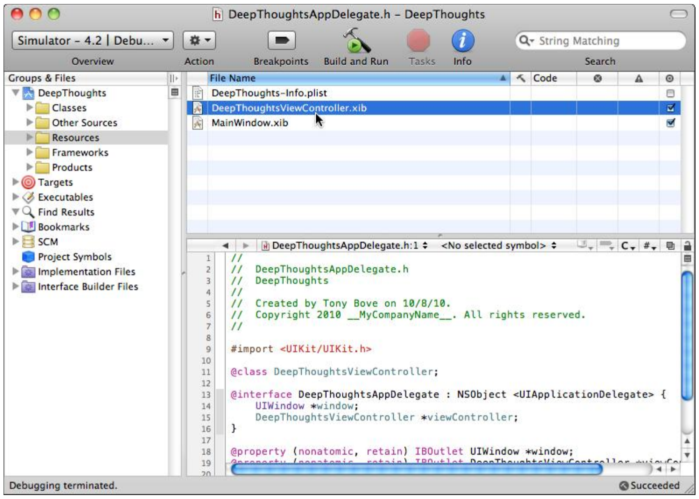
   The Detail view shows the files in the Resources group, as shown in Figure 5-10.

2. **Double-click the `DeepThoughtsViewController.xib` file in the Detail view. (Refer to Figure 5-10.)**

   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 — 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.



DeepThoughtsAppDelegate.h – DeepThoughts

Groups & Files: DeepThoughts, Classes, Other Sources, Resources, Frameworks, Products, Targets, Executables, Find Results, Bookmarks, SCM, Project Symbols, Implementation Files, Interface Builder Files

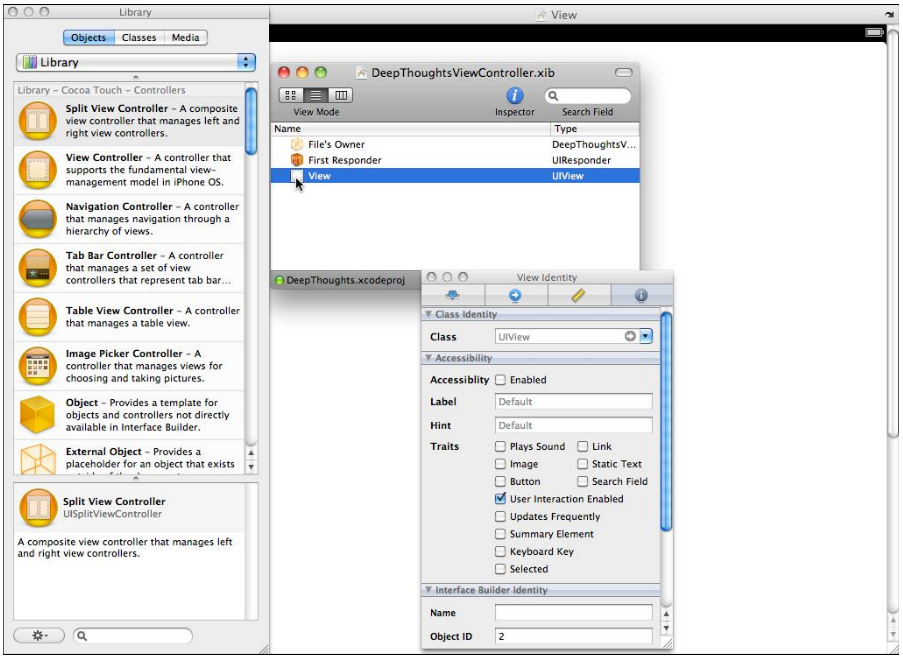
File Name: DeepThoughts-Info.plist, DeepThoughtsViewController.xib, MainWindow.xib

DeepThoughtsAppDelegate.h:1 <No selected symbol>

```
//
//  DeepThoughtsAppDelegate.h
//  DeepThoughts
//
//  Created by Tony Bove on 10/8/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <UIKit/UIKit.h>

@class DeepThoughtsViewController;

@interface DeepThoughtsAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    DeepThoughtsViewController *viewController;
}

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

Debugging terminated. Succeeded

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



Library – Cocoa Touch – Controllers

Split View Controller – A composite view controller that manages left and right view controllers.

View Controller – A controller that supports the fundamental view-management model in iPhone OS.

Navigation Controller – A controller that manages navigation through a hierarchy of views.

Tab Bar Controller – A controller that manages a set of view controllers that represent tab bar...

Table View Controller – A controller that manages a table view.

Image Picker Controller – A controller that manages views for choosing and taking pictures.

Object – Provides a template for objects and controllers not directly available in Interface Builder.

External Object – Provides a placeholder for an object that exists

Split View Controller
UISplitViewController

A composite view controller that manages left and right view controllers.

DeepThoughtsViewController.xib

Name	Type
File's Owner	DeepThoughtsV...
First Responder	UIResponder
View	UIView

View Identity

Class Identity
Class: UIView

Accessibility
Accessibility: ☐ Enabled
Label: Default
Hint: Default
Traits: ☐ Plays Sound ☐ Link ☐ Image ☐ Static Text ☐ Button ☐ Search Field ☑ User Interaction Enabled ☐ Updates Frequently ☐ Summary Element ☐ Keyboard Key ☐ Selected

Interface Builder Identity
Name:
Object ID: 2

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

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.

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) — and if you had a pal who knew the iPad backwards and forwards — 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.

  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 — how your new app appears on the iPad display.

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 — a white screen with the black status bar and battery icon — 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.

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 — 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 6

Death, Taxes, and iPad Provisioning

In This Chapter

- Running your application on the iPad
- Getting the app ready for distribution
- Taking the app to market — that is, the App Store

Benjamin Franklin once said, “In this world nothing can be said to be certain, except death and taxes.” Here’s another certainty in this earthly vale of tears: Everybody has the same hoops to jump through to get an app onto an iPad and then into the App Store — and nobody much likes jumping through hoops, but there they are.

So you’re working on your app, running it in the Simulator, as happy as a virtual clam, and all of a sudden you get this urge to see what your creation will look like on the iPad itself. Assuming that you’ve joined the requisite developer program (see Chapter 4), what do you have to do to get it to run on the iPad?

For most developers, getting their apps to run on the iPad during development can be one of the most frustrating things about developing software for the iPad. The sticking point has to do with a technical concept called *code signing,* a rather complicated process designed to ensure the integrity of the code and positively identify the code’s originator. Apple requires all iPad (and iPhone and iPod touch) apps to be digitally signed with a signing certificate — one issued by Apple to a registered developer — before the apps can be run on a development system and before they’re submitted to the App Store for distribution. This signature authenticates the identity of the developer of the app and ensures that there have been no changes to the app after it was signed.

As to why this is a big deal, here’s the short and sweet (and, to my ears, convincing) answer: Code signing is your way of guaranteeing that no bad guys have done anything to your code that can harm the innocent user.

Okay, so nobody really likes the process, but it's doable, and it's certainly worth the trouble. In this chapter, I give you an overview of how it all works by jumping right to that point where you're getting your app ready to be uploaded to the App Store and then distributed. I'm starting at the end of the process, which for all practical purposes begins with getting your app to run on an iPad during development. I'm doing the overview in this order because the hoops you have to jump through are a direct consequence of code signing and of how Apple manages it through the App Store and on the device.

After the overview, which will give you some context for the whole process, I revert back to the natural order of things and start with getting your app to run on your iPad during development.

How the Process Works

It's very important to keep clear that you have to go through *two* processes: One for development, and one for distribution. Both produce different (but similarly named) certificates and profiles, and you'll need to pay attention to keep them straight. This section starts with the *distribution* process — how you get your app to run on *other people's* iPads. Next up is the *development* process — how to get your app running on *your* iPad during development.

The distribution process

Before you can build a version of your app that will actually run on your users' iPads, Apple insists that you have the following:

- **A Distribution Certificate:** An electronic document that associates a *digital identity* (which it creates) with other information that you have provided that identifies you, including a name, e-mail address, or business. The Distribution Certificate is placed on your *keychain* — that place on your Mac that securely stores passwords, keys, certificates, and notes for users.
- **A Distribution Provisioning Profile:** These profiles are code elements that Xcode builds into your application, creating a kind of "code fingerprint" that acts as a unique *digital signature.*

After you've built your app for distribution, you then send it to Apple for approval and distribution. Apple verifies the signature to be sure that the code came from a registered developer (you) and has not been corrupted. Apple then adds its own digital signature to your signed app. iOS (the operating system for the iPad) runs only apps that have a digital signature from Apple. Doing it this way ensures iPad owners that the apps they download from the App Store have been written by registered developers and have not been altered since they were created.

To install your distribution-ready app on a device, you can also create an *Ad Hoc Provisioning Profile,* which enables you to distribute your app on up to 100 devices.

Although the system for getting apps on other people's iPads works pretty well — leaving aside the fact that Apple essentially has veto rights on every app that comes its way — there are some significant consequences for developers. In this system, there really is no mechanism for testing your app on the device it's going to run on:

- You can't run your app on an actual device until it's been code-signed by Apple, *but* Apple is hardly going to code-sign something that may not be working correctly.
- Even if Apple *did* sign an app that hadn't yet run on an iPad, that would mean an additional hassle: Every time you recompiled, you'd have to upload the app to the App Store again — *and* have it code-signed again because you had changed it, *and then* download it to your device.

A bit of Catch-22 here. (Milo Minderbinder would be proud.)

The development process

To deal with this problem, Apple has developed a process for creating a *Development Certificate* (as opposed to the Distribution Certificate discussed in the preceding section) and a *Development Provisioning Profile* (as opposed to the Distribution Provisioning Profile). It's easy to get these confused — the key words are Distribution and Development. With these items in hand, you can run your application on a *specific* device.

This process is required only because of the code-signing requirements of the distribution process.

To help you through this process, Apple provides the Development Provisioning Assistant to create your Development Provisioning Profile. This Profile includes your App ID, the Apple device UDID (a Unique Device Identifier for each iPad), and the Development Certificate (belonging to a specific developer).

An App ID is a unique identifier that iOS uses to allow your app to connect to the Apple Push Notification service, share keychain data between apps, and communicate with external hardware accessories that you want to pair your app with. But even if you don't want to do those things, you need to create an App ID anyway in order to create a complete Development Provisioning Profile to install your app on an iOS–based device such as an iPad.

This Profile must be installed on each device on which you want to run your application code. (You see how that's done in the section "'Provisioning Your iPad for Development,'" later in this chapter.) Devices specified within the Development Provisioning Profile can be used for testing only by developers whose Development Certificates are included in the Provisioning Profile. A single device can contain multiple provisioning profiles.

It's important to realize that a Development Provisioning Profile (as opposed to a distribution one) *is tied to a device and a developer.*

Even with your Provisioning Profile(s) in place, when you compile your program, Xcode will build and sign (create the required signature for) your app *only* if it finds one of those Development Certificates in your keychain. Then, when you install a signed app on your provisioned iPad, iOS verifies the signature to make sure that (a) the app was signed and (b) the app has not been altered since it was signed. If the signature is not valid or if you didn't sign the code, iOS won't let the app run.

REMEMBER

This means that each Development Provisioning Profile is also tied to a particular Development Certificate.

And to make sure the message has really gotten across:

A Development Provisioning Profile is tied to a *specific device* and a *specific Development Certificate.*

Your app, during development, must be tied to a specific *Development Provisioning Profile* (which is easily changeable).

TIP

The process you're about to go through is akin to filling out taxes: You have to follow the rules, or there can be some dire consequences. But if you *do* follow the rules, everything works out, and you don't have to worry about it again. (Until it's time to develop the next app, of course.) Although this process is definitely not my favorite part of iPad software development, I've made peace with it, and so should you.

After developing your app, it's time for the next step: getting it ready for distribution. (This process is somewhat easier.) Finally, you definitely want to find out how to get your application into the App Store. After that, all you have to do is sit back and wait for fame and fortune to come your way — or read Chapter 3 again to discover why it hasn't yet.

WARNING!

What I describe on these pages is the way things looked when I wrote this book. What you see when you go through this process yourself may be slightly different from what you see here. Don't panic. It's because Apple changes things from time to time.

Organizing Your Account in the Member Center

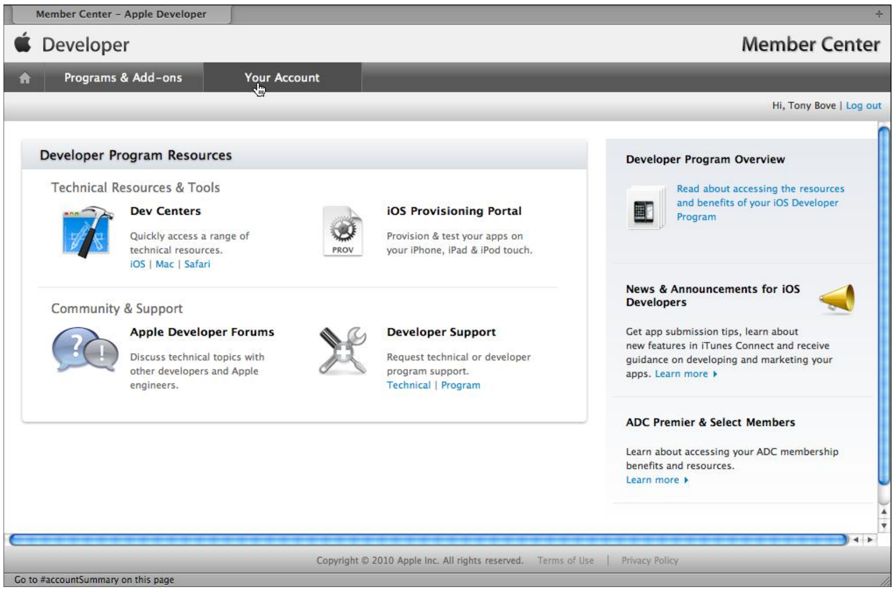
You can visit the Member Center to access account, team, and program information, to edit team members, or to request support.

You should bookmark the Member Center, shown in Figure 6-1, so that you can return to it quickly. It's really a hub for everything you need as a registered developer and iOS program member.

You've already identified yourself to Apple as one of two types of developers:

- **If you're enrolled in the Developer Program as an individual,** you're considered a Team Agent with all the rights and responsibilities.
- **If you're part of a company,** you've set up a team already. If not, visit the Member Center and click Your Account.

When you've settled the matter of which kind of developer you are (for Apple's purposes), you're ready to obtain your Development Certificates for team members' computers.


Member Center – Apple Developer
Developer
Member Center
Programs & Add-ons
Your Account
Hi, Tony Bove | Log out
Developer Program Resources
Technical Resources & Tools
Dev Centers
iOS | Mac | Safari
iOS Provisioning Portal
PROV
Community & Support
Apple Developer Forums
Developer Support
Technical | Program
Developer Program Overview
News & Announcements for iOS Developers
ADC Premier & Select Members
Learn more
Terms of Use | Privacy Policy
Go to #accountSummary on this page

Quickly access a range of technical resources.

Provision & test your apps on your iPhone, iPad & iPod touch.

Discuss technical topics with other developers and Apple engineers.

Request technical or developer program support.

Read about accessing the resources and benefits of your iOS Developer Program

Get app submission tips, learn about new features in iTunes Connect and receive guidance on developing and marketing your apps. Learn more

Learn about accessing your ADC membership benefits and resources.

Copyright © 2010 Apple Inc. All rights reserved.

Figure 6-1: The Member Center is a hub for managing development.

As I mention earlier in this chapter, to run the app on the iPad you must have a Provisioning Profile installed on the iPad, as well as a Development Certificate on your Mac.

REMEMBER

Development and Distribution stay off each other's turf. The Development Provisioning Assistant creates a *Development* Provisioning Profile, not a *Distribution* Provisioning Profile. You have to use the Provisioning section of the Program Portal, described later in this chapter, to create the Distribution Provisioning Profile required to distribute the app to customers through the App Store.

Obtaining a Development Certificate

Now I can go back to the natural order of things and start by explaining the process of getting your device ready for development. The most confusing part, in my opinion, is getting this Development Certificate. While Apple provides an Assistant that can do this (see "Getting an assist from the Development Provisioning Assistant"), and Xcode can also do it for you (see "Using Xcode to create a provisioning profile"), I found that neither worked for me after I had a hard drive failure and had to restore my Mac system. The tried-and-true manual method provided here does the job, if the other solutions don't work for you. Fortunately you have to do this only once, not for each project.

Although Apple documents the steps very well, do keep in mind that you really have to carry them out in exactly the way Apple tells you. There are no shortcuts! But if you do it the way it prescribes, you'll be up and running on a real device very quickly.

It's the rule: All iOS applications must be signed by a valid certificate before they can be run on an Apple device. In order to sign applications for testing purposes, team members each need an iOS Development Certificate. Each member of a team can have only one active certificate.

Here's the drill for getting your certificate:

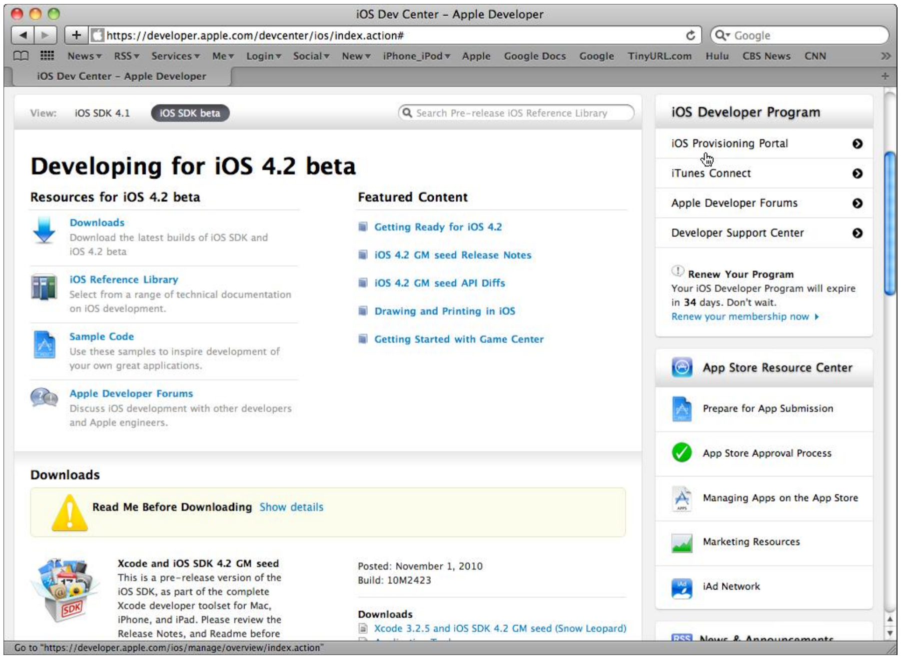
1. **Go to the iOS Dev Center Web site (which offers all of Apple's resources for iPad development) at**

   ```
   http://developer.apple.com/devcenter/ios
   ```

   If necessary, log in with your developer ID. The iOS Dev Center appears, as shown in Figure 6-2. You can see the iOS Provisioning Portal link, along with the iTunes Connect and the Developer Support Center links, in the iOS Developer Program section on the right side of the Web page. (You can see those links if you're a registered developer. You did take care of that, right? If not, look back at Chapter 4 for more on how to register.)

2. **Click the iOS Provisioning Portal link.**

   The iOS Provisioning Portal screen appears, as shown in Figure 6-3.


iOS Dev Center – Apple Developer

https://developer.apple.com/devcenter/ios/index.action#

News RSS Services Me Login Social New iPhone_iPod Apple Google Docs Google TinyURL.com Hulu CBS News CNN
View: iOS SDK 4.1 iOS SDK beta
Search Pre-release iOS Reference Library

Developing for iOS 4.2 beta

Resources for iOS 4.2 beta

Downloads
Download the latest builds of iOS SDK and iOS 4.2 beta

iOS Reference Library
Select from a range of technical documentation on iOS development.

Sample Code
Use these samples to inspire development of your own great applications.

Apple Developer Forums
Discuss iOS development with other developers and Apple engineers.

Featured Content

- Getting Ready for iOS 4.2
- iOS 4.2 GM seed Release Notes
- iOS 4.2 GM seed API Diffs
- Drawing and Printing in iOS
- Getting Started with Game Center

iOS Developer Program

- iOS Provisioning Portal
- iTunes Connect
- Apple Developer Forums
- Developer Support Center

Renew Your Program
Your iOS Developer Program will expire in **34** days. Don't wait.
Renew your membership now

App Store Resource Center

- Prepare for App Submission
- App Store Approval Process
- Managing Apps on the App Store
- Marketing Resources
- iAd Network

Downloads

Read Me Before Downloading Show details

Xcode and iOS SDK 4.2 GM seed
This is a pre-release version of the iOS SDK, as part of the complete Xcode developer toolset for Mac, iPhone, and iPad. Please review the Release Notes, and Readme before

Posted: November 1, 2010
Build: 10M2423

Downloads
Xcode 3.2.5 and iOS SDK 4.2 GM seed (Snow Leopard)

Go to "https://developer.apple.com/ios/manage/overview/index.action"

Figure 6-2: The Dev Center gateway to the Provisioning Portal.

3. **Watch the how-to video, "Obtaining Your Certificate," under Portal Resources in the right column of the Provisioning Portal.**

   This video does a stellar job of explaining how to obtain certificates. It actually explains all of the following steps, so you can read them while watching.

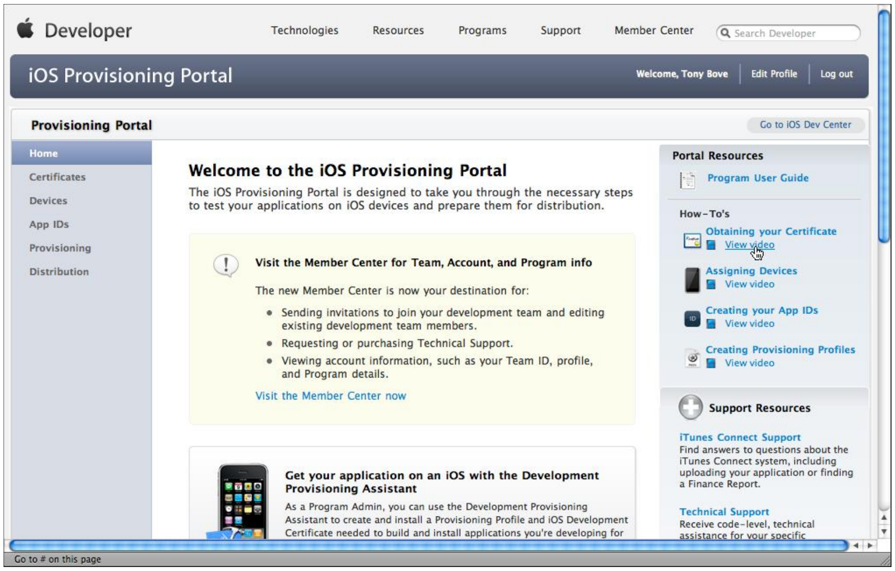
4. **Launch the Keychain Access application in Mac OS X.**

   You use Keychain Access to generate a Certificate Signing Request (CSR).

5. **In the Keychain Access Preferences menu, set the Online Certificate Status Protocol (OSCP) and Certificate Revocation List (CRL) pop-up menus to Off.**

6. **Choose Keychain Access➪Certificate Assistant➪Request a Certificate from a Certificate Authority.**

   Make sure that you are selecting Request a Certificate from a Certificate Authority and not selecting Request a Certificate from a Certificate Authority with *Private Key*. If you previously selected a noncompliant private key in the Keychain before starting this process, the resulting Certificate Request won't be accepted by the Provisioning Portal.



Welcome to the iOS Provisioning Portal

The iOS Provisioning Portal is designed to take you through the necessary steps to test your applications on iOS devices and prepare them for distribution.

Visit the Member Center for Team, Account, and Program info

The new Member Center is now your destination for:

- Sending invitations to join your development team and editing existing development team members.
- Requesting or purchasing Technical Support.
- Viewing account information, such as your Team ID, profile, and Program details.

Visit the Member Center now

Get your application on an iOS with the Development Provisioning Assistant

As a Program Admin, you can use the Development Provisioning Assistant to create and install a Provisioning Profile and iOS Development Certificate needed to build and install applications you're developing for

iTunes Connect Support
Find answers to questions about the iTunes Connect system, including uploading your application or finding a Finance Report.

Technical Support
Receive code-level, technical assistance for your specific

Figure 6-3: Behold the iOS Provisioning Portal.

7. **Enter your e-mail address in the User Email Address field, and enter your name in the Common Name field.**

   Make sure they match the information you submitted when you registered as an iOS Developer.

8. **Select both the Saved to Disk and the Let Me Specify Key Pair Information options and then click Continue.**

9. **Specify a filename and click Save, and in the following screen, select 2048 bits for the Key Size and RSA for the Algorithm. Then click Continue.**

   The Certificate Assistant creates a Certificate Signing Request (CSR) file on your Desktop.

10. **In the iOS Provisioning Portal (refer to Figure 6-3), first click Certificates in the left-most column, click the Development tab in the new page that appears, and then finally click Add Certificate.**

11. **In the dialog that appears, click the Choose file button, browse to your Desktop and select the CSR file you just created, and then click Submit.**

    After submitting the CSR file, you (or the Team Administrators for your team) are notified by e-mail of the certificate request. After submitting a CSR for approval, Team Administrators are directed to the Development tab of the Certificates section where CSRs can be approved or rejected by clicking the corresponding action next to each request. If you're part of a team, after your CSR is approved or rejected by a Team Administrator, you will be notified via e-mail of the change in your certificate status.

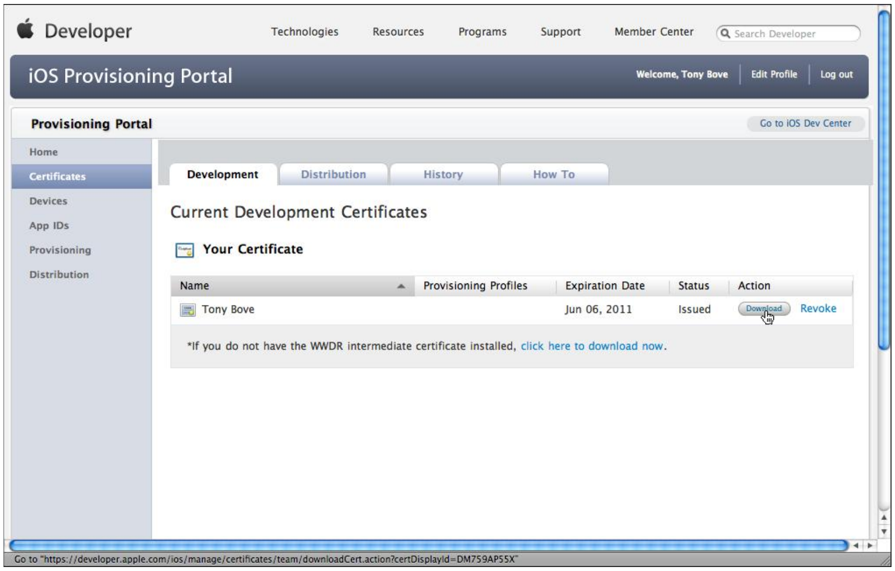
12. **After being notified that the CSR has been approved, go back to the Development tab of the Certificates section of the portal (shown in Figure 6-4), click the Click Here to Download Now link next to the WWDR Intermediate Certificate message, and then click Download to start downloading the certificate.**
13. **On your Mac Desktop, double-click the WWDR Intermediate certificate to launch Keychain Access and install the certificate.**
14. **If you are part of a development team (Company), upon CSR approval, other Team Members and Team Administrators can also click Download next to the certificate name in the Certificates section of the Provisioning Portal (refer to Figure 6-4) to download their certificates.**
15. **If you are part of a development team (Company), after the downloads are completed, other team members can double-click the downloaded `.cer` file on their Mac Desktops to launch Keychain Access and install their certificates.**

TIP

After installing the certificate, be sure to export a backup copy to another hard drive or storage medium in order to save it in case your hard drive fails — or to use with another development computer. Here's how:

1. **Choose Window➪Organizer from Xcode's main menu.**

   The Organizer window appears onscreen.
2. **In the Development group under Projects & Sources, select Developer Profile.**



Current Development Certificates

Your Certificate

Name	Provisioning Profiles	Expiration Date	Status	Action
Tony Bove		Jun 06, 2011	Issued	Download Revoke

*If you do not have the WWDR intermediate certificate installed, click here to download now.

Go to "https://developer.apple.com/ios/manage/certificates/team/downloadCert.action?certDisplayId=DM759AP55X"

Figure 6-4: Download your Development Certificate.

3. **Click Export Developer Profile to export the profile archive file.**
4. **You’re asked to create a password to use for importing to another computer. Do so.**
5. **Import this archive file into another computer you are using for development by launching Xcode, choosing Window➪Organizer, selecting Developer Profile in the Development group, clicking Import Developer Profile, and entering the password you chose in Step 4.**

Provisioning Your iPad for Development

After you’ve installed your Development Certificate, you have three choices for provisioning your iPad for development:

- **Let Xcode create a provisioning profile for you and download it to your iPad.** Xcode can even request a Development Certificate for you. I recommend this choice and show you how in the next section of this chapter.
- **Use the Development Provisioning Assistant to create your provisioning profile.** Do this if you need to start the process before you’ve connected your iPad. The Assistant can also request a Development Certificate for you. I show you how to use the Assistant in the section “Getting an assist from the Development Provisioning Assistant” in this chapter.
- **Do it all on your own.** This last option is a bit detailed, and I don’t cover it in this book (except the Development Certificate part — see “Obtaining a Development Certificate” in this chapter), but Apple provides plenty of information in the Provisioning Portal — watch the “Assigning Devices,” “Creating your App IDs,” and “Creating Provisioning Profiles” videos under Portal Resources in the right column. (Refer to Figure 6-3.)

WARNING!

The App ID created by Xcode or the Assistant *can’t be used* with the Apple Push Notification service. (This service lets your app keep its users up to date, offering the capability of sending a message that lets the user launch your app, triggering audible alerts with your own custom sounds, or adding a numbered badge to your app icon — for details, see the App IDs section of the iOS Developer Program Portal at `http:// developer.apple.com/ ios/manage/bundles`.) The App ID created by the Assistant *also can’t be used* for In App Purchase. (See Chapter 3 for details on In App Purchase.) If you’ve previously created an App ID already that can be used with the Apple Push Notification service or for In App Purchase, you *can’t* use the Assistant or Xcode to create a Development Provisioning Profile — you have to do it on your own.

Using Xcode to create a provisioning profile

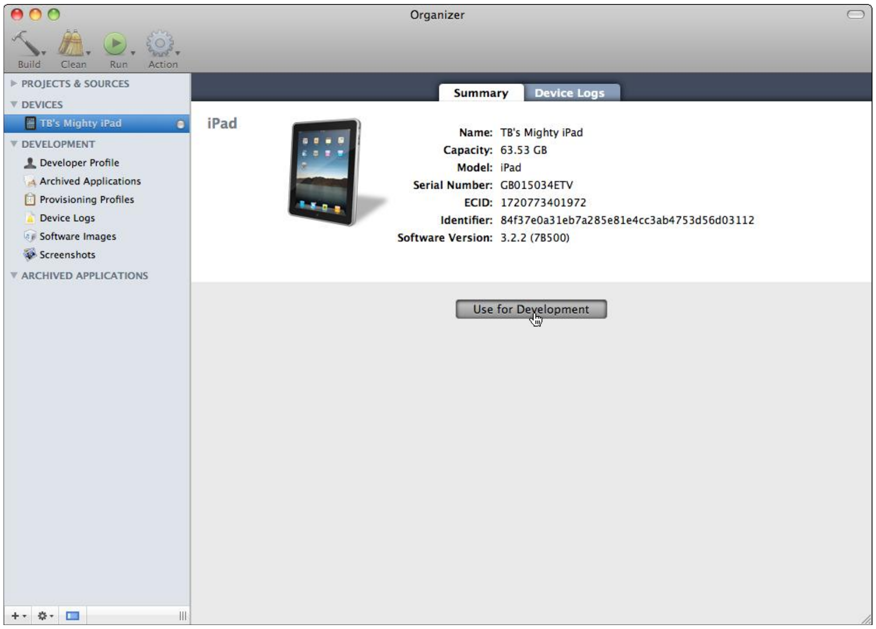
You can use Xcode (version 3.2.3 and newer) to auto-provision your iPad for you. It will create an App ID for you, create a provisioning profile for your Team Provisioning Profile, and download the profile to your iPad. All you need to do is connect your iPad to your computer. Follow these steps:

1. **Choose Window➪Organizer from Xcode's main menu to open the Organizer window. Then plug in your iPad.**

   You can see the result of this action in Figure 6-5.

2. **Click the Use for Development button.**

   After clicking Use for Development, Xcode asks you for your iOS Provisioning Portal logon (the same Apple ID you use to log in to the iOS Dev Center).



Name:	TB's Mighty iPad
Capacity:	63.53 GB
Model:	iPad
Serial Number:	GB015034ETV
ECID:	1720773401972
Identifier:	84f37e0a31eb7a285e81e4cc3ab4753d56d03112
Software Version:	3.2.2 (7B500)

Figure 6-5: Use this iPad for development.

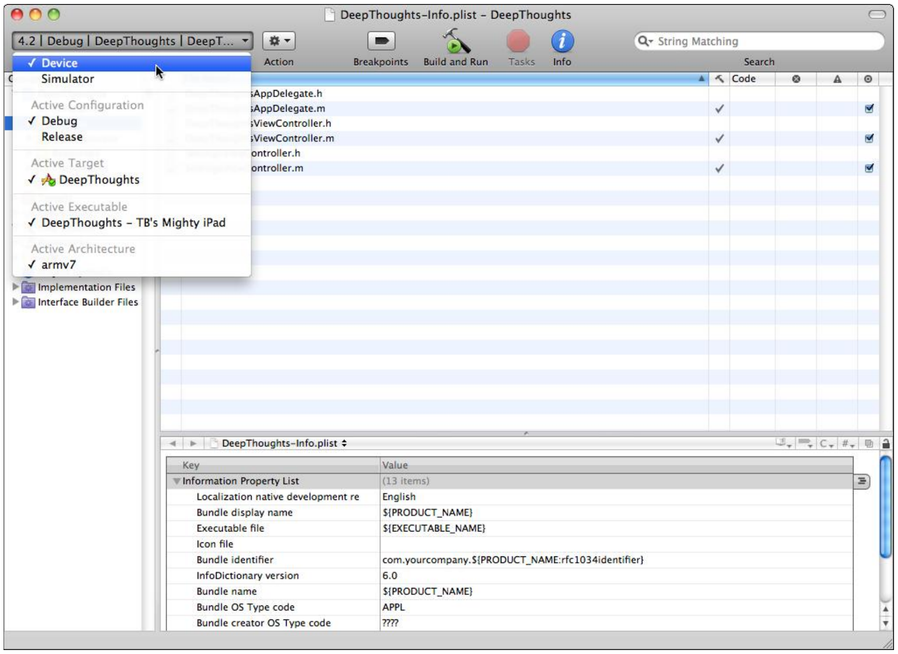
3. **Supply the requested username and password and then click Log In.**

   Xcode then looks for your Development Certificate; if you don't have one installed, you can click the Submit Request button to request one. (See

"Obtaining a Development Certificate" in this chapter.) If it finds the certificate, Xcode automatically provisions the iPad you connected. Xcode adds a provisioning profile called `Team Provisioning Profile: *` to your device that is shared among the members of your team who provision their devices automatically.

4. **Open your Xcode project (if it's not already open) and, in the Project window, choose Device as the active SDK in the Overview menu in the upper-left corner, as shown in Figure 6-6.**

You can then build your application and have it installed on the provisioned device (your iPad). When you build and run your app, you get the seemingly Catch-22 dialog that you can't run the app on your device because the device doesn't have the provisioning profile yet. To get beyond this roadblock, just click Install and Run in the dialog to install the profile and run the app.



Key	Value
Information Property List	(13 items)
Localization native development re	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.yourcompany.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????

Figure 6-6: Choose Device as the active SDK.

Getting an assist from the Development Provisioning Assistant

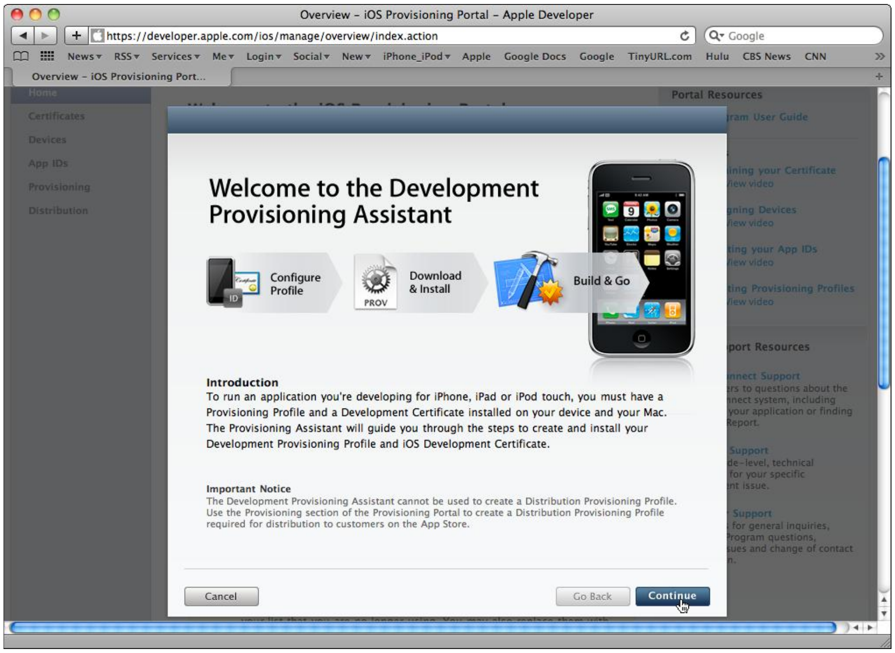
The whole point of the Development Provisioning Assistant is to guide you through the steps to create and install your Development Provisioning Profile and Development Certificate.

To use the Assistant, scroll the iOS Provisioning Portal page (refer to Figure 6-3) to the Provisioning Assistant section, and click the Launch Assistant button. The Assistant launches with a diagram showing the three steps for provisioning: configuring your profile, downloading and installing the certificate, and building your app, as shown in Figure 6-7. Click Continue to start.

Here's what the Development Provisioning Assistant has you do:

1. **Create a new or use an existing App ID.**

   If you already have an App ID for your app, you can click the Use an Existing App ID pop-up menu to select it; otherwise, select Create a New App ID.


Overview – iOS Provisioning Portal – Apple Developer
https://developer.apple.com/ios/manage/overview/index.action
Welcome to the Development Provisioning Assistant
Configure Profile
Download & Install
Build & Go
Cancel
Go Back
Continue

Introduction
To run an application you're developing for iPhone, iPad or iPod touch, you must have a Provisioning Profile and a Development Certificate installed on your device and your Mac. The Provisioning Assistant will guide you through the steps to create and install your Development Provisioning Profile and iOS Development Certificate.

Important Notice
The Development Provisioning Assistant cannot be used to create a Distribution Provisioning Profile. Use the Provisioning section of the Provisioning Portal to create a Distribution Provisioning Profile required for distribution to customers on the App Store.

Figure 6-7: Using the Development Provisioning Assistant.

2. **Enter a common name or description of your App ID to identify it.**

   This name or description appears in the iOS Provisioning Portal to identify your App ID.

3. **Choose an existing Apple device or assign a new device and then connect your iPad.**

   Development provisioning is also about the device, so you have to specify which particular device you're going to use and connect it. You can choose a device you have provisioned before by clicking Use an Existing

Apple Device and selecting it from the pop-up menu, or you can assign a new device by selecting Assign a New Apple Device. If you are assigning a new device, the Assistant asks for the device's Unique Device Identifier (UDID), which the Assistant shows you how to locate using Xcode. Connect your iPad with a USB cable to your computer, launch Xcode, and choose Window➪Organizer. The 40-character string in the Identifier field is the device's UDID; you can click it to select it and use Copy and Paste to copy it to the Assistant.

4. **Provide your Development Certificate.**

   All apps must be signed with a valid certificate before they can run on an Apple device. If your existing Development Certificate appears in the Assistant, all you need to do is click Continue. If you don't have a certificate yet, you need to create one — see "Obtaining a Development Certificate" in this chapter.

5. **Name your Provisioning Profile.**

   You then give your Provisioning Profile a name and click Generate. I suggest that you use a name that includes the app's name, the device (iPad), and the developer name. The Provisioning Profile pulls together your App ID (Step 1), Apple device UDID (Step 2), and Development Certificate (Step 3). When finished, the Assistant shows a check mark indicating success.

6. **Click Continue and then click the name of the Development Provisioning Profile to download and install it.**

   Your browser downloads the profile to your Desktop or to the Downloads folder (or to wherever your browser puts downloads).

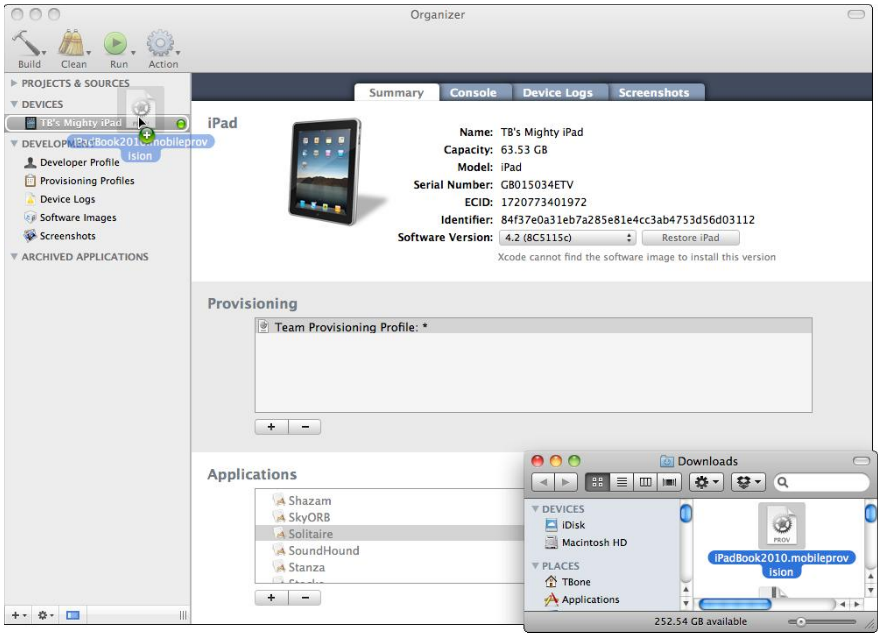
7. **Drag the provisioning profile over the Xcode icon in the Mac OS X Dock or drag it directly to the iPad's name in the Devices section of the Projects and Sources pane of the Organizer window, as shown in Figure 6-8.**

   Choose Window➪Organizer in Xcode first to open the Organizer window.

8. **Verify that the Provisioning Profile is installed.**

   In Xcode, choose Window➪Organizer and click the device's name in the Devices section of the Projects and Sources pane of the Organizer window, as shown in Figure 6-9 (TB's Mighty iPad). The profile should appear in the Provisioning section of the Summary pane for the device.

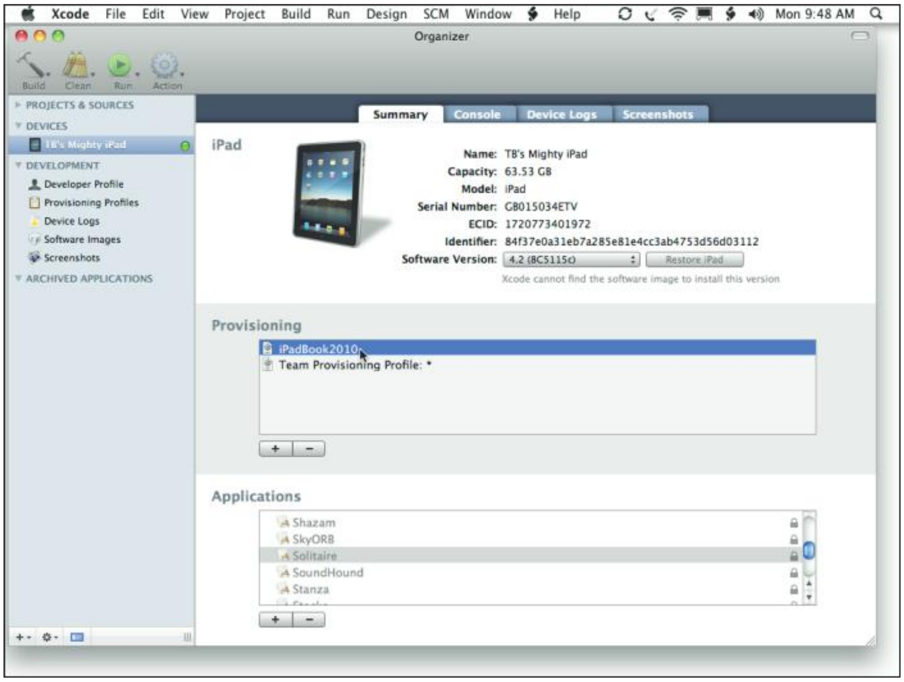
At this point, you can switch from the Organizer window to the Xcode Project window, and choose Device as the active SDK (refer to Figure 6-6). You can then rebuild your app so that it is reinstalled on the provisioned iPad — follow the Build and Run instructions in Chapter 5.



Name: TB's Mighty iPad
Capacity: 63.53 GB
Model: iPad
Serial Number: GB015034ETV
ECID: 1720773401972
Identifier: 84f37e0a31eb7a285e81e4cc3ab4753d56d03112
Software Version: 4.2 (8C5115c)
Xcode cannot find the software image to install this version

Team Provisioning Profile: *

Figure 6-8: Drag the provisioning profile to the Organize window of Xcode.



Name: TB's Mighty iPad
Capacity: 63.53 GB
Model: iPad
Serial Number: GB015034ETV
ECID: 1720773401972
Identifier: 84f37e0a31eb7a285e81e4cc3ab4753d56d03112
Software Version: 4.2 (8C5115c)
Xcode cannot find the software image to install this version

iPadBook2010
Team Provisioning Profile: *

Figure 6-9: The profile is listed in the Provisioning section as installed in the device.

Provisioning Your Application for the App Store or Ad Hoc Distribution

Before going any further, you need to visit the App Store Resource Center to get as much information as possible before submitting your app. To get there, click the App Store Resource Center link in the iOS Dev Center. (Refer to Figure 6-2.) Here you find information about how to submit your app, what to expect in the approval process, how to manage your apps in the store, and how to raise awareness and market your apps. You should read these sections carefully because Apple changes procedures and resources from time to time.

TIP

At some point, you should visit the Marketing Resources section of the App Store Resource Center to become an Authorized Licensee for marketing images. You can then use the App Store artwork and iPad images in your advertising, Web sites, and other marketing materials.

Although there's no dedicated assistant to help you provision your application for the App Store, that process is actually a little easier — which may be why there's no assistant for it.

You start at the Provisioning Portal (refer to Figure 6-3), but this time you click the Distribution link in the menu on the left side of the page. Doing so takes you to the Prepare App tab of the Distribution page, shown in Figure 6-10, where you can find an overview of the process, as well as links that take you where you need to go when you click them.

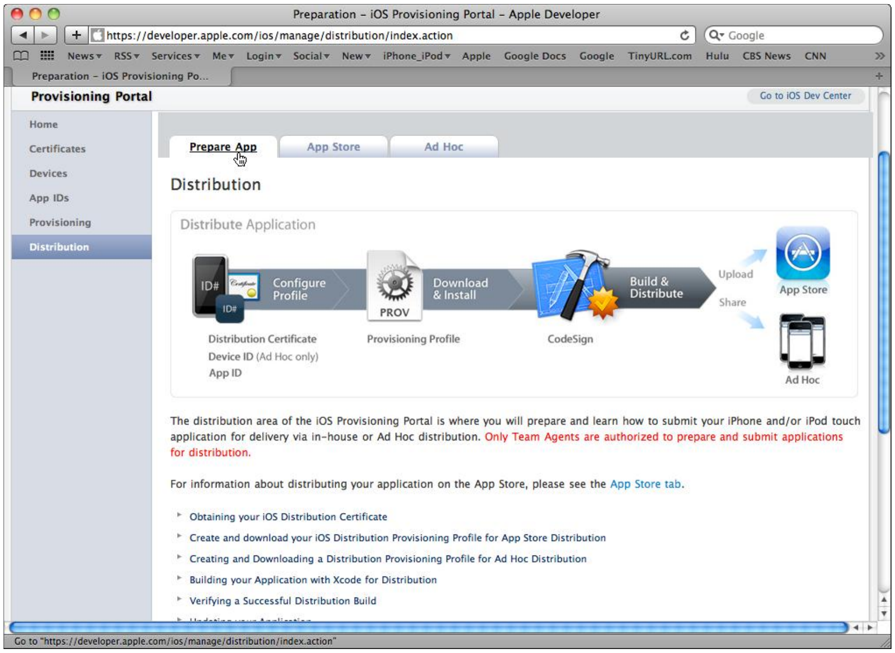
REMEMBER

You actually jump through some of the very same hoops you did when you provisioned your device for development — except that this time, you're going after a *Distribution* Certificate.

Here's the step-by-step account:

1. **Obtain your Distribution Certificate.**

   To distribute your iPad app, you (as an Individual developer, or functioning as the Team Agent for your development team) create a Distribution Certificate. This works much like the Development Certificate, except that only the Team Agent (or whoever is enrolled as an Individual developer) can get one. Clicking the Obtaining Your iOS Distribution Certificate link on the Prepare App page (shown near the bottom of Figure 6-10) leads you through the process — which is the same process I describe in "Obtaining a Development Certificate" earlier in this chapter.


Preparation – iOS Provisioning Portal – Apple Developer
https://developer.apple.com/ios/manage/distribution/index.action
Google
News RSS Services Me Login Social New iPhone_iPod Apple Google Docs Google TinyURL.com Hulu CBS News CNN
Preparation – iOS Provisioning Po...
Provisioning Portal
Go to iOS Dev Center
Home
Certificates
Devices
App IDs
Provisioning
Distribution
Prepare App
App Store
Ad Hoc
Configure Profile
Download & Install
PROV
Build & Distribute
Upload
Share
App Store
Ad Hoc
Distribution Certificate
Device ID (Ad Hoc only)
App ID
Provisioning Profile
CodeSign
Go to "https://developer.apple.com/ios/manage/distribution/index.action"

Distribution

Distribute Application

The distribution area of the iOS Provisioning Portal is where you will prepare and learn how to submit your iPhone and/or iPod touch application for delivery via in-house or Ad Hoc distribution. Only Team Agents are authorized to prepare and submit applications for distribution.

For information about distributing your application on the App Store, please see the App Store tab.

- Obtaining your iOS Distribution Certificate
- Create and download your iOS Distribution Provisioning Profile for App Store Distribution
- Creating and Downloading a Distribution Provisioning Profile for Ad Hoc Distribution
- Building your Application with Xcode for Distribution
- Verifying a Successful Distribution Build

Figure 6-10: Getting your app ready for distribution: You are here.

2. **Click Provisioning in the left column of the Provisioning Portal and click the Distribution tab, as shown in Figure 6-11, to create your Distribution Provisioning Profile for App Store Distribution.**

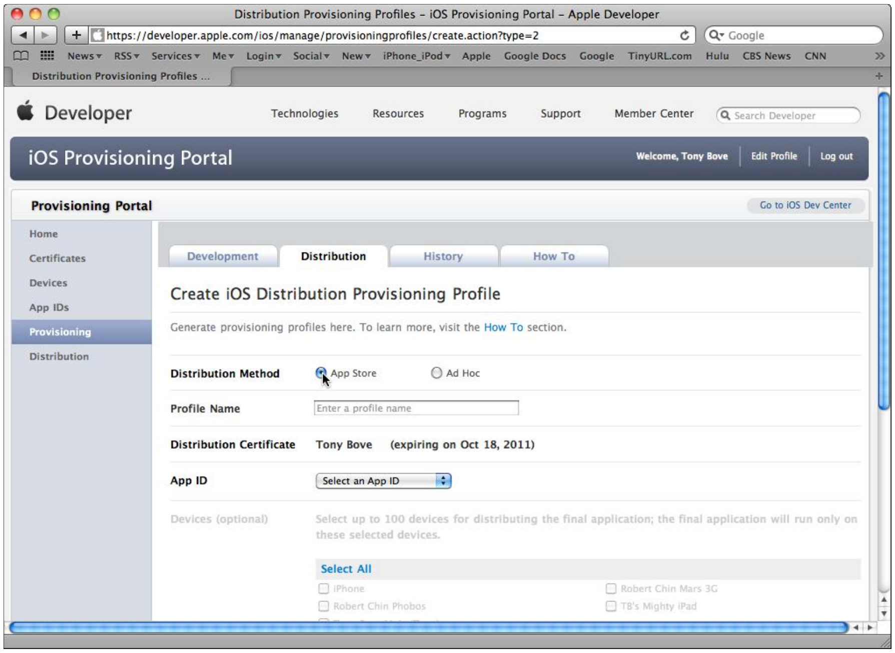
   To build your app successfully with Xcode for distribution via the App Store, first you have to create and download an App Store Distribution Provisioning Profile, which is (lest you forget) *different* from the Development Provisioning Profiles described in the previous section.

   Apple will accept an app only after it's built with an App Store Distribution Provisioning Profile.

3. **Click App Store (refer to Figure 6-11) and enter the name for your Distribution Provisioning Profile.**

   You should also see your iOS Distribution Certificate already identified under the Profile Name field. (Refer to Figure 6-11.) If it's not there, go back to Step 1.

4. **Choose the App ID for the distribution in the Select App ID pop-up menu, or choose All to build all of your apps with your single Distribution Provisioning Profile.**


Distribution Provisioning Profiles – iOS Provisioning Portal – Apple Developer

https://developer.apple.com/ios/manage/provisioningprofiles/create.action?type=2

Create iOS Distribution Provisioning Profile

Generate provisioning profiles here. To learn more, visit the How To section.

Distribution Method: [✓ App Store] Ad Hoc

Profile Name: Enter a profile name

Distribution Certificate: Tony Bove (expiring on Oct 18, 2011)

App ID: Select an App ID

Devices (optional): Select up to 100 devices for distributing the final application; the final application will run only on these selected devices.

Select All

☐ iPhone ☐ Robert Chin Mars 3G

☐ Robert Chin Phobos ☐ TB's Mighty iPad

Figure 6-11: Choose App Store as your distribution method.

5. **Click Submit, and then click on the name of the Distribution Provisioning Profile to download and install it.**

   Your browser downloads the profile to your Desktop or to the Downloads folder (or to wherever your browser puts downloads).

6. **Drag the provisioning profile over the Xcode icon in the Mac OS X Dock, or drag it directly to the iPad's name in the Devices section of the Projects and Sources pane of the Organizer window.**

   This loads your Distribution Provisioning Profile into Xcode, and you're ready to build an app you can distribute for use on actual iPads.

7. **(Optional) You can also create and download a Distribution Provisioning Profile for Ad Hoc Distribution.**

   Going the Ad Hoc Distribution route enables you to distribute your application to up to 100 users without going through the App Store. Scroll the Prepare App page (refer to Figure 6-10) to click the Creating and Downloading a Distribution Provisioning Profile for Ad Hoc Distribution link, which leads you through the process. (Ad Hoc Distribution is beyond the scope of this book — the iOS Provisioning Portal has more info about this option.)

Building Your App for Distribution

After you download the distribution profile, you can build your app for distribution — rather than just building it for testing purposes, which is what you've been doing so far. It's a well-documented process that you start by scrolling the Prepare App tab of the Distribution page (refer to Figure 6-10) and clicking the Building Your Application with Xcode for Distribution link. It goes like this:

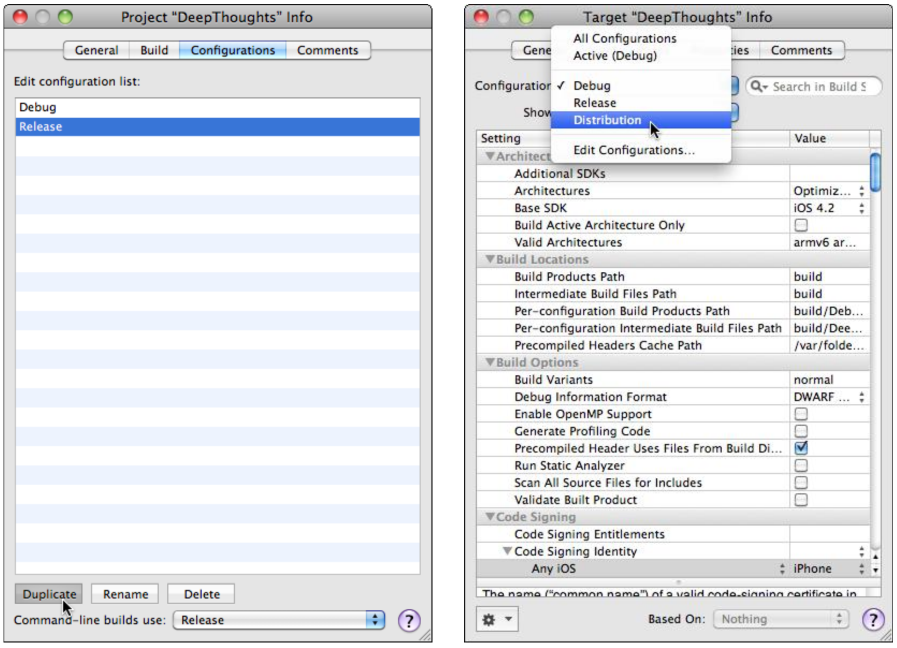
1. **Open the Xcode project, select the project name at the top of the Groups & Files list, choose File➪Get Info to show the Info window, and click the Configurations tab.**

   The Configurations pane of the Info window appears as shown in Figure 6-12 (left side).

2. **Select the Release configuration (refer to Figure 6-12), click the Duplicate button, and rename this new configuration `Distribution`.**

3. **Select the project name in the Targets section of the Groups & Files list, choose File➪Get Info to show the Target Info window, and click the Build tab.**

   The Build pane of the Target Info window appears as shown in Figure 6-12 (right side).



Setting	Value
▼Architect	
Additional SDKs	
Architectures	Optimiz…
Base SDK	iOS 4.2
Build Active Architecture Only	☐
Valid Architectures	armv6 ar…
▼Build Locations	
Build Products Path	build
Intermediate Build Files Path	build
Per-configuration Build Products Path	build/Deb…
Per-configuration Intermediate Build Files Path	build/Dee…
Precompiled Headers Cache Path	/var/folde…
▼Build Options	
Build Variants	normal
Debug Information Format	DWARF …
Enable OpenMP Support	☐
Generate Profiling Code	☐
Precompiled Header Uses Files From Build Di…	✓
Run Static Analyzer	☐
Scan All Source Files for Includes	☐
Validate Built Product	☐
▼Code Signing	
Code Signing Entitlements	
▼Code Signing Identity	
Any iOS	iPhone

The name ("common name") of a valid code-signing certificate in
Based On: Nothing

Figure 6-12: The Configurations Info panel (left) and the Target Info panel (right).

4. **Choose Distribution from the Configuration pop-up menu in the Build pane. (See Figure 6-12, right side.)**

5. **Choose the iOS Distribution Certificate/Provisioning Profile pair from the Any iOS Device pop-up menu below the Code Signing Identity field.**

   Choose the profile pair you want to use for signing and installing your app. Your iOS Distribution Certificate will be in bold with the Provisioning Profile associated with it in gray above it.

6. **Click the Properties tab of the Target Info window and enter the Bundle Identifier portion of your App ID.**

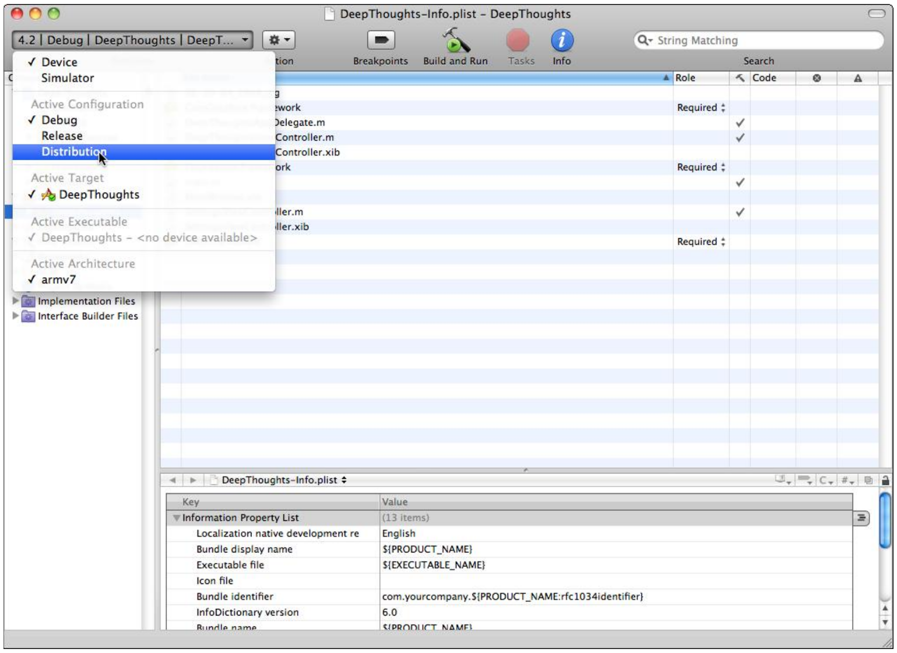
   If you have used an explicit App ID (as I did), you must enter the Bundle Identifier portion of the App ID in the Identifier field. For example, enter **com.domainname.applicationname** if your App ID is `A1B2C3D4E5.com.domainname.applicationname`.

7. **Back in the Xcode Project window, select Distribution as your Active Configuration in the Overview pop-up menu, as shown in Figure 6-13.**

8. **Choose Build➪Build.**

   WARNING!

   You must have already added an icon to your project before taking this step— see Chapter 9 for details on adding the icon, which appears on the iPad Home screen.



Key	Value
Information Property List	(13 items)
Localization native development re	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.yourcompany.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}

Figure 6-13: Select Distribution as your Active Configuration.

9. **In your Xcode project, verify that it worked.**

   Scroll the Prepare App tab of the Distribution page (refer to Figure 6-10) and click the Verifying a Successful Distribution Build link to get the verification process started. In this case, there are some things missing in the heretofore well-explained step-by-step documentation — it tells you to open the Build Log detail view and confirm the presence of the `embedded.mobileprovision` file. (In Chapter 5, I show you how to keep the Build Results window open in Xcode, but if you haven't been doing that, choose Build➪Build Results.)

   Depending on the way the Build Results window is configured, you may see a window showing only the end result of your build. To get the actual log of the process and confirm the presence of the `embedded.mobileprovision` file, you have to change Errors & Warnings Only in the drop-down menu in the scope bar to All Messages.

   Make sure that the `embedded.mobileprovision` file is located in the proper Distribution build directory and is not located in a Debug or Release build directory. Also, confirm that the destination path (at the very end of the build message) is the app you are building. Finally, search for the term `CodeSign` in the Build Log's Detail view — this will take you to the line in the build log that confirms your app was signed by your iOS Certificate.

10. **If the build didn't work properly . . .**

    If your build log lacks the `embedded.mobileprovision` file or puts it in the wrong directory, make sure Distribution is set for the Build configuration (Step 4). Then choose Build➪Clean All Targets, relaunch Xcode, and reopen your project. Finally, redo Steps 5–9.

When you've done this elaborate (but necessary) song and dance, you're ready to rock 'n' roll. You can go to iTunes Connect, which is your entryway to the App store. This is where the *real* fun starts.

Using iTunes Connect to Manage Apps in the App Store

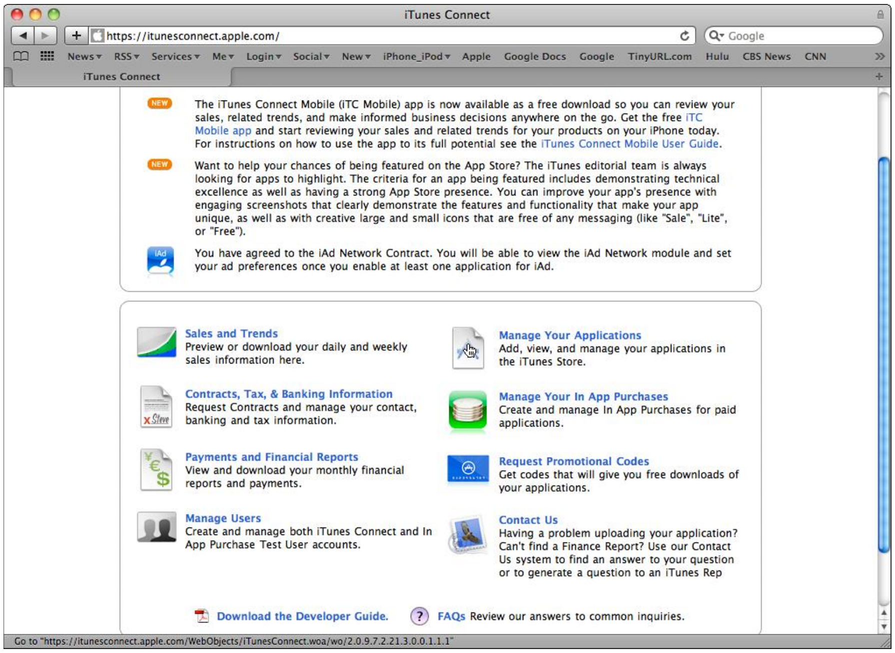
iTunes Connect is a group of Web-based tools that enables developers to submit apps to the App Store as well as to manage those apps of theirs that have found a home there. It's actually the very same set of tools that the other content providers — the music and video types — use to get their content into iTunes.

In iTunes Connect, you can check on your contracts, manage users, and submit your app with all its supporting documentation — the *metadata,* as Apple calls it — to the App Store. iTunes Connect is also where you get financial reports and daily/weekly sales trend data, as I describe in Chapter 3.

To go to iTunes Connect to add or manage apps in the store, click the iTunes Connect link under the iOS Developer Program heading in the right column of iOS Dev Center (refer to Figure 6-2) to go to the login page. You need to use your Apple ID and password to log in. Before you can do anything, you're asked to review and accept the iTunes Distribution Terms & Conditions. After taking care of that chore, you land on the iTunes Connect page, a portion of which is shown in Figure 6-14.

When you want to add an application to the App Store or manage what you already have there, the iTunes Connect main page is your control panel for getting that done.

You'll primarily be using three sections of the iTunes Connect page: the Manage Users section; the Contract, Tax & Banking Information section; and the Manage Your Applications section.


iTunes Connect
https://itunesconnect.apple.com/
Google
News RSS Services Me Login Social New iPhone_iPod Apple Google Docs Google TinyURL.com Hulu CBS News CNN
iTunes Connect
NEW

The iTunes Connect Mobile (iTC Mobile) app is now available as a free download so you can review your sales, related trends, and make informed business decisions anywhere on the go. Get the free iTC Mobile app and start reviewing your sales and related trends for your products on your iPhone today. For instructions on how to use the app to its full potential see the iTunes Connect Mobile User Guide.

Want to help your chances of being featured on the App Store? The iTunes editorial team is always looking for apps to highlight. The criteria for an app being featured includes demonstrating technical excellence as well as having a strong App Store presence. You can improve your app's presence with engaging screenshots that clearly demonstrate the features and functionality that make your app unique, as well as with creative large and small icons that are free of any messaging (like "Sale", "Lite", or "Free").

You have agreed to the iAd Network Contract. You will be able to view the iAd Network module and set your ad preferences once you enable at least one application for iAd.

Sales and Trends Preview or download your daily and weekly sales information here.

Manage Your Applications Add, view, and manage your applications in the iTunes Store.

Contracts, Tax, & Banking Information Request Contracts and manage your contact, banking and tax information.

Manage Your In App Purchases Create and manage In App Purchases for paid applications.

Payments and Financial Reports View and download your monthly financial reports and payments.

Request Promotional Codes Get codes that will give you free downloads of your applications.

Manage Users Create and manage both iTunes Connect and In App Purchase Test User accounts.

Contact Us Having a problem uploading your application? Can't find a Finance Report? Use our Contact Us system to find an answer to your question or to generate a question to an iTunes Rep

Download the Developer Guide. FAQs Review our answers to common inquiries.

Go to "https://itunesconnect.apple.com/WebObjects/iTunesConnect.woa/wo/2.0.9.7.2.21.3.0.0.1.1.1"

Figure 6-14: The iTunes Connect main page.

Managing Users

Users in this context means you and your fellow team members, not any future potential users of your app. Click the Manage Users link to find out what tools are available for managing how you and your team communicate about what's what with your app. When creating and editing an iTunes Connect user account, you can define user roles and *notifications* — the type of e-mails your fellow team members will receive regarding the main iTunes Connect account. When setting up accounts, keep in mind that you have four distinct user roles to choose from: Admin, Legal, Finance, and Technical.

Adding contract, tax, and banking information

After you've set up your various user accounts, proceed to the Contracts, Tax & Banking section to complete the paid application agreements and provide financial information relating to payment and tax withholdings from the sale of your apps.

If you plan on selling your application, you need to have your paid commercial agreement in place and signed before your application can be posted to the App Store.

REMEMBER

If your application is free, you've already entered into the freeware distribution agreement by being accepted into the iOS Developer Program; however, there is still a contract setup process that free application contracts will need to go through before your application will go live in the App Store. Contract approval can take a while, so you should probably fill out the contract information just to get it out of the way.

Start by clicking the Contracts, Tax & Banking Information link on the iTunes Connect main page. The Manage Your Contracts page appears. You use this page to create a contract for your paid app. You can also see that you already have, by default, a contract in effect for free apps. To create a new contract, select the box under Request Contract in the Request New Contracts section, and you're taken through a series of pages that ask you to provide the information Apple needs, including all the bank information.

If you're going to charge for your application, you have to provide even more information. Most of it is pretty straightforward, except for some of the banking information, which you do need to have available.

To change some of the information after you've entered it, you have to e-mail iTunes technical support. It behooves you to get it right the first time.

Here's what I'm talking about:

- **Bank name**
- **Bank address**
- **Account number**
- **Branch/Branch ID**
- **ABA/Routing Transit Number or SWIFT Code:** What this number is will depend upon where your bank is located. For United States banks, this number is the first nine digits of that long number at the bottom of your checks that also contains the account number. If you aren't sure what the routing number is, contact your bank. For non-U.S. banks, you may have to enter the SWIFT Code instead. You have to get that from your bank. The process also provides a look-up function to help you out.

Take it from me: It's far easier if you have all the bits and pieces together *before you start the actual upload process,* rather than having to scramble at 3 a.m. to find some obscure piece of information you need.

Adding the metadata and artwork

Here's an overview of the kind of information you need as you submit your app with iTunes Connect. (For more information, click the Prepare for App Submission link in the App Store Resource Center section of the iOS Dev Center page shown earlier in Figure 6-2.)

Metadata is the ever-present data about data. Here's what Apple wants from you:

- **Application Name:** The name must conform to guidelines for using Apple and all other trademarks and copyrights. Apple takes this very seriously, as evidenced by the company sending a cease-and-desist order to Neal's ISP when Neal tried (innocently) to use the word *iPhone* in his domain name. (A word to the wise: Don't mess with Apple.)
- **Application Description:** When you go through the process of uploading your data, the field you have to paste this into will say you're limited to 4,000 characters. Apple suggests no more than 580 characters, so that customers can view your entire iTunes Connect Application Description without clicking the More button in the App Store.

This description is what users will see when they click your app in the App Store or follow a link from another Web page to a browser page for the App Store, so it's important that this description is well written and points out all your app's key features.

TIP Don't include HTML tags; they will be stripped out when the data is uploaded. Only line breaks are respected.

- **Device:** Choose iPad (as of this writing).
- **Primary Category:** A drop-down menu offers the primary category choices for your app — choose one (the most important category). The App Store offers about 20 categories ranging from Reference to Games to Social Networking to Travel to Utility.
- **Secondary Category:** (Optional) These categories are the same that you see for the Primary Category — choose one.
- **Rating Information:** You're asked to provide additional information describing the content. This allows you to set your rating for your app for the purpose of parental controls on the App Store. You may see content types such as Cartoon or Fantasy Violence, Simulated Gambling, Mature/Suggestive Themes, and so on. For each type of content, you need to describe the level of frequency for that content — None, Infrequent/Mild, Frequent/Intense. Apple has strict rules stating that an app must not contain any obscene, pornographic, or offensive content. Oh and by the way, it's entirely up to Apple what is to be considered offensive or inappropriate.
- **Copyright:** Use a line such as

  © Copyright *your name* 2010. All rights reserved.

  You can type the copyright symbol by pressing Option-G. If you have any questions about copyright registration, talk to your lawyer or check out `www.copyright.gov`.
- **Version Number:** People usually start with 1.0. Then, as you update the app to respond to suggestions and constructive criticism, you can move on to 1.1 and eventually version 2.0.
- **SKU Number:** The Stock Keeping Unit (SKU) number is any alphanumeric sequence of letters and numbers that uniquely identifies your app in the system. (Be warned — this is not editable after you submit it.)
- **Keywords:** Keywords help people find your app in App Store searches. They can be single words or phrases — the text field is limited to 100 characters.

  Spend some time on this one — you can't change them until you submit a new version of your app or if the app status is Rejected. Keywords must be related to your application content — you can't use other app names, company names, or trademarked terms as keywords, or any kind of offensive language. Your App Name and Company are already searchable, so you don't need to include them in your keywords list.

- **Support URL and Company URL:** You need a support URL, which appears on the app product page at the App store — this is the link users will click if they need technical support from you or have a question about your app. You also need a company URL, which also appears on the app product page and enables potential customers to find out more about you. After you've assigned these URLs, you want to keep them unchanged for as long as possible, because people bookmark them. (You can use a redirect in your Web page to direct them to another page, if necessary.)

  If you don't have a Web site yet and don't know how to build one, try using iWeb with MobileMe (if you already have the service) or with your friendly ISP. MobileMe offers automatic Web publishing to a reasonably unique domain name that can serve well enough for your URLs — to find out more, see my book *iLife For Dummies.* You can also find out more about building a professional-looking site from David Crowder's book *Building a Web Site For Dummies,* 3rd Edition.

- **Support E-mail Address:** (For use by Apple only, not visible to end users of your app.) This address will likely be the one you used when you registered for the developer program.

- **Demo Account — Full Access:** This is a test account that the App Store *reviewers* can use to test your app. Include usernames, passwords, access codes, demo data, and so on. You should include any messages to the Apple app reviewers, in case they might incorrectly reject something — for example, by wrongly assuming you don't have permission to use a piece of music in the app when in fact the piece is in the public domain. Make sure the demo account works correctly. You'd hate to have your app rejected because you didn't pay attention to setting up a demo account correctly.

- **End User License Agreement:** (Optional) If you don't know what this is, don't worry. It's the legal document that spells out to your app's users what they're agreeing to do in order to use your app. Fortunately, the iTunes Store has a standard agreement, which has been time-tested — but you should read it anyway before you use it.

- **Availability Date:** This is the date your app will be available for download (for free apps) or for purchase-and-download.

- **Application Price:** Free is easier, but if you want to get paid, you have to select a price tier. The last time I tried it, you couldn't see the pricing matrix unless you had first selected one. To help you along, Tier 1 is US$0.99, Tier 2 is US$1.99, and so on.

- **Localization:** Specify additional languages (besides English) for your metadata. You can have your text and images in Italian in all Italian-speaking stores, for example.
- **App Store Availability:** The territories in which you would like to make your app available. (The default is all countries iTunes supports.)

When it comes to your artwork, a picture is worth a thousand words, so the App store gives you the opportunity to dazzle your app's potential users with some nice imagery:

- **iPad Home Screen Icon:** Your built app must include an icon sized at 72 x 72 pixels. You can add the icon directly to the app following the procedure I describe in Chapter 9. This icon will be displayed on the iPad home screen. You also need to supply a smaller version of this icon, at 48 x 48 pixels, for display in Spotlight search results and in the Settings application (if you provide settings).
- **Large Application Icon:** This icon will be used to display your app on your App Store page and other App Store pages. It needs to meet the following requirements, although the version you see in the App Store is resized by Apple:
  - 512 x 512 pixels (a square image)
  - 72 dots-per-inch (dpi)
  - JPEG or TIFF format (saved without separate layers)
- **Primary Screenshot:** This shot will be used on your application product page in the App Store.

  REMEMBER Apple doesn't want you to include the iPad status bar in your screenshot.

  Up to four additional optional screenshots can appear on the application product page. These may be resized by Apple to fit the space provided. Follow the same requirements from the preceding list.

  TIP To take a screenshot on an iPad, quickly press and release the Sleep/Wake and Home buttons at the same time. The screen flashes (and if your volume is up, you can hear a shutter click). This flash indicates that the screen was saved in the Saved Images album — choose the album in the Photos app to see the image. You can take as many screenshots as you like. The next time you sync your iPad, your photo application (such as iPhoto) launches to receive these new images.

  TIP You can also capture a screenshot using the Xcode Organizer window. Open Xcode and choose Window⇨Organizer. Plug in your iPad, and in a few seconds, it should appear in the list of devices on the left. Click the Screenshot tab at the top of the Organizer window (which appears only when the iPad is connected). Use the iPad to show the screen you want to capture and then click the Capture button. To make that screenshot your application's default image, click Save As Default Image. To get a PNG file of the screenshot, drag it to the Desktop.

- **Additional Artwork:** (Optional) If you're really lucky — I mean *really* lucky (or that good) — you may be included on featured pages in the App Store. Apple will want "high-quality layered artwork with a title treatment for your application," which will then be used in small banners to feature your app in the App Store.

Uploading your app and its data

After you've set the wheels of commerce in motion, you can then go back to Xcode to create an archive to submit to the App Store. (An archive is a compressed package of all the app's files that is simpler to upload.) Choose Build⇨Build and Archive. Your new application archive appears in the Archived Applications list in the Organizer. (Choose Window⇨Organizer to see it.) Each archive is identified with the date and time it was created. Select the application archive you want to submit and click Submit Application to iTunes Connect. In the dialog that appears, enter your developer name and password and click Submit.

You can then jump to the iTunes Connect main page and manage your app and data. Click the Manage Your Applications link (refer to Figure 6-14) to call up the Manage Your Applications page. On that page, click the new application's icon and go to town. Fill in all the blanks, using all that info I ask you to collect in the "Adding the metadata and artwork" section earlier in this chapter.

TIP

Click the Download the Developer Guide link at the bottom of the iTunes Connect main page (refer to Figure 6-14) to obtain comprehensive information about submitting apps to the App Store. You also get a ton of great information in the App Store Resource Center (`http://developer.apple.com/appstore`).

TIP

Apple also offers Application Loader, a Mac application that analyzes your app's archive file and verifies all the certificates and icons before uploading your app. To download it, scroll down the Manage Your Applications page to the bottom and then click the Download Application Loader link. To use it, go ahead with the process of adding a new app to the App Store, but when iTunes Connect asks you to upload your app, select the Check Here to Upload Your Binary Later check box. Complete the rest of the information required for uploading an app. Then, to begin uploading, start Application Loader and choose File⇨New. You're asked to log in, and if all is well, you see a dialog with a drop-down menu of all the apps that iTunes Connect recognizes that were selected above to upload the binary later. Follow the instructions to upload.

REMEMBER

The Manage Your Applications page is your one-stop center for keeping tabs on all your app creations. To edit your app's information, manage in-app purchasing, remove the app from the store, or update the app, click the app's icon to see your choices. Click View Details to see your app's information and metadata and then click Edit to edit the information. Click the Crash Reports, App Details, or View in App Store link to see more information about your app. You can also click App Summary to return to the app management page and then click Add Version to submit an update to your app.

Avoiding the App Store Rejection Slip

Apple is very strict about the App Store. For example, the first time co-author Neal submitted his ReturnMeTo iPhone app from his other book, *iPhone Application Development For Dummies,* he received a polite, but firm, e-mail rejecting the application because the app's icon used an iPhone image. You may not think this is such a big deal, but it certainly is with Apple. The artwork you use for the app icon is just one of many pieces of information that must be submitted in advance and is subject to Apple's approval.

During the opening Worldwide Developer Conference (WWDC) keynote address in June of 2010, Apple CEO Steve Jobs mentioned that there were over 225,000 applications available. (There are close to 300,000 as of this writing.) He also said that 15,000 applications are submitted per week, and that 95 percent of all apps submitted are approved within one week.

That's a pretty high approval rate. But just to be sure, it wouldn't hurt to know what was up with the 5 percent that *did* get rejected. It turns out the majority of those rejected were rejected for one of the following reasons:

- The app doesn't function as advertised by the developer.
- The app uses private frameworks or APIs (application programming interfaces).
- The app crashes.

Sounds reasonable, but in addition to the Big Three, there are a few other reasons why apps are rejected. Besides the typical rejections (mostly for bugs or for improper use of artwork or trademarks in the app), some apps have been rejected for pornographic images, and some were rejected for being too similar to Apple's own apps, while others fell into some gray area that Apple hadn't anticipated — for instance, apps that help people cheat at gambling in casinos.

For guidelines on how to avoid rejection due to coding or user interface issues, see Chapter 2. For the complete guide to avoiding rejection, see Apple's posted Guidelines for the App Store (you must be logged in as a developer):

```
https://developer.apple.com/appstore/guidelines.html
```

Before you upload your app and its data, make sure you haven't run afoul of any of Apple's rules about trademarks, copyrights, and artwork. Be sure to peruse Apple's posted Guidelines for Using Apple's Trademarks and Copyrights, which you can find here:

```
www.apple.com/legal/trademark/guidelinesfor3rdparties.html
```

Here are some tips:

- **Use the same icon for the app (the bundle icon) and the App Store page icon.** Make sure the 72 x 72 pixel icon for your iPad app, and the 48 x 48 pixel version for Spotlight search and Settings, is the same image as the 512 x 512 pixel version for your App Store page.
- **Icons must be different for *lite* and *pro* versions (such as free and paid versions).** Use a different icon image for the app and page for a lite version than the one you use for the pro version. Using the same icon image for both sends your app straight to the rejection bin.
- **Don't use any part of an Apple image and certainly none of the company's trademark images or names.** Your app can't include any photos or illustrations of the iPad, including icons that resemble the iPad, or any other Apple products (including the Apple logo itself). I've heard of projects being rejected for using the Bonjour logo, as well as Apple's network icon (the little picture of the globe with all the glowing lines). Your app can't include the word *iPad* in its title (although the iPhone app title *Tony's Tips for iPhone Users* is okay because the app's content is about the iPhone), and its use in the title or description of any components or features is very strict and probably not worth the trouble.
- **If you use any of Apple's user interface graphics, you must use them in the way they were intended.** For example, the blue + button should be used only to add an item to a list.
- **Don't infringe on other trademarks, either.** Your app's title, description, and content must not potentially infringe upon other non-Apple

trademarks or product likenesses. I've heard of an app rejected for using an icon resembling Polaroid photos.

- **Keywords can get you in trouble.** Keyword terms must be related to your app's content. It should be obvious, but some developers do it: You can't use offensive terms. And it's a big no-no to refer to other apps, competitive or not.
- **Don't include pricing information in your app's description and release notes.** Your app's marketing text — the application description and release notes — should not include pricing information, mostly because it would cause confusion in other countries due to pricing differences.
- **Don't mention Steve.** Apple will reject any app that mentions Steve Jobs in any context, even as a clue in a puzzle — it does not matter how trivial the reference; just the name is enough.
- **Don't try to fool the ratings.** Apps are rated accordingly for the highest (meaning most adult) level of content that the user is able to access. If you hide it, they will find it, and if Apple's review indicates that the app's content is in any way inconsistent with the information you provided, out you go!

Now What?

You wait for your app's approval or rejection. The timeframe is, on average, about two weeks, though some developers have claimed both shorter and much longer timeframes, and I can attest to it taking only a week for my app.

So it varies, but if you follow my advice about submitting your app, in the section "Avoiding the App Store Rejection Slip" in this chapter, and if you take my advice in Chapter 2 about development and user interface practices to avoid, it shouldn't take longer than a few weeks. Use the time wisely to set up your marketing campaigns, as I describe in Chapter 3.

Finally, at what may seem at long last (although it's really been only a few chapters), you're ready to look behind the screen and see exactly how an iPad app works. So take a break if you need to, but come back ready to explore Part III.

Part III

Understanding How Apps Work

The 5th Wave By Rich Tennant

"Hold on Barbara. I'm pretty sure there's an app for this."

In this part . . .

This part, although short, offers a peek behind the curtain of the great and powerful iOS (the operating system for the iPad, that is — and Toto, we're not in Kansas anymore). Your app, in a sense, becomes the Wizard of iOS, conjuring up blazing content and performing amazing tricks on the iPad.

Beware! The secrets described herein are not for the uninitiated; one must embark on a soul-searching journey through Part I of this book to discover the True Meaning of the Killer iPad App, and then one must register with an oath of confidentiality, join the cadre of iOS developers, and train with the SDK tools that perform the alchemy of app development, as described in Part II. At some point in this quest, you may experience the rapture of what your killer app might be, in which case you are ready to read the following:

- Chapter 7 explains the SDK frameworks for iOS that form the raw material of your iPad app (which you then refine with your code and user interface objects), and it explains the design patterns that you should adopt to make use of these frameworks. It also shows how windows, views, and view controllers work on the iPad.
- Chapter 8 describes in detail the (possibly) short and happy life of an iPad app, from launch to termination. You see how an application object is created and connected to the window object. You get to know all about the event loop and how it all starts with the main nib file, which you can select in Xcode and look at in Interface Builder. When you finish, you should have enough information to get started coding your app.

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 — 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 — 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 — 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 — the content as well as the controls and views that enable the user to access and use that content — *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 — 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 — you get the picture

TIP

Some developers talk in terms of "using a framework" — 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 — 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 — especially `UIKit` — 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 — or (better put) how the framework objects want to use *your code* best — 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.

TECHNICAL STUFF

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 — 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.

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 — so far — 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 — 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 — 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 — 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 16.)

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 — 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 — either the appropriate page from the Web site or the `Weather is not available` offline message.

All this is illustrated in Figure 7-1.

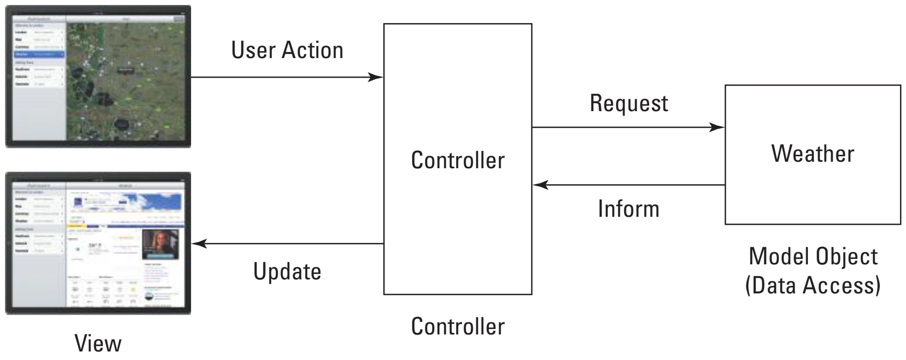


Figure 7-1: Models, controllers, and views.

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 — windows, views, and view controllers.

Working with Windows and Views

iPad apps have only a single window. When your application is running — even though other apps may be hibernating or running in the background — 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 — 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.

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.

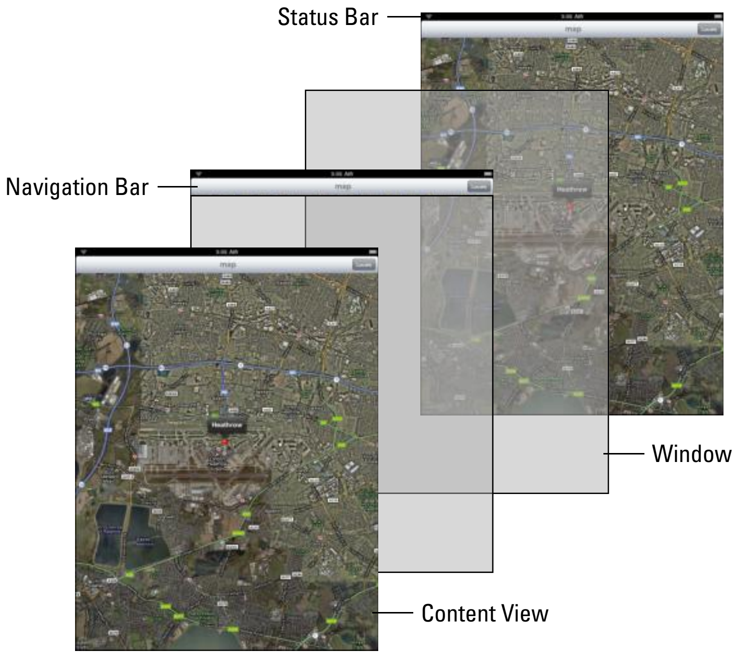


Figure 7-2: The iPad Travel411 app window layout.

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 appears between the upper and lower bars shown in Figure 7-2.

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, they make 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 — "A Content View," with A, B, and C subviews.

Controls — such as buttons, text fields, and the like — are really view subclasses that become subviews, as 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.

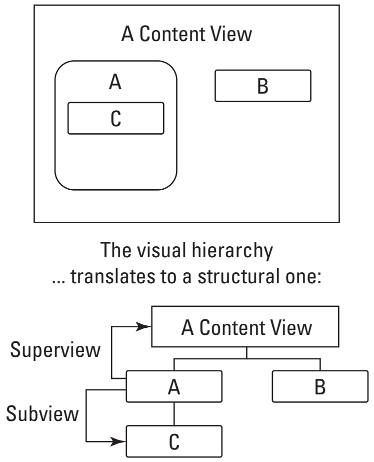


Figure 7-3: The view hierarchy is both visual and structural.

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 — 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 — like you do in the DeepThoughts app — 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 — 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 16, 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 — 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:

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 — including creating them or flushing them from memory during low-memory situations
- Responding to a change in the device's orientation — say, landscape to portrait — 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

  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?").

Apple recommends that apps should support all iPad landscape and portrait orientations when appropriate — 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 — 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.

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.

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.

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 — 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 — 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 — usually a view controller — 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 — a button, say — 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`.

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 — as this chapter explains — 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/iphone applicationdevelopment`.

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 — 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

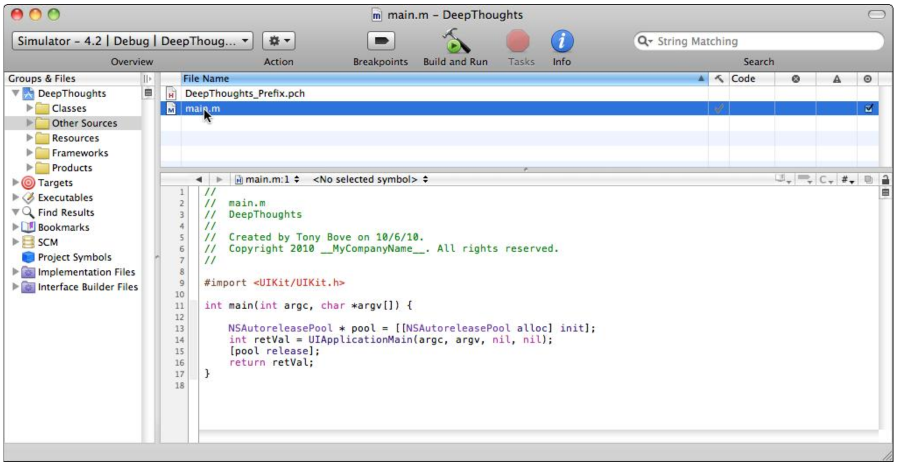
When you create an Xcode project and select a template, as I show in Chapter 5, you get a considerable head start on the process of coding your very own iPad app. In that chapter, I choose the View-based Application template for the DeepThoughts app, and as a result, I have a working app that offers 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 — the content as well as the controls and views that enable the user to access and use that content — *to* the frameworks.

App Anatomy 101 — 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.



```
//
//  main.m
//  DeepThoughts
//
//  Created by Tony Bove on 10/6/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {

    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

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.**

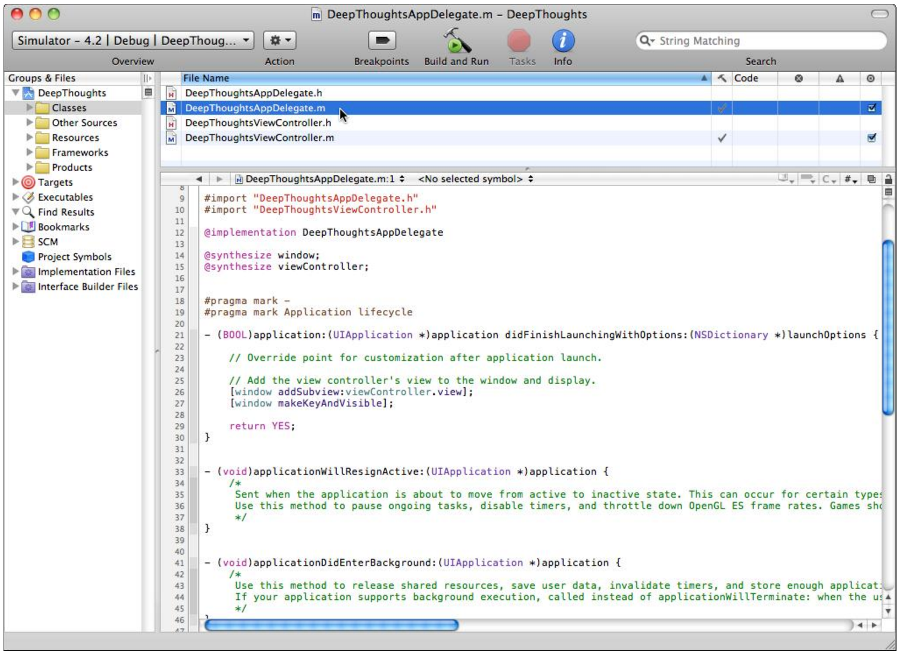
   A *nib file* is a resource file that contains the specifications for one or more objects. The main nib file usually contains a window object of some kind, the application delegate object, and any other key objects. When the file is loaded, the objects are reconstituted (think "instant application") in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



```
#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 application launch.

    // Add the view controller's view to the 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
     Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates. Games sho
     */
}


- (void)applicationDidEnterBackground:(UIApplication *)application {
    /*
     Use this method to release shared resources, save user data, invalidate timers, and store enough applicat
     If your application supports background execution, called instead of applicationWillTerminate: when the us
     */
```

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 — the screen, for example. Events, such as touches on the screen, are sent to the object — say, a controller — 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 — for example, an incoming FaceTime call, calendar alerts, or the user pressing the Sleep/Wake button — 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* — 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 — 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.)

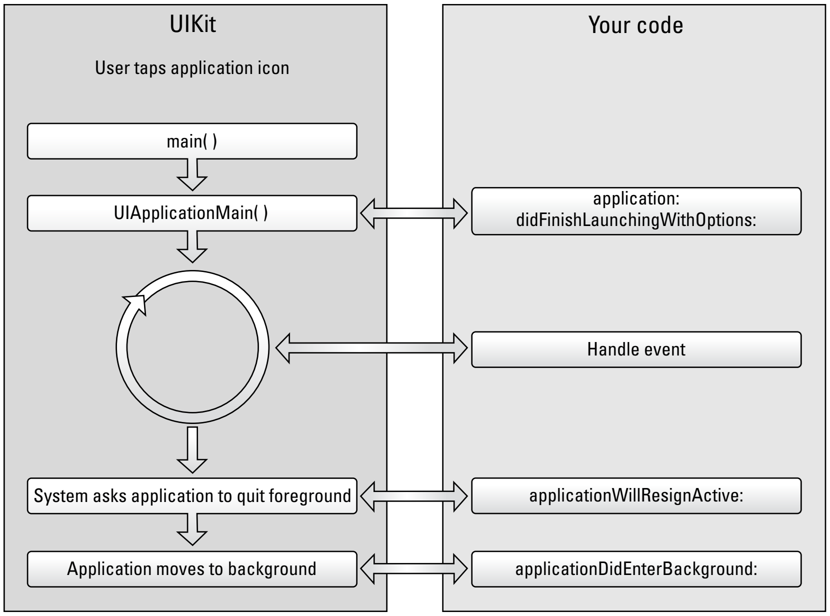

UIKit
Your code
User taps application icon
main( )
UIApplicationMain( )
application:
didFinishLaunchingWithOptions:
Handle event
System asks application to quit foreground
applicationWillResignActive:
Application moves to background
applicationDidEnterBackground:

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 — quite the normal state of affairs, as I show in Chapter 5 — 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 — despite the fact that you haven't done a lick of coding.

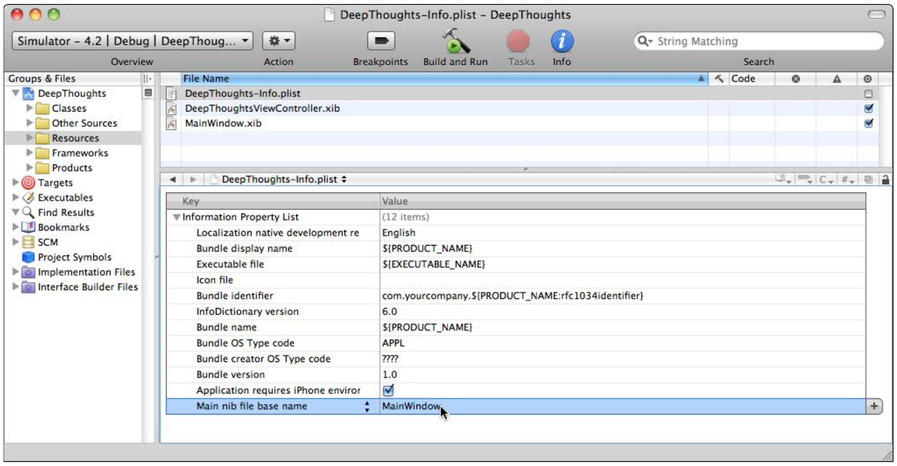
Most of this work is done by the `UIApplicationMain` function, as illustrated back in Figure 8-3. To take advantage of this once-in-a-lifetime opportunity to see how all this works, go back to your project window in Xcode, which you started by choosing the View-based Application template in Chapter 5, 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.)



Key	Value
Information Property List	(12 items)
Localization native development re	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.yourcompany.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
Application requires iPhone enviror	✓
Main nib file base name	MainWindow

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

3. **`UIApplication` loads `MainWindow.xib`.**

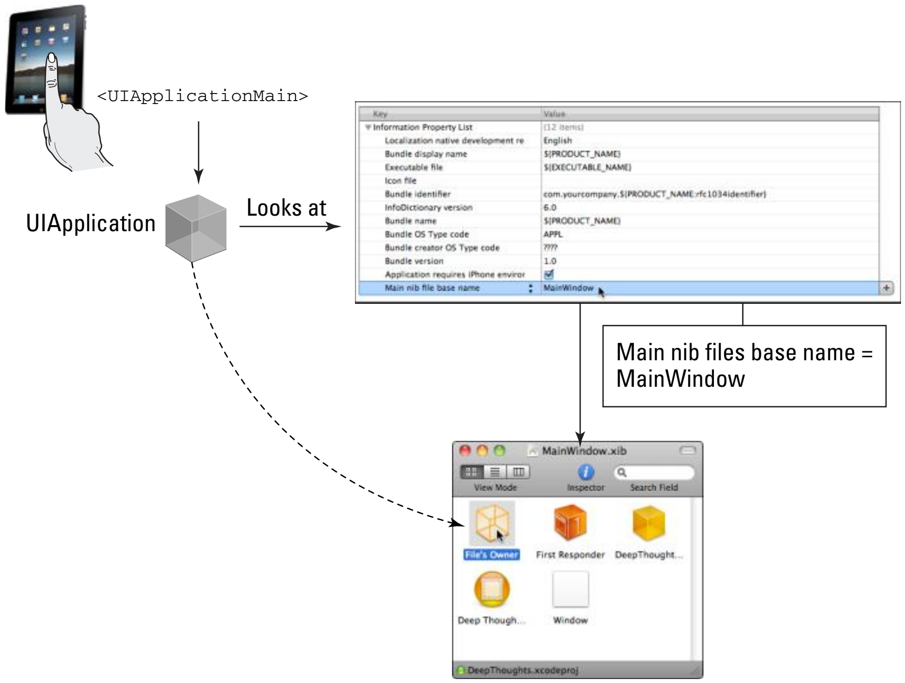
Figure 8-5 illustrates this process of loading the main window's nib file.

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.)

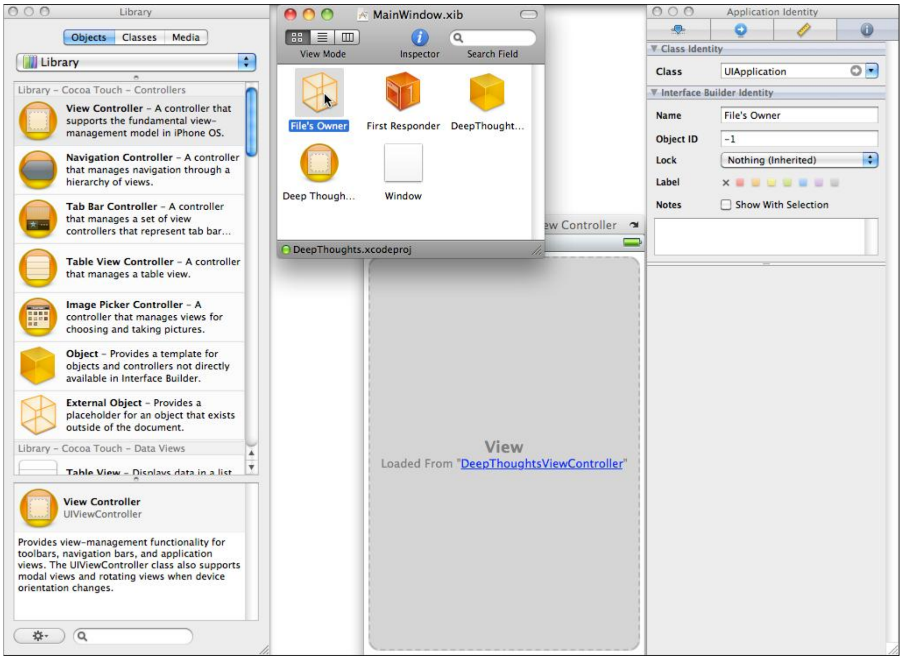
When Interface Builder opens, take a look at the nib file's main window — the one labeled `MainWindow.xib` (as shown in Figure 8-6). Select File's Owner in the window, click the *i* Inspector button at the top of the window, and then click the *i* Identity tab of the Inspector window if it's not already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

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:



Key	Value
Information Property List	(12 items)
Localization native development re	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.yourcompany.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
Application requires iPhone enviror	✓
Main nib file base name	MainWindow

Figure 8-5: The application is launched.



Library – Cocoa Touch – Controllers

View Controller – A controller that supports the fundamental view-management model in iPhone OS.

Navigation Controller – A controller that manages navigation through a hierarchy of views.

Tab Bar Controller – A controller that manages a set of view controllers that represent tab bar…

Table View Controller – A controller that manages a table view.

Image Picker Controller – A controller that manages views for choosing and taking pictures.

Object – Provides a template for objects and controllers not directly available in Interface Builder.

External Object – Provides a placeholder for an object that exists outside of the document.

Library – Cocoa Touch – Data Views

Table View – Displays data in a list

View Controller
UIViewController

Provides view-management functionality for toolbars, navigation bars, and application views. The UIViewController class also supports modal views and rotating views when device orientation changes.

Class Identity

Class	UIApplication

Interface Builder Identity

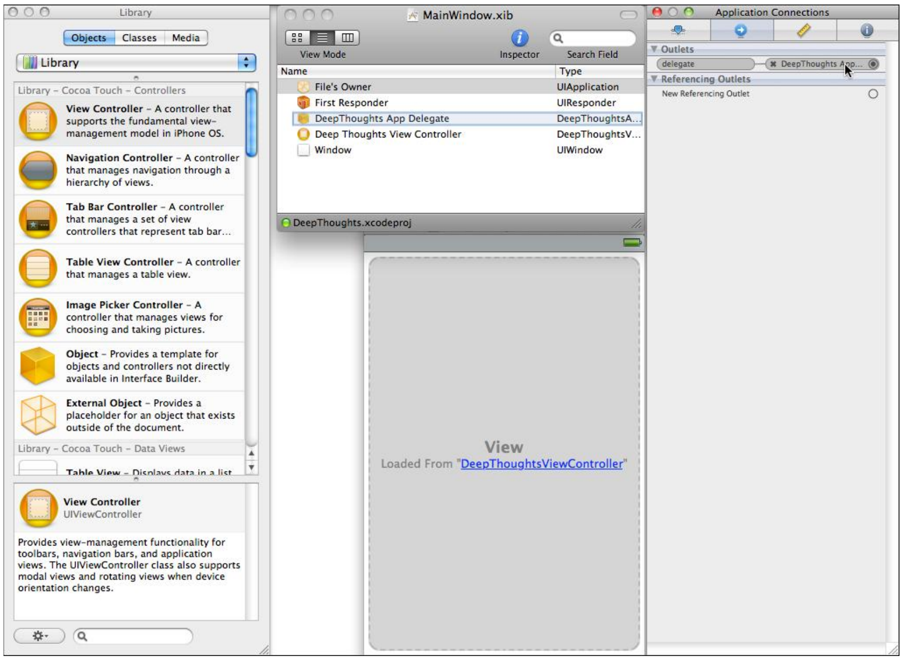
Name	File's Owner
Object ID	-1
Lock	Nothing (Inherited)
Label	
Notes	☐ Show With Selection

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

- **File's Owner (proxy object):** The File's Owner — the object that's going to use (or *own*) this file — is of the class `UIApplication`. This object isn't created when the file is loaded, as are the window and views — 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 — 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 — 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.



View Controller – A controller that supports the fundamental view-management model in iPhone OS.

Navigation Controller – A controller that manages navigation through a hierarchy of views.

Tab Bar Controller – A controller that manages a set of view controllers that represent tab bar...

Table View Controller – A controller that manages a table view.

Image Picker Controller – A controller that manages views for choosing and taking pictures.

Object – Provides a template for objects and controllers not directly available in Interface Builder.

External Object – Provides a placeholder for an object that exists outside of the document.

Table View – Displays data in a list

View Controller
UIViewController

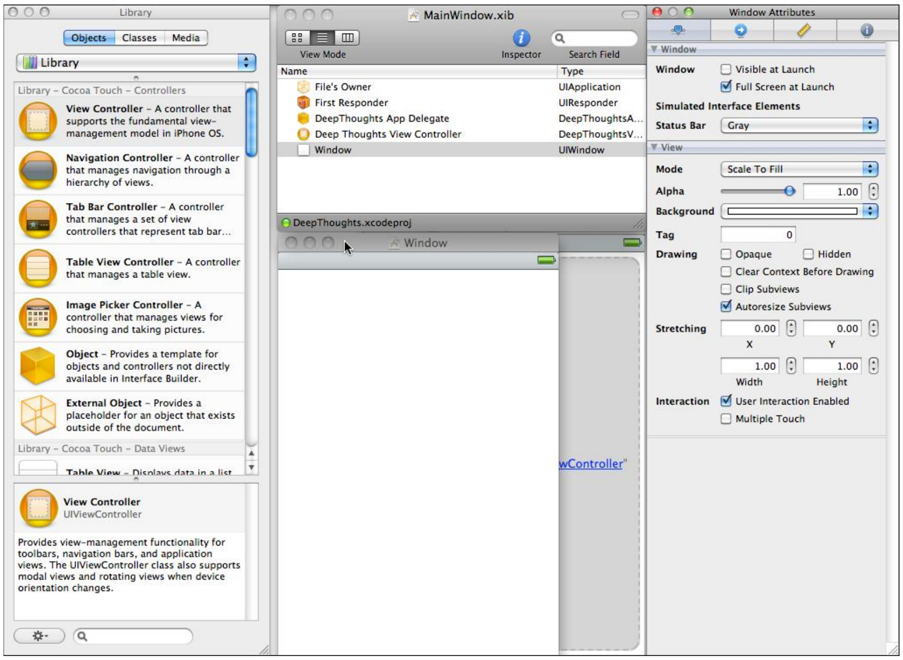
Provides view-management functionality for toolbars, navigation bars, and application views. The UIViewController class also supports modal views and rotating views when device orientation changes.

Name	Type
File's Owner	UIApplication
First Responder	UIResponder
DeepThoughts App Delegate	DeepThoughtsA...
Deep Thoughts View Controller	DeepThoughtsV...
Window	UIWindow

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

- **Window:** The window has its background set to white and its status bar set to gray, as shown in Figure 8-8, and it 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 Listings 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`.



View Controller – A controller that supports the fundamental view-management model in iPhone OS.

Navigation Controller – A controller that manages navigation through a hierarchy of views.

Tab Bar Controller – A controller that manages a set of view controllers that represent tab bar...

Table View Controller – A controller that manages a table view.

Image Picker Controller – A controller that manages views for choosing and taking pictures.

Object – Provides a template for objects and controllers not directly available in Interface Builder.

External Object – Provides a placeholder for an object that exists outside of the document.

Library – Cocoa Touch – Data Views

Table View – Displays data in a list

View Controller
UIViewController

Provides view-management functionality for toolbars, navigation bars, and application views. The UIViewController class also supports modal views and rotating views when device orientation changes.

Name	Type
File's Owner	UIApplication
First Responder	UIResponder
DeepThoughts App Delegate	DeepThoughtsA...
Deep Thoughts View Controller	DeepThoughtsV...
Window	UIWindow

Window
Window: ☐ Visible at Launch ☑ Full Screen at Launch
Simulated Interface Elements
Status Bar: Gray
View
Mode: Scale To Fill
Alpha: 1.00
Background:
Tag: 0
Drawing: ☐ Opaque ☐ Hidden ☐ Clear Context Before Drawing ☐ Clip Subviews ☑ Autoresize Subviews
Stretching: X 0.00 Y 0.00 Width 1.00 Height 1.00
Interaction: ☑ User Interaction Enabled ☐ Multiple Touch

Figure 8-8: The Main Window.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 didF
          inishLaunchingWithOptions:(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
```

(continued)

Listing 1-2 *(continued)*

```
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 `applicationDid FinishLaunchingWithOptions` 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.

Your goal during startup should be to present your application's user interface as quickly as possible — 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.

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 — 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 — placed into, in other words — 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.

   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 — 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 — 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 — 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 — 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) — 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 re-created when the app relaunches. If memory becomes constrained, iOS may purge background apps to make more room for the foreground app.

Because these interruptions cause a temporary loss of control by your app — meaning that touch events, as I describe in Chapter 11, are no longer sent to your app — 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 — 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 — if you've paused a game, for example — 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 — or the system launches another application — 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 — 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 additional 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 — 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 the 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 — 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 — 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 — 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 — 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 — 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:` — 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 — as you will be doing, if you stay as up-to-date as me — 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 — 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 — 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 — or even other view controllers — 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 about 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 your app 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 — it can diminish performance by reducing the amount of available memory.

Avoiding the warnings

When you create an object — a window or button for example — 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* — 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 — 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.

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 — 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 — in the previous 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.

Although there's a lot left unsaid (though less than you might suspect), the details in the previous chapter and this chapter are enough not only to get you started but also to keep you going as you develop your own iPad apps. These chapters provide a frame of reference on which you can hang the concepts I throw around with abandon in upcoming chapters — as well as the groundwork for a deep enough understanding of the application lifecycle to give you a handle on the detailed documentation.

Time to move on to the really fun stuff: building DeepThoughts into an app that actually does something.

Part IV
Building DeepThoughts

"It's a docking system for the iPad that comes with 3 bedrooms, 2 baths, and a car port."

In this part . . .

To wrap your head around the entire process of building an app, I present to you DeepThoughts, a sample app, which you build in this part. It's simple enough to understand, and yet, it demonstrates enough of the building blocks for creating a sophisticated iPad app that you should pay attention to these chapters:

- ✓ Chapter 9 takes you on a tour of the View-based Application template, on which DeepThoughts is based. You also add an image to your first iPad view-based app, and you add an interface element (an Info button). You also do one of the more important graphical tasks you need to do when you build an app: supply an icon for the app for the iPad display.

- ✓ Chapter 10 dives right into custom-coding your app. You find out how to use Xcode's documentation and help windows while adding code, and you learn the answer to that monumental question: Where does my code go? The code you add controls and animates the view and sets up the methods and variables for applying user preference settings.

- ✓ Chapter 11 gets you right into the thick of iPad development, creating a modal view for uses to change their preference settings, connecting interface objects such as sliders and text entry fields, and adding recognition for tap and swipe gestures.

- ✓ Chapter 12 shows you how to swat the bugs in your apps using the Debugger, the mini-debugger, the Console, and even the Static Analyzer. You learn all about setting breakpoints that stop your app cold in the Simulator. At the end of this chapter, DeepThoughts is actually finished, and you know how to debug apps with Xcode.

Chapter 9

Building the User Interface

In This Chapter

- Inspecting an app with a view
- Using Interface Builder to add graphics and buttons
- Adding an application icon

Steve Jobs said it best: “Design is not just what it looks like and feels like. Design is how it works.” That’s why you should know how an iPad app works before trying to design a user interface for one.

For one thing, you need to consider the memory limitations and display orientation of the iPad. That’s why the Xcode templates are so useful — they take care of the display and memory management so that you can focus on what your app can do. After seeing how easy it is to add graphics and interface elements to the template-based project, you may think the user interface for your app will be a piece of cake — and to some extent, it probably will be, thanks to Interface Builder.

The template you select for your Xcode project (as I show in Chapter 5) provides the skeleton of a user interface. For example, the View-based Application template for the DeepThoughts app offers a view and a view controller that you can customize. Other templates offer rudimentary interface objects — for example, the Navigation-based Application template offers a Navigation controller, and the Utility-based Application template offers a Flipside view that a user opens by tapping an Info button and closes by tapping a Done button. The Split View-based Application template offers a Split view controller as well as the two view controllers you’d use to manage a master-detail-style display.

Make sure you choose the appropriate template so that you don’t have to reinvent the wheel. And before you start coding, examine how the template’s interface works and how to add your custom interface objects and graphics. That’s what this chapter is all about.

Running the View-Based Application Template

When you start a project with the View-based Application template, you get the Main window, a view (using a white background) scaled to fill the entire Main window, and a black status bar at the top. The view and status bar automatically change orientation for you when the user rotates the iPad.

You can see this in action in the Simulator, even before writing any code — after choosing the template to create your project (as I describe in Chapter 5), build and run the project by choosing Build➪Build and Run from the Xcode main menu. (I also show you how to build, run, and use the Simulator in Chapter 5.)

After a user launches your app, the functionality provided in the `UIKit` framework manages most of the application's infrastructure. Part of the initialization process mentioned in Chapter 8 involves setting up the main run loop and event-handling code, which is the responsibility of the `UIApplication` object. When the application is onscreen, it's driven by external events, such as stubby fingers touching sleek buttons.

As you discover in Chapter 8, the nib file `MainWindow.xib` causes the application's delegate, window, and view controller instances to get created at runtime, and both `MainWindow.xib` and the view controller's nib file, `DeepThoughtsViewController.xib`, are provided as part of the View-based Application template. An instance of `DeepThoughtsViewController` is set to be the application's view controller, and that's where you put your code to control the view.

Before doing that, however, you can build this view to have a background image and some interface elements. To see what you have in the view now, you can inspect the view in Interface Builder.

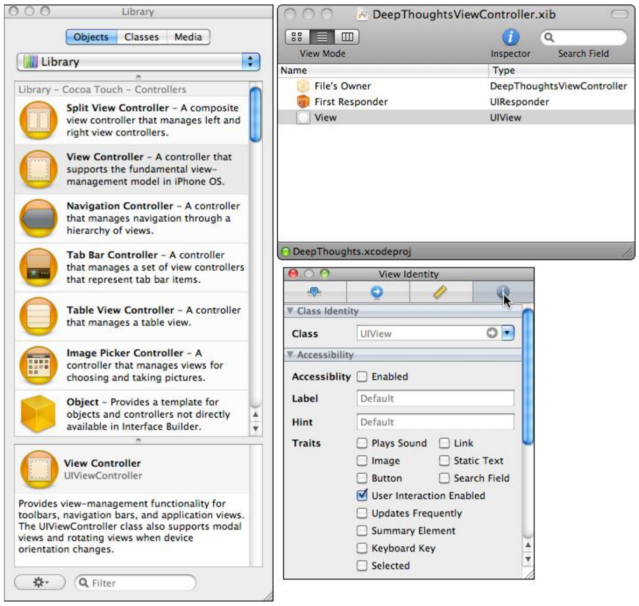
Inspecting the View

To see how the view is created and connected to the template code, start up Interface Builder from Xcode by first clicking the Resources folder in the Groups & Files list (refer to Chapter 5) and then double-clicking the `DeepThoughtsViewController.xib` file to launch Interface Builder.

To inspect the view, click the Identity tab of the Inspector window (or choose Tools➪Identity Inspector) and then click the View icon in the `DeepThoughtsViewController.xib` window to see the identity of the view, as shown in Figure 9-1.

TIP

The four icons across the top of the Inspector window from left to right correspond to the Attributes, Connections, Size, and Identity Inspectors, respectively, in the Tools menu.



Split View Controller – A composite view controller that manages left and right view controllers.

View Controller – A controller that supports the fundamental view-management model in iPhone OS.

Navigation Controller – A controller that manages navigation through a hierarchy of views.

Tab Bar Controller – A controller that manages a set of view controllers that represent tab bar items.

Table View Controller – A controller that manages a table view.

Image Picker Controller – A controller that manages views for choosing and taking pictures.

Object – Provides a template for objects and controllers not directly available in Interface Builder.

View Controller
UIViewController

Provides view-management functionality for toolbars, navigation bars, and application views. The UIViewController class also supports modal views and rotating views when device orientation changes.

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

Class: UIView

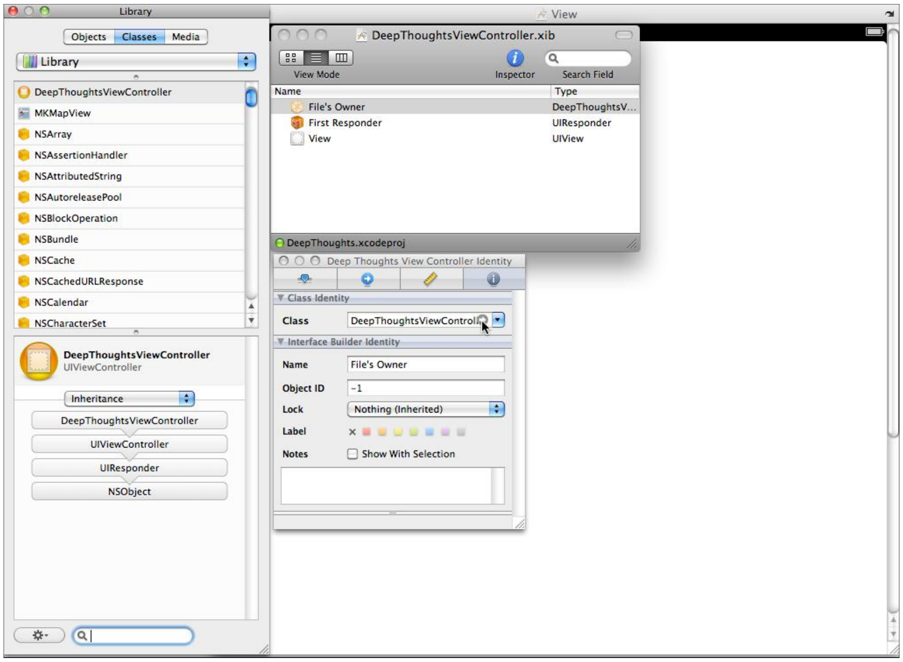
Accessibility: ☐ Enabled
Label: Default
Hint: Default
Traits: ☐ Plays Sound ☐ Link ☐ Image ☐ Static Text ☐ Button ☐ Search Field ✓ User Interaction Enabled ☐ Updates Frequently ☐ Summary Element ☐ Keyboard Key ☐ Selected

Figure 9-1: The view's identity in Interface Builder.

You can see that the view belongs to the `UIView` class, and that user interaction has been enabled. `UIView` is an abstract superclass that provides concrete subclasses with a structure for drawing and handling events. The `UIView` class provides common methods you can use to create all types of views as well as access their properties.

You can also click the Attributes tab of the Inspector window (or choose Tools➪Attributes Inspector) to see the view's attributes — and you find that it includes a black status bar and the default (white) background.

To get info about the class that uses, or *owns*, this view, click File's Owner in the `DeepThoughtsViewController.xib` window and then click the Identity tab of the Inspector window (or choose Tools➪Identity Inspector). In the Class pop-up menu, you can see that the File's Owner — the object that's going to use (or *own*) this file — is set to the class `DeepThoughtsViewController` (as shown in Figure 9-2). Click the circled arrow next to `DeepThoughtsViewController` in the Class pop-up menu to display the class in the Library window along with its description. (Refer to Figure 9-2.)



Name	Type
File's Owner	DeepThoughtsV...
First Responder	UIResponder
View	UIView

Class: DeepThoughtsViewControll
Name: File's Owner
Object ID: -1
Lock: Nothing (Inherited)
Label
Notes

Figure 9-2: The File's Owner is the view controller.

How does this work? The template sets this all up for you, without you having to lift a finger.

Understanding How the View is Initialized

If you look back to Chapter 8, you can see that the template supplies the following code in `DeepThoughtsAppDelegate.h`:

```
@class DeepThoughtsViewController;
@interface DeepThoughtsAppDelegate : NSObject
           <UIApplicationDelegate> {
    UIWindow *window;
 DeepThoughtsViewController *viewController;
}
```

This sets up the `UIApplicationDelegate` protocol with `window`. The template also declares an accessor method for `window` and tags it with an `IBOutlet` (so that Interface Builder can discover it) while also declaring an accessor method for `viewController`:

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

In the file `DeepThoughtsAppDelegate.m`, the `@synthesize` statements tell the compiler to create accessor methods for you — one for each `@property` declaration (`window` and `viewController`). After the delegate receives notification that the application has launched in the `application:didFinishLaunchingWithOptions:` method, the code uses the `addSubview` and `makeKeyAndVisible` methods to display the view:

```
@implementation DeepThoughtsAppDelegate
@synthesize window;
@synthesize viewController;

- (BOOL)application:(UIApplication *)application didF
          inishLaunchingWithOptions:(NSDictionary *)
          launchOptions {

    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];

        return YES;
}
```

The view controller is initialized, and `addSubView` adds `viewController.view` to `window` in order to display the view. Calling `makeKeyAndVisible` on `window` makes the window visible as well as making it the main window and the first responder for events (touches).

In `DeepThoughtsViewController.h`, you find this:

```
@interface DeepThoughtsViewController : UIViewController {
}
```

This tells you that `DeepThoughtsViewController` is a subclass of `UIViewController`. The `UIViewController` class provides the fundamental view-management model for iPad apps. You use each instance of `UIViewController` to manage a full-screen view.

In `DeepThoughtsViewController.m` near the top, you find commented-out code you can use to set up custom view initialization (though I don't need it for DeepThoughts):

```
/*
// The designated initializer. Override to perform setup
          that is required before the view is loaded.
- (id)initWithNibName:(NSString *)nibNameOrNil
          bundle:(NSBundle *)nibBundleOrNil {
```

```
    if ((self = [super initWithNibName:nibNameOrNil
            bundle:nibBundleOrNil])) {
        // Custom initialization
    }
    return self;
}
*/
```

The `DeepThoughtsViewController` object is created by a nib file directly passed on to `UIViewController` to handle the initialization. (You can add custom initialization at the point where the `// Custom initialization` comment appears.)

Adding an Image to the View

So far, the DeepThoughts app displays a white view with a black status bar — for the entire 1,024 x 768 pixels (at up to 24 bits per pixel). You need to put those pixels to work! A compelling iPad app needs to immerse the user in an experience, even if that experience is the appearance of a leather-bound address book rather than a simple contact entry form. The basis of any kind of immersive experience is the image you display in the view.

To place an image in your app, first you need the image. Although you have plenty of pixels to work with, you should create artwork for the final image in a larger multiple of the pixel dimensions you need so that you can add depth and details before scaling it down accurately to the iPad display size.

TIP

The preferred format for the image is `.png`. Although most common image formats, such as `.jpg` (JPEG) will display correctly, Xcode automatically optimizes `.png` images at build time to make them the fastest and most efficient image type for use in iPad applications.

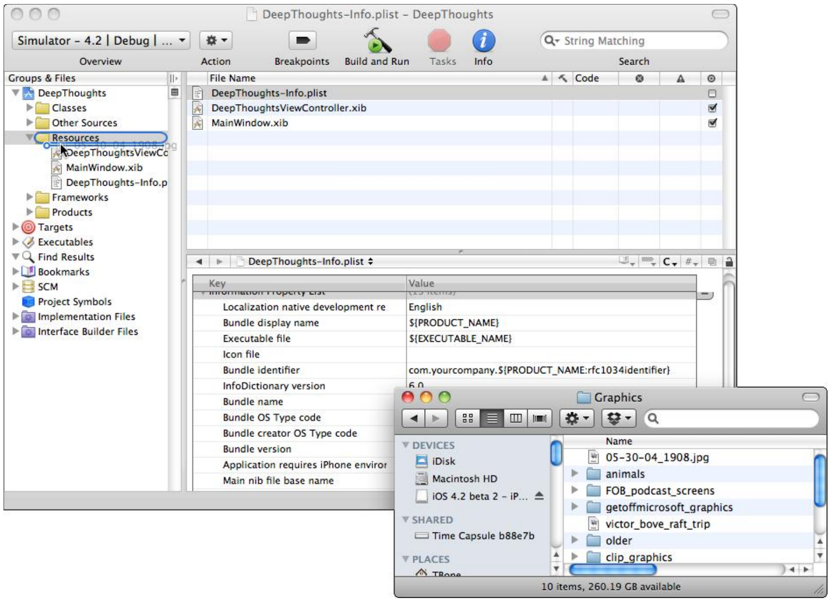
After you have your image (in my case, I'm using a JPEG image), do the following:

1. **In Xcode, drag the image file into the Resources folder in the Groups & Files list, as shown in Figure 9-3.**

   Although you can drag it to any location in the Groups & Files list, I like to keep my projects uncluttered, and I use the Resources folder to hold all interface-related files.

   An alternative is to click the Resources folder in Xcode, choose Project➪Add to Project, and then navigate to the file you want to add.

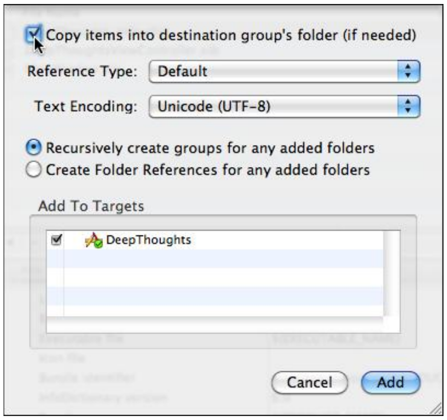
   After you drag or add the file, Xcode displays a dialog for making a copy of the file and specifying its reference type, text encoding, and other options, as shown in Figure 9-4.



Key	Value
Localization native development re	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.yourcompany.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	6.0
Bundle name	
Bundle OS Type code	
Bundle creator OS Type code	
Bundle version	
Application requires iPhone enviror	
Main nib file base name	

Figure 9-3: Drag an image file into the Resources folder.

2. **Select the Copy Items into Destination Group's Folder check box to copy the file, and then click Add to finish copying the image file into your project.**



[✓] Copy items into destination group's folder (if needed)

Reference Type: Default

Text Encoding: Unicode (UTF-8)

[✓ Recursively create groups for any added folders]
Create Folder References for any added folders

Add To Targets

[✓] DeepThoughts

Figure 9-4: Tell Xcode to make a copy of the file and add it to the project.

If you don't make a copy of the file, Xcode simply creates a pointer to the file. The advantage of using a pointer is that if you modify the image later, Xcode will use that modified version. The disadvantage is that Xcode won't be able to find the image file if you move it.

TIP

I'm all for copying. If you change the image, use the same name for the changed file and drag it back into Resources to replace the older file. Then build the project again.

You don't need to change the default settings for Reference Type, Text Encoding, or Recursively Create Groups for Any Added Folders.

3. **Double-click `DeepThoughtsViewController.xib` in the Resources folder to open Interface Builder.**

   The Interface Builder windows should appear. (Refer to Figure 9-1.) If the Library window doesn't appear with the other windows, choose Tools➪Library to show it.

4. **Click the Objects tab at the top of the Library window and select Data Views from the drop-down menu below the Objects tab.**

REMEMBER

   Selecting Objects and then Data Views narrows your search through the Library window so that you can find the object you need quickly.

   It turns out that, to place the image in the view, you need the Image View object, as shown in Figure 9-5.

5. **Drag the Image View object from the Library onto the View window.**

   If the View window is obscured by other windows, just drag the Image View object directly on top of the View icon in the `DeepThoughtsViewController.xib` window as I do in Figure 9-5, so that it appears underneath and part of the view.

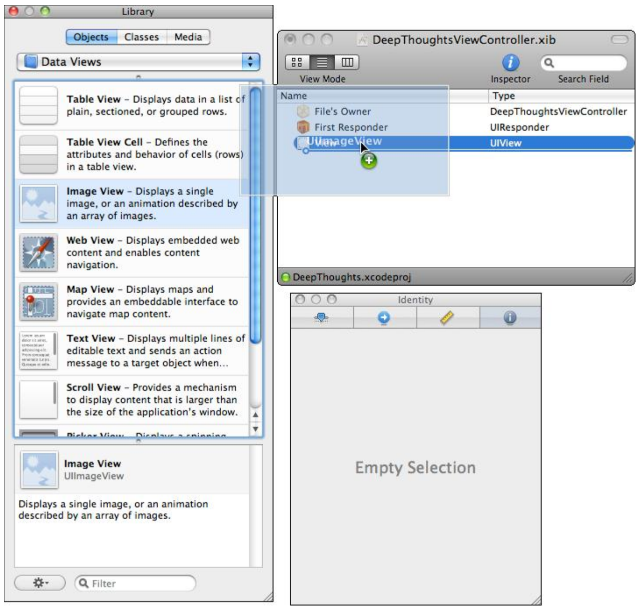


Table View – Displays data in a list of plain, sectioned, or grouped rows.
Table View Cell – Defines the attributes and behavior of cells (rows) in a table view.
Image View – Displays a single image, or an animation described by an array of images.
Web View – Displays embedded web content and enables content navigation.
Map View – Displays maps and provides an embeddable interface to navigate map content.
Text View – Displays multiple lines of editable text and sends an action message to a target object when...
Scroll View – Provides a mechanism to display content that is larger than the size of the application's window.

Displays a single image, or an animation described by an array of images.

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

Figure 9-5: Adding the Image View object to the view.

6. **Select Image View in the `DeepThoughtsViewController.xib` window and click the Attributes tab of the Inspector window (or choose Tools➪Attributes Inspector).**

   The Attributes Inspector shows the Image View attributes.

7. **In the Attributes Inspector, choose the image file from the Image drop-down menu, as shown in Figure 9-6.**

   That Inspector window's a handy little critter, isn't it? Just select the image file you dragged into your project in Step 1. Your Image View object does what it's told and loads your view so that it ends up looking something like Figure 9-7.

   TIP

   If you don't see the image file in the Image drop-down menu, choose File➪Reload All Class Files. It should appear the next time you use the Image drop-down menu.

8. **Choose File➪Save from the main menu to save your changes to the `DeepThoughtsViewController.xib` file.**

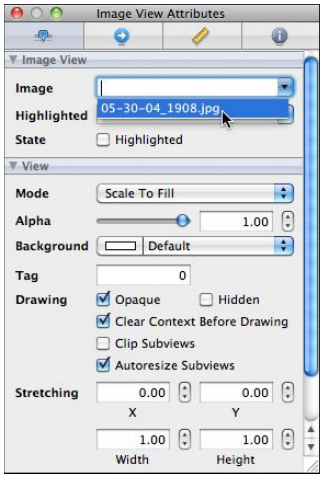


Image View Attributes

▼ Image View

Image: [05-30-04_1908.jpg]
Highlighted
State: ☐ Highlighted

▼ View

Mode: Scale To Fill
Alpha: 1.00
Background: Default
Tag: 0
Drawing: ☑ Opaque ☐ Hidden
☑ Clear Context Before Drawing
☐ Clip Subviews
☑ Autoresize Subviews
Stretching: 0.00 X, 0.00 Y; 1.00 Width, 1.00 Height

Figure 9-6: Select the image file for the Image View object.

This example uses a full-screen image. If you place images that are smaller, as I show in the next section when I place an Info button, Interface Builder gives you help placing the image in the view.

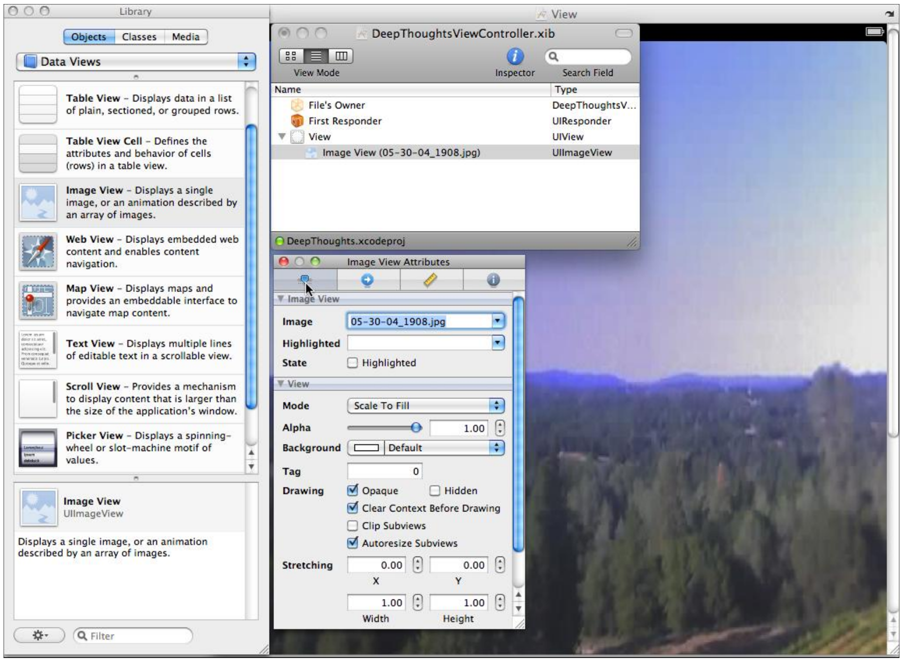


Table View – Displays data in a list of plain, sectioned, or grouped rows.

Table View Cell – Defines the attributes and behavior of cells (rows) in a table view.

Image View – Displays a single image, or an animation described by an array of images.

Web View – Displays embedded web content and enables content navigation.

Map View – Displays maps and provides an embeddable interface to navigate map content.

Text View – Displays multiple lines of editable text in a scrollable view.

Scroll View – Provides a mechanism to display content that is larger than the size of the application's window.

Picker View – Displays a spinning-wheel or slot-machine motif of values.

Image View
UIImageView
Displays a single image, or an animation described by an array of images.

Name	Type
File's Owner	DeepThoughtsV...
First Responder	UIResponder
View	UIView
Image View (05-30-04_1908.jpg)	UIImageView

Image View
Image: 05-30-04_1908.jpg
Highlighted:
State: Highlighted ☐

View
Mode: Scale To Fill
Alpha: 1.00
Background: Default
Tag: 0
Drawing: ✓ Opaque ☐ Hidden ✓ Clear Context Before Drawing ☐ Clip Subviews ✓ Autoresize Subviews
Stretching: X 0.00 Y 0.00 Width 1.00 Height 1.00

Figure 9-7: The image file now appears in the view.

Adding an Info Button

The DeepThoughts app is supposed to display an animation of text flowing down the view, and the user (after the app is finished) will be able to change the text and the speed of the animation either by tapping an Info button (which is similar to the iPhone version of this app) or by tapping the view itself. I show you how to connect the Info button to your code in Chapter 11 so it can show a modal dialog for changing the text and setting the speed, but you need to first add the Info button to the user interface, as I show here.

To add a user interface button and set it to be an Info button, follow these steps:

1. **Make your way over to the Groups & Files list and double-click `DeepThoughtsViewController.xib` in the Resources folder to open the file in Interface Builder (if it isn't already open).**

   Interface Builder should appear with the `DeepThoughtsViewController.xib` window, the Library window, the Inspector window, and the View window. (Refer to Figure 9-7.) If you followed the steps of the previous section, the Library window should already be open; if it isn't already open, choose Tools➪Library to show it.

2. **Click the Classes tab at the top of the Library window and then select `UIResponder` from the drop-down menu below the Classes tab.**

   Selecting Classes and then `UIResponder` narrows your search through the Library window so that you can find the class you need quickly. To place the Info button in the view, you need the `UIButton` class, as shown in Figure 9-8.

3. **Drag the `UIButton` class object from the Library onto the View window.**

   Again, if the View window is obscured by other windows, just drag the `UIButton` class object directly on top of the View icon in the `DeepThoughtsViewController.xib` window as I do in Figure 9-8, so that it appears underneath the image you added in the previous section (which is also part of view).

   After all this dragging, the `UIButton` class object appears under View in the `DeepThoughtsViewController.xib` window as a Rounded Rect Button, as shown in Figure 9-9, which is its default setting.

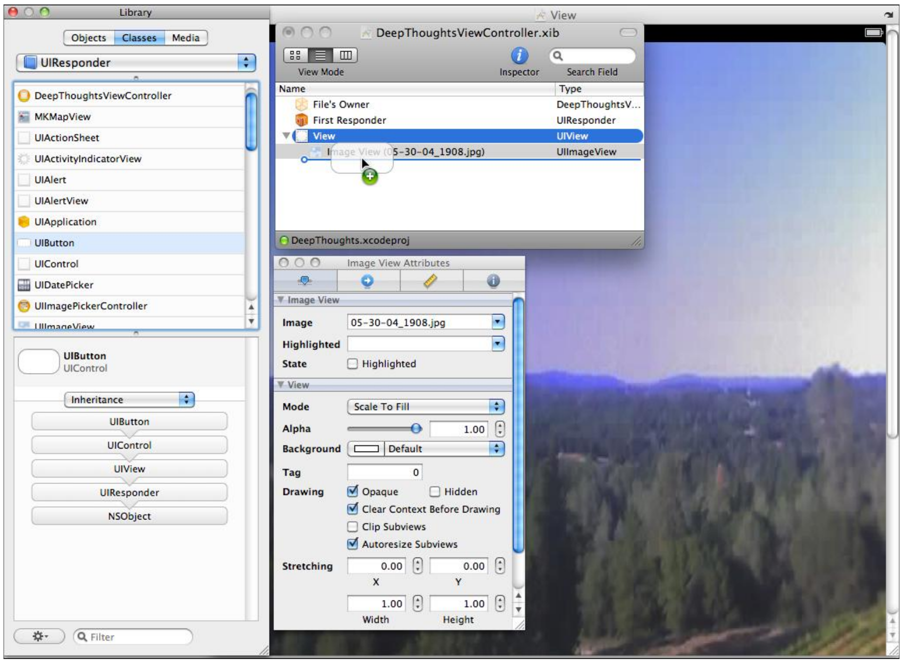
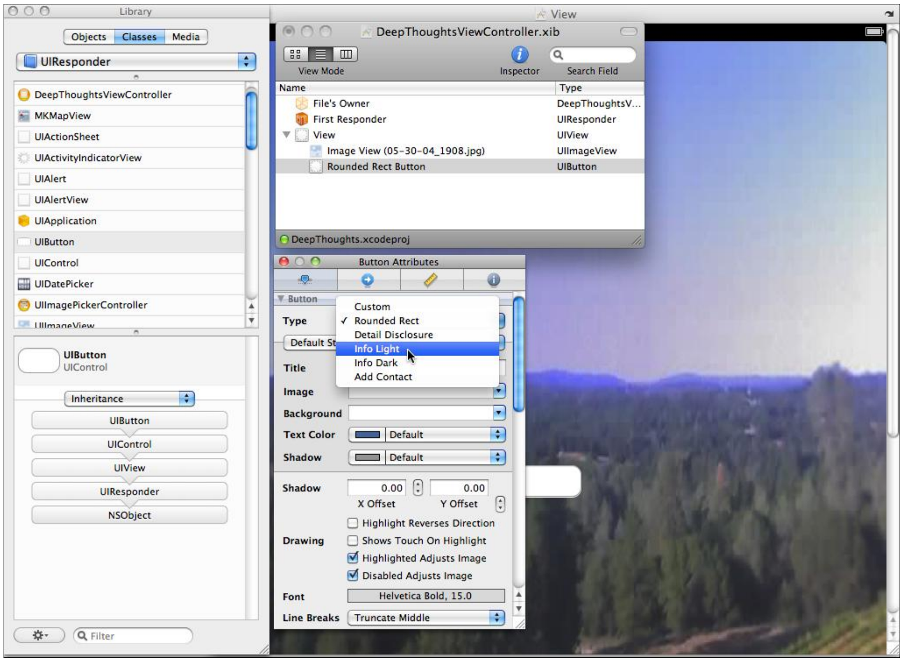


Image: 05-30-04_1908.jpg
Highlighted:
State: Highlighted ☐
Mode: Scale To Fill
Alpha: 1.00
Background: Default
Tag: 0
Drawing: ✓ Opaque ☐ Hidden
✓ Clear Context Before Drawing
☐ Clip Subviews
✓ Autoresize Subviews
Stretching: X 0.00, Y 0.00, Width 1.00, Height 1.00

Figure 9-8: Add the UIButton class object to the view.



Name	Type
File's Owner	DeepThoughtsV...
First Responder	UIResponder
View	UIView
Image View (05-30-04_1908.jpg)	UIImageView
Rounded Rect Button	UIButton

Type: ✓ Rounded Rect
Text Color: Default
Shadow: Default
Shadow: 0.00 X Offset, 0.00 Y Offset
Drawing: ☐ Highlight Reverses Direction; ☐ Shows Touch On Highlight; ✓ Highlighted Adjusts Image; ✓ Disabled Adjusts Image
Font: Helvetica Bold, 15.0
Line Breaks: Truncate Middle

Figure 9-9: Change the UIButton class object to an Info Light button.

4. **Select the Rounded Rect Button item in the `DeepThoughtsView Controller.xib` window and then click the Attributes tab of the Inspector window (or choose Tools➪Attributes Inspector) if it isn't already selected.**

   The Attributes Inspector shows Button Attributes. (Refer to Figure 9-9.)

5. **In the Attributes Inspector, choose Info Light from the Type drop-down menu. (Refer to Figure 9-9.)**

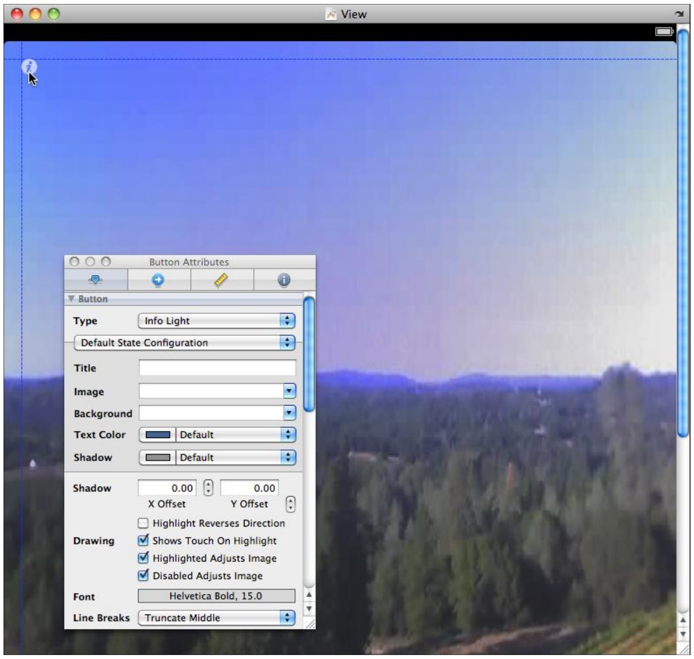
   The Rounded Rect Button item transforms itself into the Light Info Button item in the `DeepThoughtsViewController.xib` window.

6. **Drag the Light Info Button to its position in the View window, as shown in Figure 9-10.**

   Notice the lines displayed by Interface Builder. They're there to make it easy to place the interface element. Interface Builder also displays lines at the borders to help you stay within Apple's User Interface Guidelines. (The lines are actually blue, but that's kind of hard to see in a black-and-white illustration.)

7. **Choose File➪Save to save your changes to the `DeepThoughtsView Controller.xib` file.**

   You can also close Interface Builder and save changes to the file by choosing Interface Builder➪Quit Interface Builder and then clicking Save while closing it.



▼ Button

Type	Info Light
	Default State Configuration
Title	
Image	
Background	
Text Color	Default
Shadow	Default
Shadow	0.00 X Offset, 0.00 Y Offset
Drawing	☐ Highlight Reverses Direction
	✓ Shows Touch On Highlight
	✓ Highlighted Adjusts Image
	✓ Disabled Adjusts Image
Font	Helvetica Bold, 15.0
Line Breaks	Truncate Middle

Figure 9-10: Drag the button into position in the View window.

I placed the Info button in the upper-left corner of the app's view so that when the iPad is rotated (either left or right), and the display orientation switches from portrait to landscape (or vice-versa), the Info button would still be in the upper-left corner.

TIP

It's extremely important with iPad apps to design your app's view to take into consideration both portrait and landscape orientations — that's one of the features of the iPad that truly improves the user experience with the content.

You can see what the Light Info button looks like in its Default State Configuration. (Refer to Figure 9-10.) Click the pop-up menu under the Type pop-up menu in the Attributes Inspector window to choose other configurations, such as Highlighted State Configuration or Selected State Configuration, to see what the button looks like when highlighted or selected. You can also change the text color, shadow, background, and other attributes in the Attributes Inspector.

While you're in a graphical mindset — especially if you're in the middle of processing graphic images for your app or designing interface elements — take the time to create your app icon and add it to your app, as spelled out in the next section.

Adding an Application Icon

You shouldn't procrastinate about adding an application icon. A well-designed icon adds a professional touch, and it takes time to get it right. You may start out with a "placeholder" icon until you've had a chance to explore the App Store and look at other icons, but whether or not you have a finished icon, it helps you identify the app in the Simulator (in case you've installed other projects), and psychologically, it boosts your confidence that the app is real.

An application icon is simply a 57-x-57-pixel `.png` graphics file. Add the file in the same way you added the image in the "Adding an Image to the View" section, earlier in this chapter. Follow these steps:

1. **In Xcode, drag the graphics file into the Resources folder in the Project window's Groups & Files list. (Refer to Figure 9-3.)**

   An alternative is to click the Resources folder in Xcode, choose Project⇨Add to Project, and then navigate to the file you want to add.

   After dragging or adding the file, Xcode displays a dialog for making a copy of the file. (Refer to Figure 9-4.)

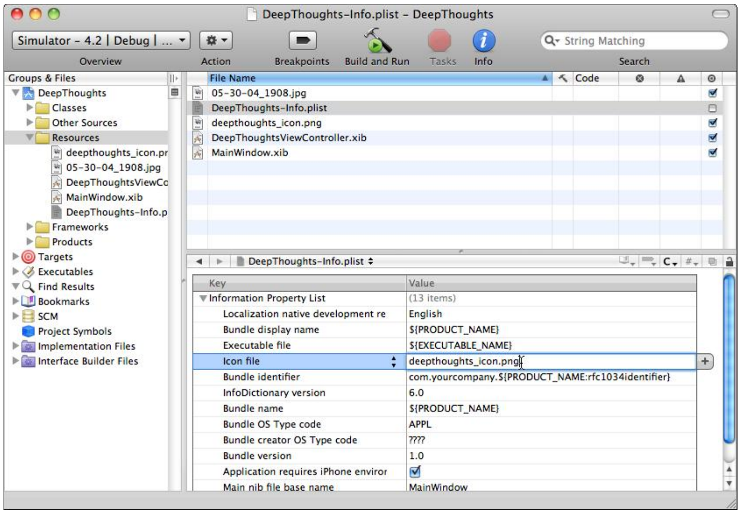
2. **Select the Copy Items into Destination Group's Folder check box to copy the file, and then click Add to finish copying the graphics file into your project.**

   You don't need to change the default settings for Reference Type, Text Encoding, or Recursively Create Groups for Any Added Folders.

After you add the icon's graphics file, you also need to specify that this file is what you want used as the application's icon. You do that by using one of those other mysterious files you see in the Resources folder: `DeepThoughts-Info.plist`. The "plist" part is your clue: The file is a *property list.* Property lists are used extensively by applications as a uniform and convenient means of organizing, storing, and accessing data such as the filename for the app's icon. Xcode lets you edit property lists directly so that you can create and change them as you need to. Here's how:

1. **In the Resources folder, click the `DeepThoughts-Info.plist` file, as shown in Figure 9-11.**

   The contents of the `info.plist` file are displayed in the Editor pane. You're treated to some information about the application, including an item in the Key column labeled `Icon file`.



Key	Value
Information Property List	(13 items)
Localization native development re	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	deepthoughts_icon.png
Bundle identifier	com.yourcompany.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
Application requires iPhone enviror	✓
Main nib file base name	MainWindow

Figure 9-11: Enter the name of the icon file in the info. plist.

2. **Double-click in the empty space in the Value column next to the Icon file.**
3. **Type the name of your `.png` graphics file and then build the project as you normally would.**

   You know — click the Build and Run button in the Project Window toolbar, choose Build⇨Build and Go (Run) from the main menu, or press ⌘+Return. Building the project gives you the opportunity to save it. (You could also quit Xcode, which also gives you the opportunity to save the project.)

After building and running the project, you see your new app icon for the app rather than a blank icon in the Simulator. Now your project looks serious! Which means it's now time to add some serious code that does something interesting with the view, as I show in the next chapter.

Chapter 10

Animating the View

In This Chapter

- Using your friendly Xcode Text editor
- Accessing documentation for quick help with coding
- Searching through your code and establishing markers
- Adding a Constants.h file
- Adding code to the view controller
- Creating animation in the view

I wanted to keep the DeepThoughts sample app as simple as possible so that you can focus on how to build any iPad app with Xcode. As Albert Einstein said, "Everything should be as simple as it is, but not simpler." As you can see by the detail of the previous chapters, building *any* type of iPad app is not *simple* by any means.

But most iPad apps start off with a view, and the View-based Application template creates a skeleton for a fully functioning iPad app, as you find out in this chapter. You also get to flesh out the template with some code that transforms it from an app that just sits there and looks pretty to an app that actually *does* something.

REMEMBER

DeepThoughts is supposed to display falling words — text flowing down the view in different sizes, starting with the words "Peace Love Groovy Music" — at a speed the user can change. DeepThoughts should also allow the user to enter text to substitute different words for "Peace Love Groovy Music" as well as set the speed in advance.

As you add the code to DeepThoughts, I also explain some of the features of the Xcode Text editor.

Using the Xcode Text Editor

The main tool you use to write code for an iPad application is the Xcode Text editor. Apple has gone out of its way to make the Text editor as user-friendly as possible, as evidenced by the following list of (quite convenient) features:

- **Code Sense:** Code Sense is a feature of the editor that shows arguments, placeholders, and suggested code as you type statements. Code Sense can be really useful, especially if you're like me and forget exactly what the arguments are for a function. When Code Sense is active (it is by default), Xcode uses the text you typed, as well as the context within which you typed it, to provide suggestions for completing what it thinks you're *going to* type. You can accept suggestions by pressing Tab or Return. You may also display a list of completions by pressing the Escape key. You can set options for code sensing by choosing Xcode⇨Preferences and clicking the Code Sense tab.
- **Code Folding in the Focus ribbon:** With Code Folding, you can collapse code that you're not working on and display only the code that requires your attention. You do this by clicking in the Focus Ribbon column to the left of the code you want to hide to show a disclosure triangle. Clicking the disclosure triangle hides or shows blocks of code. (Not sure where the Focus Ribbon column is? Look right there between the gutter, which displays line numbers and breakpoints, and the editor.)
- **Switching between header and implementation windows:** On the toolbar above the Text editor, click the last icon before the lock to switch from the `.h` (header) file to the `.m` (implementation) file, and vice versa. While the header declares the class's instance variables and methods, the implementation holds the logic of your code. If you look inside the `Classes` section of the Groups & Files list of the Project window, you can see the separate `.h` and `.m` files for the `DeepThoughtsAppDelegate` and `DeepThoughtsViewController` view classes.
- **Opening a file in a separate window:** Double-click the filename to open the file in a new window. If you have a big monitor, or multiple monitors, this new window enables you to look at more than one file at a time. You can, for example, look at the method of one class *and* the method it invokes in the same class or even a different class.

Accessing Documentation

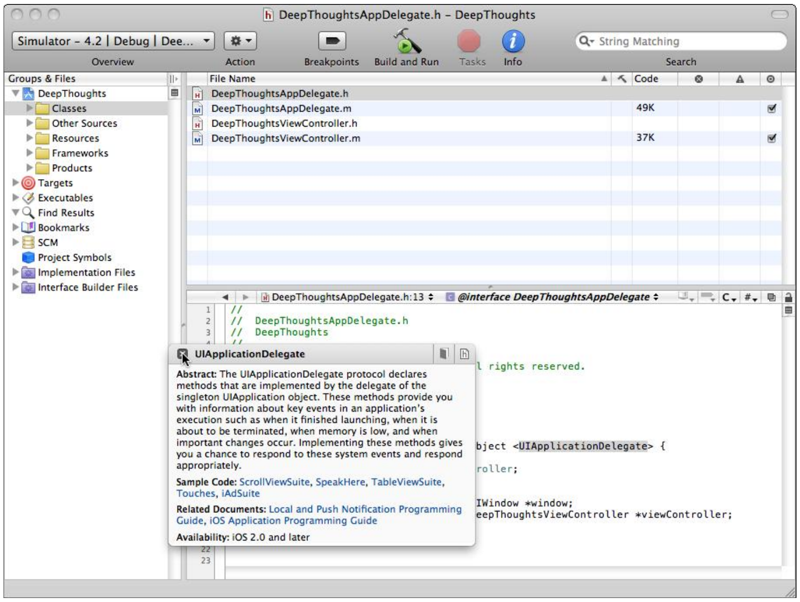
Like many developers, you may find yourself wanting to dig deeper when it comes to a particular bit of code. That's when you'll really appreciate Xcode's Quick Help, header file access, Documentation window, Help menu, and Find tools. With these tools, you can quickly access the documentation for a particular class, method, or property.

To see how this works, say you have the Project window open with the code displayed in Figure 10-1. What if you want to find out more about `UIApplicationDelegate`? What, practically speaking, could you do?

Quick Help

Quick Help is an unobtrusive window that provides the documentation for a single *symbol* — a programming language keyword. It pops up inline, although you can use Quick Help as a symbol inspector (which stays open) by moving the window after it opens. You can also customize the display in Documentation preferences in Xcode preferences.

To get Quick Help for a symbol, double-click to select the symbol in the Text editor (in this case, `UIApplicationDelegate`; see Figure 10-1) and then choose Help➪Quick Help. Alternatively, ⌘-click (right-click) and choose Quick Help from the contextual menu that appears.



```
//
//  DeepThoughtsAppDelegate.h
//  DeepThoughts
//
l rights reserved.
bject <UIApplicationDelegate> {
roller;
IWindow *window;
eepThoughtsViewController *viewController;
```

UIApplicationDelegate

Abstract: The UIApplicationDelegate protocol declares methods that are implemented by the delegate of the singleton UIApplication object. These methods provide you with information about key events in an application's execution such as when it finished launching, when it is about to be terminated, when memory is low, and when important changes occur. Implementing these methods gives you a chance to respond to these system events and respond appropriately.

Sample Code: ScrollViewSuite, SpeakHere, TableViewSuite, Touches, iAdSuite

Related Documents: Local and Push Notification Programming Guide, iOS Application Programming Guide

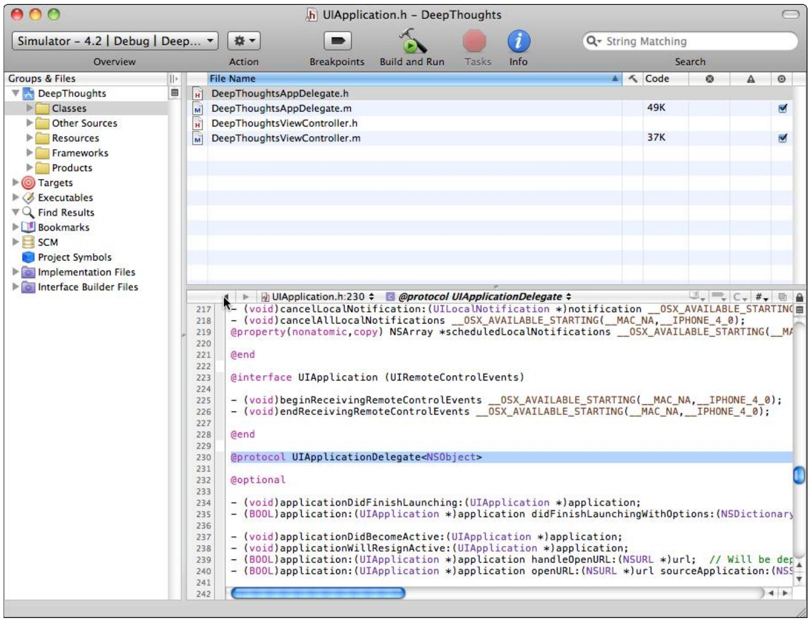
Availability: iOS 2.0 and later

Figure 10-1: Get Quick Help.

The header file for a symbol

Headers are a big deal in code because they're the place where you find the class declaration, which includes all of its instance variables and method declarations. To display the header file for a symbol, press ⌘ while double-clicking the symbol in the Text editor. For example, to get to the header file in Figure 10-2, I pressed ⌘ and then double-clicked `UIApplicationDelegate` (in Figure 10-1). This trick works for the classes you create as well.

To return to the previous view in the Text editor, click the Go Back button (refer to Figure 10-2).



File Name	Code
DeepThoughtsAppDelegate.h	
DeepThoughtsAppDelegate.m	49K
DeepThoughtsViewController.h	
DeepThoughtsViewController.m	37K

```
- (void)cancelLocalNotification:(UILocalNotification *)notification __OSX_AVAILABLE_STARTING
- (void)cancelAllLocalNotifications __OSX_AVAILABLE_STARTING(__MAC_NA,__IPHONE_4_0);
@property(nonatomic,copy) NSArray *scheduledLocalNotifications __OSX_AVAILABLE_STARTING(__MA

@end

@interface UIApplication (UIRemoteControlEvents)

- (void)beginReceivingRemoteControlEvents __OSX_AVAILABLE_STARTING(__MAC_NA,__IPHONE_4_0);
- (void)endReceivingRemoteControlEvents __OSX_AVAILABLE_STARTING(__MAC_NA,__IPHONE_4_0);

@end

@protocol UIApplicationDelegate<NSObject>

@optional

- (void)applicationDidFinishLaunching:(UIApplication *)application;
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary

- (void)applicationDidBecomeActive:(UIApplication *)application;
- (void)applicationWillResignActive:(UIApplication *)application;
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url;  // Will be dep
- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url sourceApplication:(NSS
```

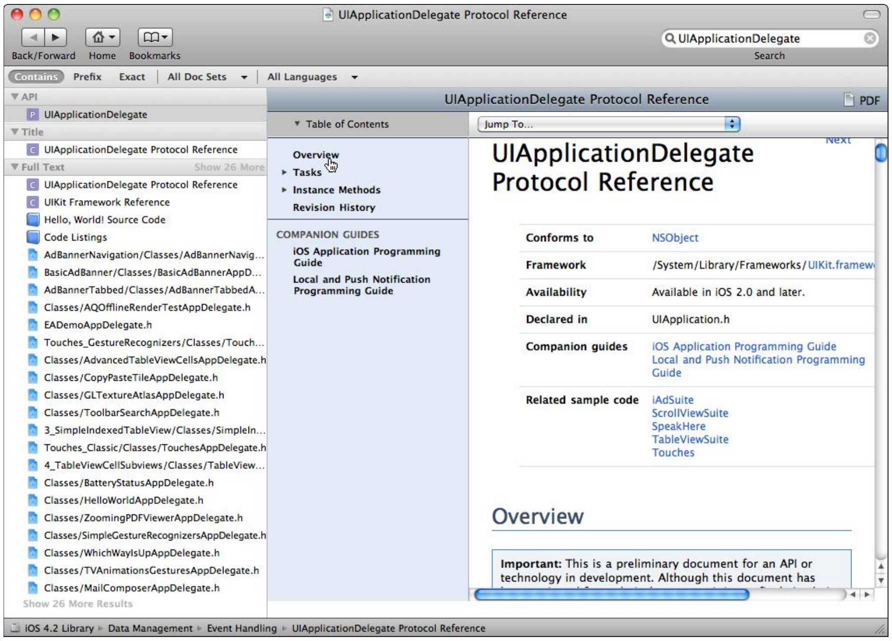
Figure 10-2: The header file for UIApplicationDelegate.

Documentation window

The Documentation window lets you browse and search items that are part of the Apple Developer Connection Reference Library (a collection of developer documentation and technical notes) as well as any third-party documentation you have installed.

You access the documentation by pressing ⌘+Option while double-clicking a symbol. Among other pieces of valuable information, you get access to the Application Programming Interface (API) reference that provides information about the symbol. This access enables you to get the documentation about a method to find out more about it or the methods and properties in a framework class. In Figure 10-3, I pressed ⌘+Option while double-clicking `UIApplicationDelegate`.

Using the Documentation window, you can browse and search the developer documentation — the API references, guides, and article collections about particular tools or technologies — installed on your computer. It's the go-to place for getting documentation about a method or more info about the methods and properties in a framework class.



API
UIApplicationDelegate
Title
UIApplicationDelegate Protocol Reference
Full Text — Show 26 More
UIApplicationDelegate Protocol Reference
UIKit Framework Reference
Hello, World! Source Code
Code Listings
AdBannerNavigation/Classes/AdBannerNavig...
BasicAdBanner/Classes/BasicAdBannerAppD...
AdBannerTabbed/Classes/AdBannerTabbedA...
Classes/AQOfflineRenderTestAppDelegate.h
EADemoAppDelegate.h
Touches_GestureRecognizers/Classes/Touch...
Classes/AdvancedTableViewCellsAppDelegate.h
Classes/CopyPasteTileAppDelegate.h
Classes/GLTextureAtlasAppDelegate.h
Classes/ToolbarSearchAppDelegate.h
3_SimpleIndexedTableView/Classes/SimpleIn...
Touches_Classic/Classes/TouchesAppDelegate.h
4_TableViewCellSubviews/Classes/TableView...
Classes/BatteryStatusAppDelegate.h
Classes/HelloWorldAppDelegate.h
Classes/ZoomingPDFViewerAppDelegate.h
Classes/SimpleGestureRecognizersAppDelegate.h
Classes/WhichWayIsUpAppDelegate.h
Classes/TVAnimationsGesturesAppDelegate.h
Classes/MailComposerAppDelegate.h
Show 26 More Results

Overview
Tasks
Instance Methods
Revision History

COMPANION GUIDES
iOS Application Programming Guide
Local and Push Notification Programming Guide

UIApplicationDelegate Protocol Reference

Conforms to	NSObject
Framework	/System/Library/Frameworks/UIKit.framew
Availability	Available in iOS 2.0 and later.
Declared in	UIApplication.h
Companion guides	iOS Application Programming Guide Local and Push Notification Programming Guide
Related sample code	iAdSuite ScrollViewSuite SpeakHere TableViewSuite Touches

Overview

Important: This is a preliminary document for an API or technology in development. Although this document has

iOS 4.2 Library ▸ Data Management ▸ Event Handling ▸ UIApplicationDelegate Protocol Reference

Figure 10-3: The Documentation window.

Help menu

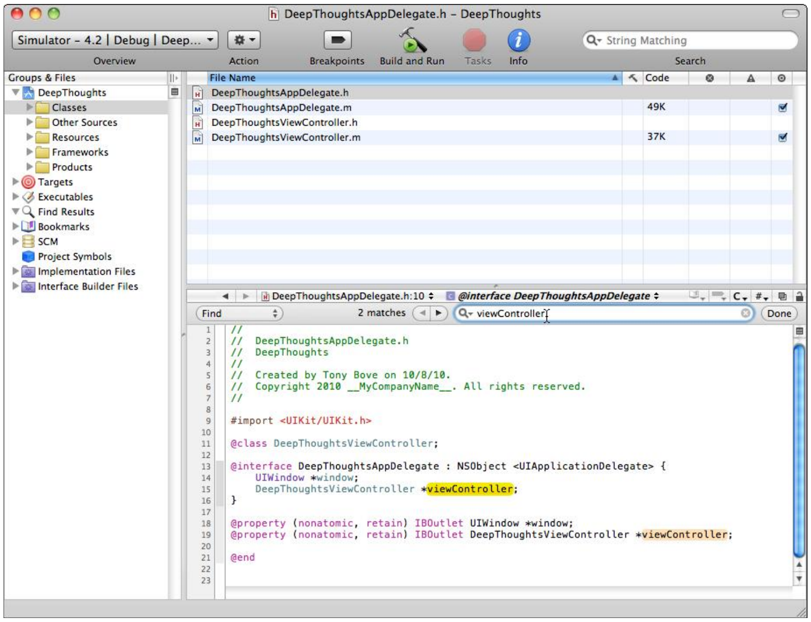
The Help menu's search field also lets you search Xcode documentation as well as open the Documentation window and Quick Help.

You can also ⌘-click (right-click) a symbol to display a contextual pop-up menu that gives you similar options to what you see in the Help menu, including Quick Help (and other related functions).

Find

Xcode can also help you find things in your own project. You'll find that, as your classes get bigger, sometimes you'll want to find a single symbol or all occurrences of a symbol in a file or class. You can easily do that by choosing Edit➪Find➪Find or pressing ⌘+F, which opens a Find toolbar to help you search the file in the editor window.

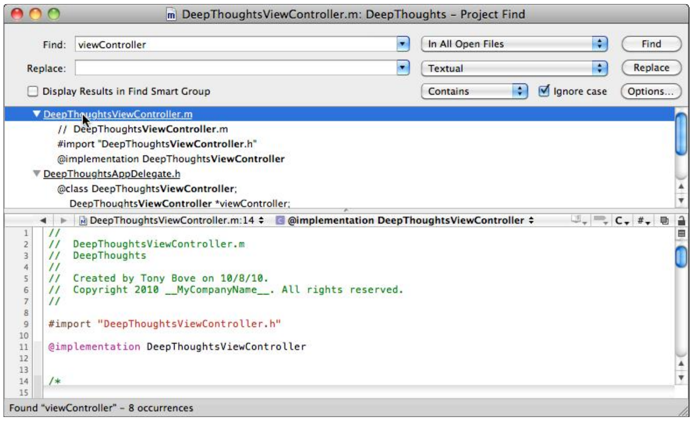
For example, in Figure 10-4, I first press ⌘+F and then type **viewController** in the Find toolbar between the Text editor and Detail view. Xcode finds all the instances of `viewController` in that file and highlights them in the Text editor.



```
//
//  DeepThoughtsAppDelegate.h
//  DeepThoughts
//
//  Created by Tony Bove on 10/8/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#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
```

Figure 10-4: Find "viewController" in the file open in the Text editor.

You can also use Find to go through your whole project by choosing Edit➪Find➪Find in Project, or by pressing ⌘+Shift+F, which opens the Project Find window shown in Figure 10-5. You can type something like **viewController** in the Find field, and then choose In Project — or, as I chose in Figure 10-5, In All Open Files — in the drop-down menu on the right side of the field. (Project Find is a great feature for tracking down something in your code — you're sure to use it often.)



DeepThoughtsViewController.m: DeepThoughts – Project Find

Find: viewController | In All Open Files | Find
Replace: | Textual | Replace
Display Results in Find Smart Group | Contains | [✓ Ignore case] | Options…

```
▼ DeepThoughtsViewController.m
    // DeepThoughtsViewController.m
    #import "DeepThoughtsViewController.h"
    @implementation DeepThoughtsViewController
▼ DeepThoughtsAppDelegate.h
    @class DeepThoughtsViewController;
      DeepThoughtsViewController *viewController;
```

DeepThoughtsViewController.m:14 | @implementation DeepThoughtsViewController

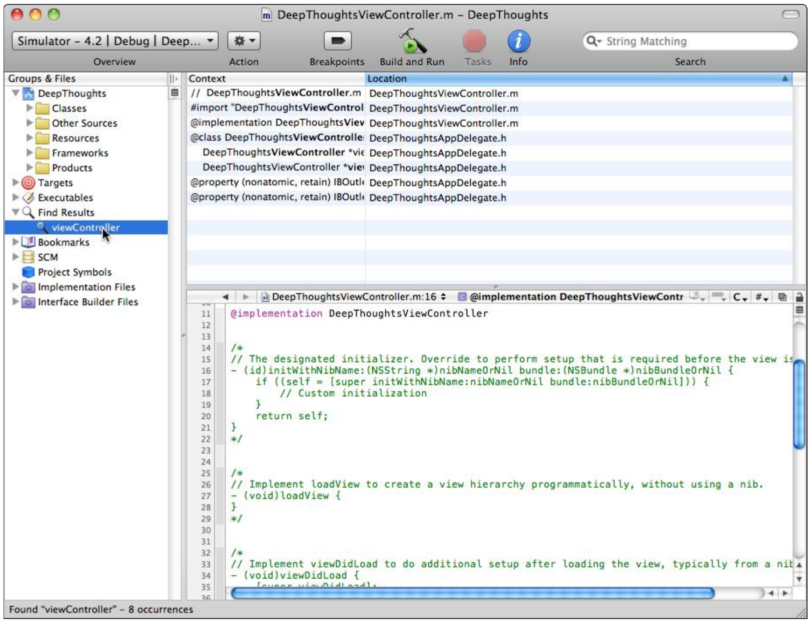
```
//
//  DeepThoughtsViewController.m
//  DeepThoughts
//
//  Created by Tony Bove on 10/8/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import "DeepThoughtsViewController.h"

@implementation DeepThoughtsViewController


/*
```

Found "viewController" – 8 occurrences

Figure 10-5: The Project Find window.

If you select a line in the top pane of the Project Find window, as you can see in Figure 10-5, the file in which that instance occurs is opened in the bottom pane and the reference highlighted.

Your searches are saved in your project. Click the triangle next to Find Results in the Groups & Files list to reveal your searches in the Detail view, as shown in Figure 10-6 (my search for "viewController"). Select a search to see the search results.



DeepThoughtsViewController.m – DeepThoughts

Context	Location
// DeepThoughtsViewController.m	DeepThoughtsViewController.m
#import "DeepThoughtsViewControl	DeepThoughtsViewController.m
@implementation DeepThoughtsViev	DeepThoughtsViewController.m
@class DeepThoughtsViewControlle	DeepThoughtsAppDelegate.h
DeepThoughtsViewController *vie	DeepThoughtsAppDelegate.h
DeepThoughtsViewController *vie	DeepThoughtsAppDelegate.h
@property (nonatomic, retain) IBOutl	DeepThoughtsAppDelegate.h
@property (nonatomic, retain) IBOutl	DeepThoughtsAppDelegate.h

DeepThoughtsViewController.m:16 | @implementation DeepThoughtsViewContr

```
@implementation DeepThoughtsViewController


/*
// The designated initializer. Override to perform setup that is required before the view is
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil {
    if ((self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil])) {
        // Custom initialization
    }
    return self;
}
*/


/*
// Implement loadView to create a view hierarchy programmatically, without using a nib.
- (void)loadView {
}
*/


/*
// Implement viewDidLoad to do additional setup after loading the view, typically from a nib
- (void)viewDidLoad {
```

Found "viewController" – 8 occurrences

Figure 10-6: Revisit your searches, which are saved in your project.

Now that you have some idea of how to use the Xcode Text editor, it's time to write some code.

Figuring Out Where Your Code Goes

One of the biggest challenges facing a developer working with an unfamiliar framework and template is figuring out where in the *control flow* — the sequence in which messages are sent during execution — to put the code to get something done.

The delegate object

Okay, here's how the template set up the DeepThoughts app:

1. `UIApplication` loads the parts of the `MainWindow.xib` to create `DeepThoughtsAppDelegate` and the window.
2. `UIApplication` sends `DeepThoughtsAppDelegate` the `application:didFinishLaunchingWithOptions:` message.
3. The `application:didFinishLaunchingWithOptions:` method in `DeepThoughtsAppDelegate` initializes the view (`viewController`).

Note that the `application:didFinishLaunchingWithOptions:` message is sent at the very beginning, before the user can even see anything on the screen. Here's where you'd insert your code to initialize your application — where you'd load data, for example, or restore the state of the application to where it was the last time the user exited. (For DeepThoughts, you don't need to change anything here — it's all supplied by the template.)

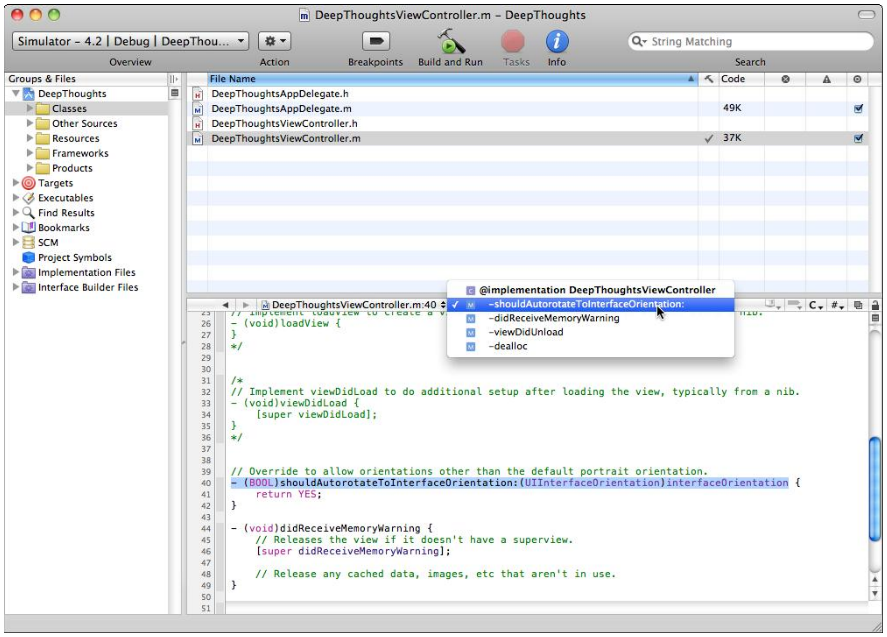
The view controller object

`DeepThoughtsViewController` is the controller responsible for *managing* the view. Select `DeepThoughtsViewController.m` to see its code in the Text editor. Then click the Methods list pop-up menu in the Xcode Text Editor's Navigation bar (to the left of the Bookmarks menu), as shown in Figure 10-7, to see the controller object's methods.

What you see in the pop-up menu is a list of active methods in the controller object.

You can also see that the code in the Text editor starts off with sections of comments that include code you can implement if you want to, simply by removing the comments. (As Objective-C programmers already know, the lines beginning with `//` are single-line comments that don't do anything. A line beginning with `/*` followed by a line ending with `*/` marks an entire comment section that doesn't do anything.)

The first of these (commented out) sections starts off with `The designated initializer`, the second with `Implement loadView`, and the third with `Implement viewDidLoad`.



DeepThoughtsViewController.m – DeepThoughts
Simulator – 4.2 | Debug | DeepThou...
Overview
Action
Breakpoints
Build and Run
Tasks
Info
String Matching
Search
Groups & Files
DeepThoughts
Classes
Other Sources
Resources
Frameworks
Products
Targets
Executables
Find Results
Bookmarks
SCM
Project Symbols
Implementation Files
Interface Builder Files
File Name
Code
DeepThoughtsAppDelegate.h
DeepThoughtsAppDelegate.m 49K
DeepThoughtsViewController.h
DeepThoughtsViewController.m 37K
DeepThoughtsViewController.m:40
@implementation DeepThoughtsViewController
✓ -shouldAutorotateToInterfaceOrientation:
-didReceiveMemoryWarning
-viewDidUnload
-dealloc

```
- (void)loadView {
}
*/


/*
// Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
}
*/


// Override to allow orientations other than the default portrait orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation {
    return YES;
}

- (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.
}
```

Figure 10-7: Click the Methods list pop-up to see all the methods in the controller object.

After those comment sections, you encounter the first active method:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
            (UIInterfaceOrientation)interfaceOrientation {
   return YES;
}
```

This method (which appears as `shouldAutorotateToInterface Orientation:` in the Methods list pop-up, as shown in Figure 10-7) starts your app with the default portrait orientation, unless you override it. The comment above this code says "Override to allow orientations other than the default portrait orientation."

TIP

For DeepThoughts, you don't have to do anything with this code because you *want* the app to start off in portrait orientation, but for other apps you may create, you can override the code by simply adding comment markers to *comment it out* (make the code inactive). To add comment markers, insert `/*` before the beginning of the statement (right before `- (BOOL)`), and insert `*/` on a line after the last bracket (`}`).

Marking code sections in the view controller

Before adding your code to `DeepThoughtsViewController.m`, it helps to know what each section of the template-provided code does and to mark off each section so that you can navigate quickly to the sections you're interested in. You can do so by marking each of the code territories.

The `# pragma mark` statement marks each territory, and the marks themselves appear in the Methods list pop-up menu. You use the `# pragma mark` statement with a label (such as `View life cycle` or `Orientation`) to add the label in bold to the Methods list so that you can logically identify and keep separate the methods in the list.

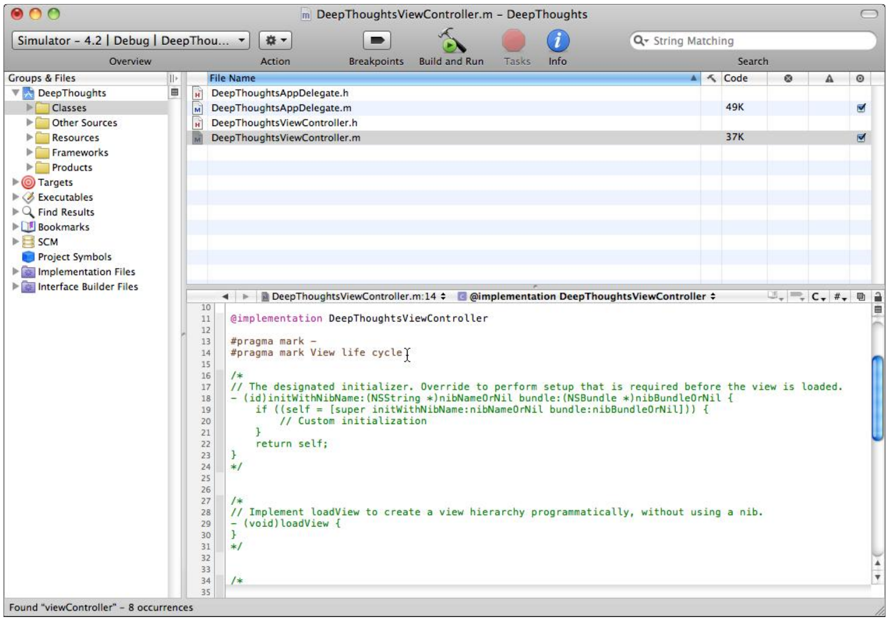
For example, in Figure 10-8, I add two `# pragma mark` statements above the commented sections containing the initialization and load-view code:

```
#pragma mark -
#pragma mark View life cycle
```

The first `# pragma mark` (with a space and a dash) places a one-pixel horizontal line with space around it in the Methods list, to separate list items — you don't have to include these, but they make the list easier to navigate. The second `# pragma mark` places `View life cycle` in the Methods lists.

In Figure 10-9, I add two `# pragma mark` statements above the `shouldAutorotateToInterfaceOrientation:` method to mark the section as `Orientation`, and then I put two more `# pragma mark` statements above the `didReceiveMemoryWarning` method to mark that section as `Memory Management`.

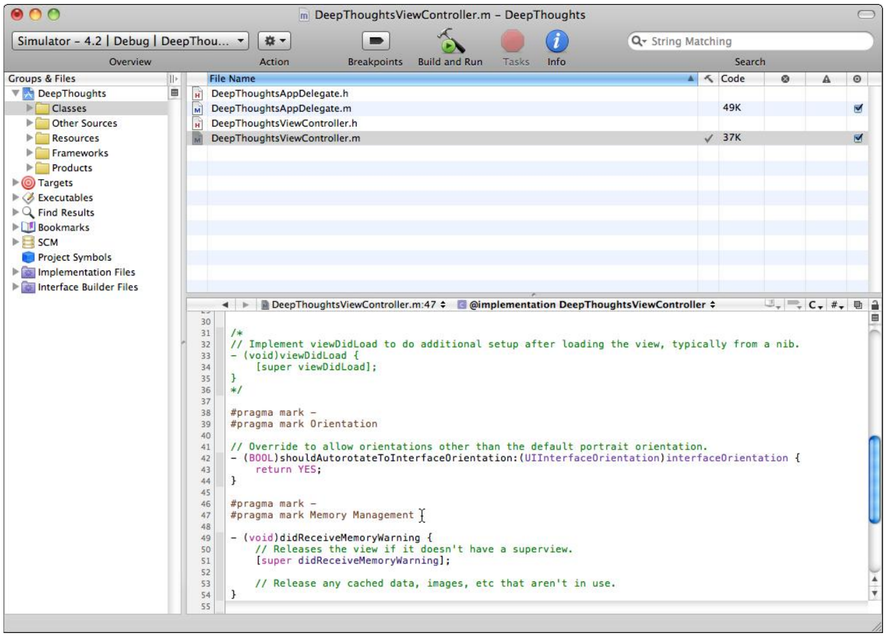
You can now click the Methods list pop-up menu, as shown in Figure 10-10, to see these markers and to navigate to each section quickly. This trick is useful for finding code sections, organizing your code, and adding new code in the proper sections.



```
@implementation DeepThoughtsViewController

#pragma mark -
#pragma mark View life cycle

/*
// The designated initializer. Override to perform setup that is required before the view is loaded.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil {
    if ((self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil])) {
        // Custom initialization
    }
    return self;
}
*/


/*
// Implement loadView to create a view hierarchy programmatically, without using a nib.
- (void)loadView {
}
*/


/*
```

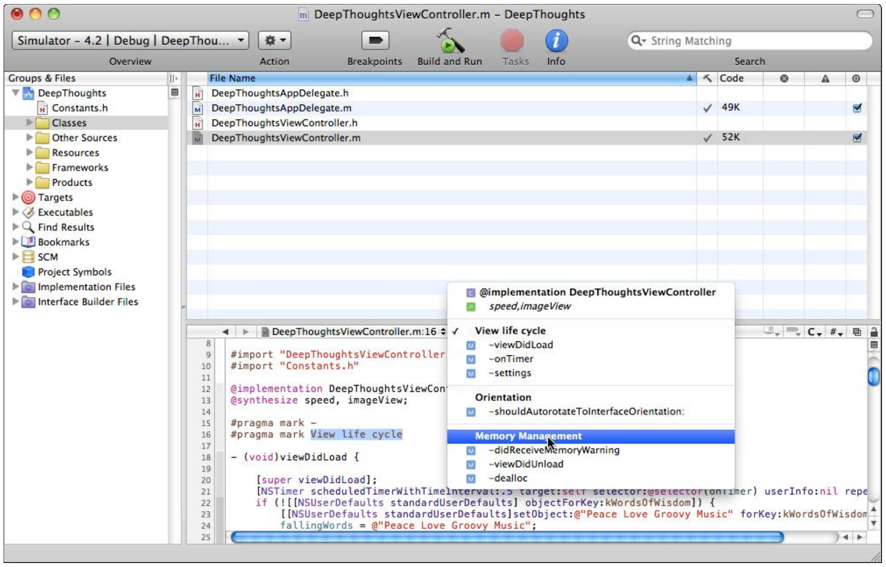
Found "viewController" – 8 occurrences

Figure 10-8: Add markers for the first major section of code.



```

/*
// Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
}
*/

#pragma mark -
#pragma mark Orientation

// Override to allow orientations other than the default portrait orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation {
    return YES;
}

#pragma mark -
#pragma mark Memory Management

- (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.
}

```

Figure 10-9: Add markers for the Orientation and Memory Management sections.



```
#import "DeepThoughtsViewController
#import "Constants.h"

@implementation DeepThoughtsViewCon
@synthesize speed, imageView;

#pragma mark -
#pragma mark View life cycle

- (void)viewDidLoad {

    [super viewDidLoad];
    [NSTimer scheduledTimerWithTimeInterval:.5 target:self selector:@selector(onTimer) userInfo:nil repe
    if (![[NSUserDefaults standardUserDefaults] objectForKey:kWordsOfWisdom]) {
        [[NSUserDefaults standardUserDefaults]setObject:@"Peace Love Groovy Music" forKey:kWordsOfWisdom
        fallingWords = @"Peace Love Groovy Music";
```

Figure 10-10: The markers appear in the Methods list.

Preparing for User Settings

Before diving into the code that animates the view in DeepThoughts, keep in mind one of the important tenets of object-oriented programming. Yes, I'm talking about *encapsulation,* the idea that you should keep the details of how an object works hidden from the other objects that use it.

In the case of DeepThoughts, the actual text that is shown in animation as well as the actual speed of the animation are details that should not be hard-coded into your view controller. In Chapter 11, I show you how to enable the user to change the text and the speed setting, so you want to keep these details generic in the code you add to the view controller.

You also need to add a method to the view controller that connects to the Light Info button added to the user interface in Chapter 9. This button will enable the user to change the text and speed settings. For now, this method will be a placeholder until you fill it out with code in Chapter 11.

Editing the view controller header

With DeepThoughts, the idea is to enable the user to enter his or her own words for the falling words animation, as well as change the speed of the animation. The Light Info button, added in Chapter 9, is supposed to react to a Touch Up Inside event — which occurs when the button is touched and then released, symbolizing a click. When that happens, it invokes a method in the view controller to display a *modal view,* which is a view presented modally like a dialog for the user to enter new words and change the speed setting, as I show in Chapter 11.

For now, you need to declare an action method by using the `IBAction` qualifier found in `DeepThoughtsViewController.h` and add a placeholder method corresponding to it in `DeepThoughtsViewController.m`. You need to use the `IBAction` type qualifier, which is used by Interface Builder to synchronize actions added programmatically with its internal list of action methods defined for a project. You also need to add the code needed to display the falling words at a certain speed.

First open `DeepThoughtsViewController.h` (the header file) and insert the code in bold in Listing 10-1.

Listing 10-1: DeepThoughtsViewController.h

```
@interface DeepThoughtsViewController : UIViewController
          {

  UIImage     *fallingImage;
  NSString    *fallingWords;
  UIImageView *imageView;
  double       speed;
}
- (IBAction)settings;

@property (readwrite)  double speed;
@property (nonatomic, retain) UIImageView *imageView;
@property (nonatomic, retain) NSString    *fallingWords;
@end
```

This code establishes the falling image itself (`fallingImage`), which will contain the text string in `fallingWords` and will flow down the display according to the `speed`.

Following that code is the action method declaration using the `IBAction` qualifier. In Chapter 11, you use Interface Builder to specify the fact that, when the user taps the Light Info button, the target is the `DeepThoughts ViewController` object and the method to invoke is `settings`. This is an example of the Target-Action pattern described in Chapter 7.

The `@property` declarations declare that there are accessor methods for the compiler to create, and the corresponding `@synthesize` statements you add in the next section tell the compiler to actually create them for you. I explain accessor methods in Chapter 11.

You're done with the `DeepThoughtsViewController.h` file for this chapter, but before you edit the code in `DeepThoughtsViewController.m`, you need to add the keys for your settings in a `Constants.h` file.

Adding a Constants.h file

To save and read settings in your app, you can use a built-in, easy-to-use class that lets you read and set user preference settings from your app — `NSUserDefaults`. The class is also used by the Settings app that comes with your iPad (which Apple has graciously consented to let us peons use).

You use `NSUserDefaults` to read and store preference data to a default database, using a key value — just as you would access keyed data from an `NSDictionary`. I explain how this works in Chapter 11, but for now, the particular keys you will use are `kWordsOfWisdom` for the falling words replacement text, `kSpeed` for the animation speed, and `kMaxSpeed` for the maximum speed possible.

To use keys like `kWordsOfWisdom`, `kSpeed`, or `kMaxSpeed`, you need to first define them in a `Constants.h` file. To implement the `Constants.h` file in your project, do the following:

1. **Select the project name (DeepThoughts) in the Groups & Files list and then choose File➪New File from the Xcode main menu.**
2. **In the New File dialog that appears, choose Other from the listing on the left (under the Mac OS X heading) and then choose Empty File in the main pane, as shown in Figure 10-11.**

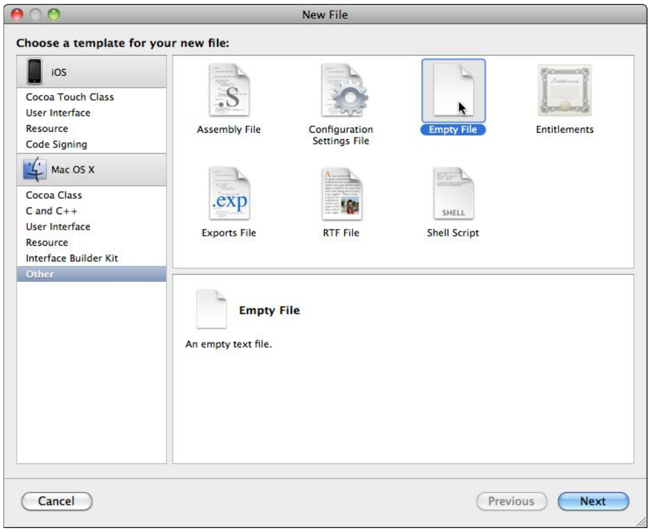


Figure 10-11: Create an empty file.

3. **In the new dialog that appears, name the file `Constants.h` (as shown in Figure 10-12) and then click Finish.**

   Don't make any other changes — just the name. The Add to Project pop-up menu should already be set to the name of the project — DeepThoughts (if not, choose the name of the project from the menu).

   The new empty `Constants.h` file is saved in your project at the bottom of the DeepThoughts group (under Products), but you can drag it up to the top of the group, above Classes, as shown in Figure 10-13. I typically do this so that the `Constants.h` file is easy to find and change.

With a new home for your constants all set up and waiting, all you have to do is add the constants you need, as shown in Figure 10-14:

```
#define kWordsOfWisdom   @"wordsOfWisdomPreference"
#define  kSpeed                  @"speedPreference"
#define  kMaxSpeed                             20.0
```

Having a `Constants.h` file in hand is great, but you have to let `DeepThoughtsViewController.m` know that you plan to use it, as I show in the next section.

It may seem like you are starting at the end and working backwards, but it makes sense to show the code in DeepThoughts that uses these settings first and then show in Chapter 11 how you can enable the user to change and save these settings.

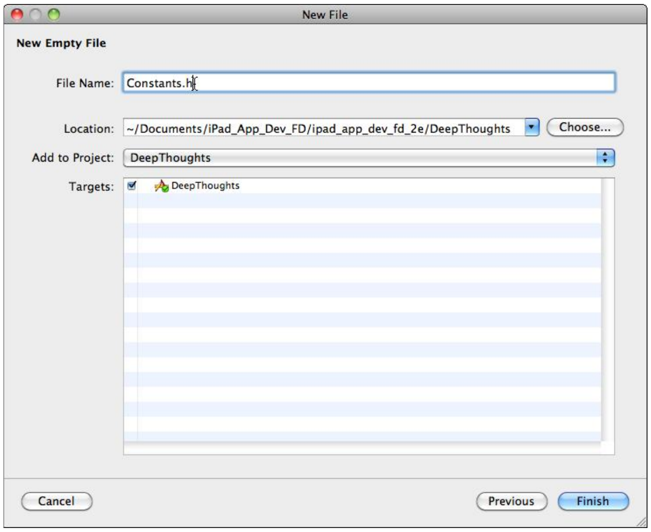

New File
New Empty File
File Name: Constants.h
Location: ~/Documents/iPad_App_Dev_FD/ipad_app_dev_fd_2e/DeepThoughts
Add to Project: DeepThoughts
Targets: ✓ DeepThoughts

Figure 10-12: Name the new file.

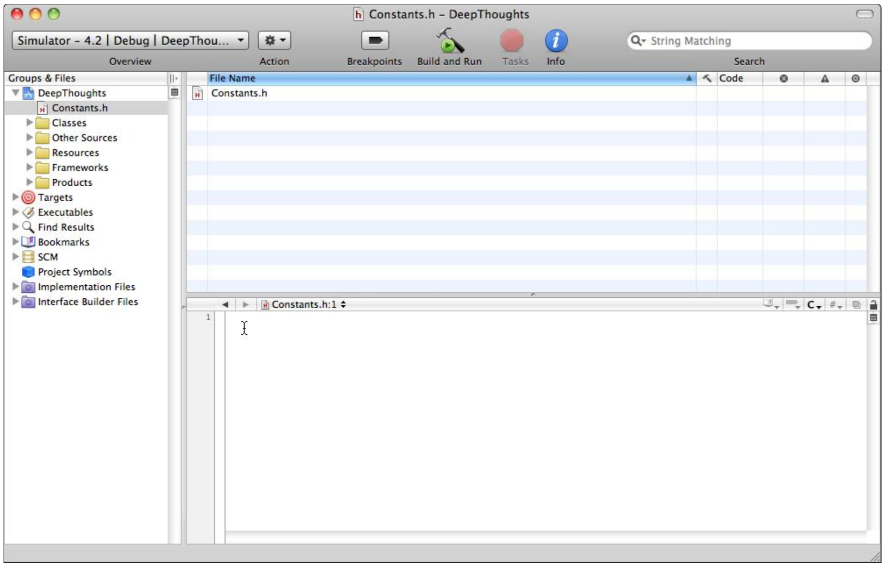
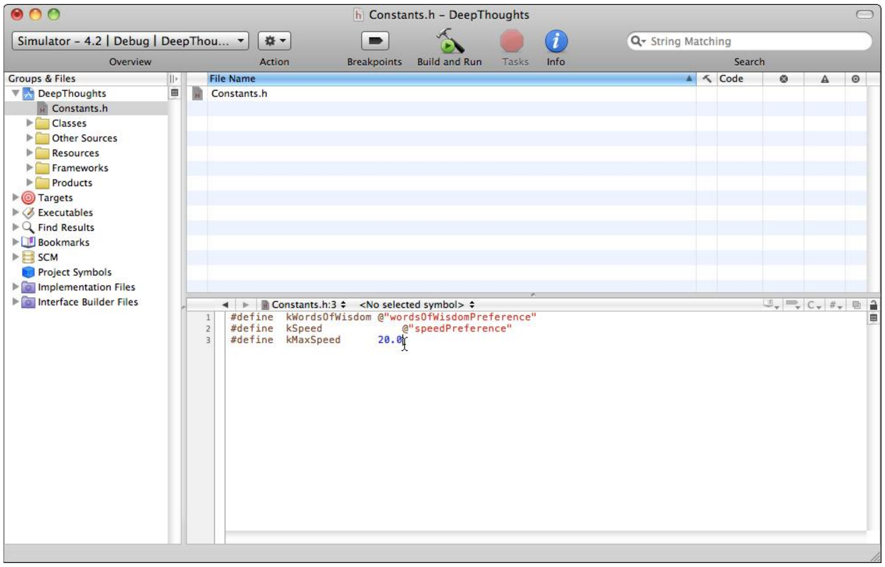


Figure 10-13: The empty Constants.h file.



```
#define  kWordsOfWisdom @"wordsOfWisdomPreference"
#define  kSpeed              @"speedPreference"
#define  kMaxSpeed         20.0
```

Figure 10-14: Define the keys in the Constants.h file.

To put the settings to use in the app's view, you have to link it up with the view's controller — in this case, `DeepThoughtsViewController`. The best place to do that is `viewDidLoad`, which is invoked right after the view has been loaded from the nib file. `viewDidLoad` is found in the `DeepThoughtsViewController.m` file, so that's where you'd go to insert your code to use the settings to control the animated view. The next section shows you how that's done.

Controlling the View

Select the `DeepThoughtsViewController.m` file so that it appears in the Text editor, and insert the code in bold in Listing 10-2. (The code that's not in bold is supplied by the View-based Application template, except the non-bolded `#pragma mark` markers, which you added earlier in this chapter, in "Marking code sections in the view controller.")

Listing 10-2: DeepThoughtsViewController.m

```
#import "DeepThoughtsViewController.h"
#import "Constants.h"

@implementation DeepThoughtsViewController
@synthesize speed, imageView;
@synthesize fallingWords;
#pragma mark -
#pragma mark View life cycle
/*
// The designated initializer. Override to perform setup
          that is required before the view is loaded.
- (id)initWithNibName:(NSString *)nibNameOrNil
          bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil
          bundle:nibBundleOrNil]) {
        // Custom initialization
    }
    return self;
}
*/
/*
// Implement loadView to create a view hierarchy
          programmatically, without using a nib.
- (void)loadView {
}
*/
/*
// Implement viewDidLoad to do additional setup after
          loading the view, typically from a nib.
- (void)viewDidLoad {
    [super viewDidLoad];
}
*/
- (void)viewDidLoad {

  [super viewDidLoad];
  [NSTimer scheduledTimerWithTimeInterval:.5 target:self
          selector:@selector(onTimer) userInfo:nil
          repeats:YES];
```

(continued)

Listing 10-2 *(continued)*

```
  if (![[NSUserDefaults standardUserDefaults]
          objectForKey:kWordsOfWisdom]) {
    [[NSUserDefaults standardUserDefaults]
          setObject:@"Peace Love Groovy Music"
          forKey:kWordsOfWisdom];
    fallingWords = @"Peace Love Groovy Music";
  }
  else {
    fallingWords = [[NSUserDefaults standardUserDefaults]
          stringForKey:kWordsOfWisdom];
  }
  if (![[NSUserDefaults standardUserDefaults]
          objectForKey:kSpeed] ){
    [[NSUserDefaults standardUserDefaults]setDouble:10.0
          forKey:kSpeed];
    speed = kMaxSpeed-10.0;}
  else {
    speed = kMaxSpeed-[[NSUserDefaults
          standardUserDefaults] doubleForKey:kSpeed] ;
  }
}

#pragma mark -
#pragma mark Animation

- (void)onTimer{

  UILabel *fallingImageView = [[UILabel alloc]
          initWithFrame:CGRectMake(0, 0, 100, 30)];
  fallingImageView.text = fallingWords;
  fallingImageView.textColor = [UIColor  purpleColor];
  fallingImageView.font = [UIFont systemFontOfSize:30];
  fallingImageView.backgroundColor = [UIColor
          clearColor];

  fallingImageView.adjustsFontSizeToFitWidth = YES;

  int startX = round(random() % 400);
  int endX =  round(random() % 400);
  //speed of falling
  double randomSpeed = (1/round(random() % 100) +1)
          *speed;
  // image size;
  double scaleH = (1/round(random() % 100) +1) *60;
  double scaleW = (1/round(random() % 100) +1) *200;

  CGRect startImageFrame = fallingImageView.frame;
  CGRect endImageFrame = fallingImageView.frame;
  startImageFrame.origin = CGPointMake(startX, -100);
  endImageFrame = CGRectMake(endX, self.view.frame.size.
           height,scaleW, scaleH);
  fallingImageView.frame = startImageFrame;
  fallingImageView.alpha = .75;
  [self.view addSubview:fallingImageView];

[UIView animateWithDuration:randomSpeed
                   animations:^ {
                     [UIView setAnimationDelegate:self];
                     fallingImageView.frame =
           CGRectMake(endX, self.view.frame.size.height,
           scaleW, scaleH);
                   }
                   completion:^(BOOL finished){
                     [fallingImageView
           removeFromSuperview];
                     [fallingImageView release];
                   }];

}

#pragma mark -
#pragma mark Controls

- (IBAction)settings {

}

#pragma mark -
#pragma mark Orientation

// Override to allow orientations other than the default
           portrait orientation.
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfac
           eOrientation)interfaceOrientation {
    return YES;
}

#pragma mark -
#pragma mark Memory Management

- (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.
}
```

(continued)

Listing 10-2 ***(continued)***

```
- (void)viewDidUnload {
        // Release any retained subviews of the main view.
        // e.g. self.myOutlet = nil;
  self.imageView = nil;
  self.fallingWords = nil;

}


- (void)dealloc {
  self.imageView = nil;
  self.fallingWords = nil;
  [super dealloc];
}

@end
```

That's a lot to swallow at once, but I explain how all this works in the rest of this chapter and in Chapter 11.

The first statement you add imports the `Constants.h` file:

```
#import "Constants.h"
```

You can now use the keys you set up in "Adding a Constants.h file" in this chapter with `NSUserDefaults` in the subsequent code to retrieve the user settings.

Although the `@property` declarations way back in Listing 10-1 tell the compiler that there are accessor methods (which I describe in more detail in Chapter 11), these methods still have to be created. Fortunately, Objective-C will create these accessor methods for you whenever you include an `@synthesize` statement — the next bolded item in Listing 10-2:

```
@synthesize speed, imageView;
@synthesize fallingWords;
```

The `@synthesize` statements tell the compiler to create accessor methods for you — one for each `@property` declaration (`speed`, `imageView`, and `fallingWords`).

At the end of the bolded code you add in Listing 10-2 is a new `#pragma mark` section titled `Controls` that includes the placeholder `settings` method for connecting the Light Info button to the view controller:

```
#pragma mark -
#pragma mark Controls

- (IBAction)settings {

}
```

This is the action method using the `IBAction` qualifier. In Chapter 11, you use Interface Builder to specify that when the user taps the Light Info button, the target is the `DeepThoughtsViewController` object, and the method to invoke is `settings`. This is an example of the Target-Action pattern described in Chapter 7.

The viewDidLoad method

Now look at the bolded code section you add in Listing 10-2 marked as `View life cycle`, which occurs right after the following commented-out code:

```
/*
 // Implement viewDidLoad to do additional setup after
          loading the view, typically from a nib.
 - (void)viewDidLoad {
 [super viewDidLoad];
 }
 */
```

The `viewDidLoad` message is sent right after the view has been loaded from the nib file, which is the `.xib` file that you can modify in Interface Builder. (Check out Chapter 8 for a complete explanation of that loading process.) This is the place where you insert your code for *view initialization,* which in this case means displaying the DeepThoughts' falling words.

TIP

This would also be the place to insert your code to do anything needed before the view becomes visible. Although I don't use it in this example, you could take advantage of the commented-out `loadView` statement to create a view hierarchy programmatically, *without* using a nib file. However, that info is beyond the scope of this book. You could also include a `viewWillAppear` message, which is sent right before the view will appear. Both `viewDidLoad` and `viewWillAppear` are methods declared in the `UIViewController` class and are invoked at the appropriate times by the framework.

REMEMBER

Although I left the commented-out code in place to show where you would insert your version of the `viewDidLoad` method (right below it), you can delete the commented-out code.

The `viewDidLoad` method you inserted (in bold in Listing 10-2) starts out by setting up a timer for the interval between each display of falling words:

```
- (void)viewDidLoad {

  [super viewDidLoad];
  [NSTimer scheduledTimerWithTimeInterval:.5 target:self
          selector:@selector(onTimer) userInfo:nil
          repeats:YES];
```

You use the `NSTimer` class to create timers. A timer waits until a certain time interval has elapsed and then fires, sending a specified message to a target object. I use the `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` class method to create the timer and schedule it on the current run loop in the default mode. The *interval* here is 0.5 seconds, the *target* is `self`, and the *selector* is the message to send to the target when the timer fires — in this case, `onTimer`. The *userInfo* is the user info for the timer (set to `nil`), and `repeats` is set to `YES` — that is, the timer will repeatedly reschedule itself until invalidated.

Next, the code checks to see whether the `kWordsOfWisdom` setting has been moved into `NSUserDefaults`:

```
if (![[NSUserDefaults standardUserDefaults]
          objectForKey:kWordsOfWisdom]) {
    [[NSUserDefaults standardUserDefaults]
          setObject:@"Peace Love Groovy Music"
          forKey:kWordsOfWisdom];
    fallingWords = @"Peace Love Groovy Music";
  }
  else {
    fallingWords = [[NSUserDefaults standardUserDefaults]
          stringForKey:kWordsOfWisdom];
  }
  if (![[NSUserDefaults standardUserDefaults]
          objectForKey:kSpeed] ){
    [[NSUserDefaults standardUserDefaults]setDouble:10.0
          forKey:kSpeed];
    speed = kMaxSpeed-10.0;}
  else {
    speed = kMaxSpeed-[[NSUserDefaults
          standardUserDefaults] doubleForKey:kSpeed] ;
  }
```

The code moves the user’s preferences into `NSUserDefaults` only *after* the application runs for the first time. However, if you decide to make user preference settings available in the Settings app (as shown in Chapter 19), Settings will update preferences in NSUserDefaults if the user makes any changes.

If the settings have not been moved into NSUserDefaults yet, the code uses the initial preference value ("Peace Love Groovy Music") for fallingWords.

```
[[NSUserDefaults standardUserDefaults]setObject:@"Peace
          Love Groovy Music" forKey:kWordsOfWisdom];
    fallingWords = @"Peace Love Groovy Music";
```

If the settings *have* been moved into NSUserDefaults, the code reads them in and then sets fallingWords to whatever the user's preference is.

```
else {
    fallingWords = [[NSUserDefaults standardUserDefaults]
          stringForKey:kWordsOfWisdom];
```

The code then repeats this check with the speed setting.

You use standardUserDefaults (a NSUserDefaults class method) to gain access to the standard user default settings. You can store data there, as you discover in Chapter 11.

Drawing the view

Connecting the timer to the actual drawing of the display is the onTimer method. Take a good look at the code for this method (from the bold code in Listing 10-2), which starts with a new #pragma marker titled Animation:

```
#pragma mark -
#pragma mark Animation

- (void)onTimer{

 UILabel *fallingImageView = [[UILabel alloc]
          initWithFrame:CGRectMake(0, 0, 100, 30)];
 fallingImageView.text = fallingWords;
 fallingImageView.textColor = [UIColor  purpleColor];
 fallingImageView.font = [UIFont systemFontOfSize:30];
 fallingImageView.backgroundColor = [UIColor  clearColor];

 fallingImageView.adjustsFontSizeToFitWidth = YES;

 int startX = round(random() % 400);
 int endX =  round(random() % 400);
```

```
  //speed of falling
  double randomSpeed = (1/round(random() % 100) +1) *speed;
  // image size;
  double scaleH = (1/round(random() % 100) +1) *60;
  double scaleW = (1/round(random() % 100) +1) *200;

  CGRect startImageFrame = fallingImageView.frame;
  CGRect endImageFrame = fallingImageView.frame;
  startImageFrame.origin = CGPointMake(startX, -100);
  endImageFrame = CGRectMake(endX, self.view.frame.size.
           height,scaleW, scaleH);
  fallingImageView.frame = startImageFrame;
  fallingImageView.alpha = .75;
  [self.view addSubview:fallingImageView];

  [UIView animateWithDuration:randomSpeed
                    animations:^ {
      [UIView setAnimationDelegate:self];
        fallingImageView.frame = CGRectMake(endX, self.
            view.frame.size.height, scaleW, scaleH);
                    }
        completion:^(BOOL finished){
                      [fallingImageView
            removeFromSuperview];
                      [fallingImageView release];
                    }];

}
```

The `UILabel` class implements a read-only text view. You can use this class to draw one or multiple lines of static text. In this case, the block of code uses the `initWithFrame` method with `CGRectMake` to create a rectangle, with the x-coordinate and y-coordinate of the rectangle's origin point at (0, 0) and a specified width and height (100, 30).

The code converts the `fallingWords` string to `fallingImageView` for display; sets up the text color, font, and background color; and adjusts the font size for the width. The `font` and `textColor` properties apply to the entire text string.

The next block of code uses the `random` function for the starting and ending points (`startX` and `endX`), for `speed`, and for width (`scaleW`) and height (`scaleH`) for `fallingImageView`. The `random` function uses a nonlinear additive-feedback random number generator, with a default table of size 31 long integers, and returns successive pseudo-random numbers in the range from 0 to 2,147,483,647. The code uses a `CGRect` structure for the location and dimensions of the rectangle, using CGPointMake to get the rectangle's origin point and using CGRectMake to build the rectangle. Then, addSubview adds a view so that it's displayed above its siblings.

The animation block

The UIView class provides common methods you use to create all types of views. In this particular code, it's used for a block of animation. The block-based animation method animateWithDuration:animations: (available only with iOS version 4.0 and newer) greatly simplifies creating an animation. With one method call, you can specify the animation and its options. (If your application runs on earlier versions of iOS, you must use the beginAnimations:context: and commitAnimations class methods to mark the beginning and ending of your animations.)

Inside the block, the code sets property values to make visual changes that comprise the animation. In this case, the code changes the rectangle's starting coordinates from startX to endX, and from -100 to self.view.frame.size.height:

```
fallingImageView.frame = CGRectMake(endX, self.view.frame.
            size.height, scaleW, scaleH);
```

The animateWithDuration:animations:completion method sets the animation duration and code to use upon completion. The completion code uses removeFromSuperview (an instance method of the UIView class) to remove fallingImageView from its superview, from its window, and from the responder chain; it then uses release (an instance method of the NSAutoreleasePool class) to release fallingImageView. Remember, you own any object you create with alloc, which means you're responsible for releasing it when you're done.

Freeing up memory

That's not all you're responsible for. You have to always assume that low-memory conditions prevail, and that the view controller will need to release its view and any objects associated with the view to free up memory.

The viewDidUnload method is the counterpart to viewDidLoad — you use it to relinquish ownership in the view and its objects. Back in Listing 10-2 (in case you missed it at the end), you add the following to it (in bold):

```
- (void)viewDidUnload {
       // Release any retained subviews of the main view.
       // e.g. self.myOutlet = nil;
  self.imageView = nil;
  self.fallingWords = nil;
}
```

You're using the preferred method of relinquishing ownership: using the corresponding accessor method to set the value of the object to `nil`.

Because the view controller also stores references to views and other objects, it's also responsible for relinquishing ownership of those objects safely in its `dealloc` method, which is why you add them in Listing 10-2:

```
- (void)dealloc {
  self.imageView = nil;
  self.fallingWords = nil;
  [super dealloc];
}
```

An object's `dealloc` method is invoked indirectly through the `release` instance method of `NSObject`. Subclasses must implement their own versions of `dealloc` to allow the release of any additional memory consumed by the object — such as dynamically allocated storage for data or object instance variables owned by the deallocated object. After performing the class-specific deallocation, the subclass method should incorporate superclass versions of `dealloc` through a message to `super`.

If you're building your application for compatibility with the older version of iOS (2.x), your `dealloc` method should release each object but should also set the reference to that object to `nil` before calling `super`.

Testing the View

Save your Xcode project by choosing File➪Save. Then, to see the magic you've just wrought, click the Build and Run button. You should see the Simulator launch, run the app, and display the falling words, as shown in Figure 10-15.

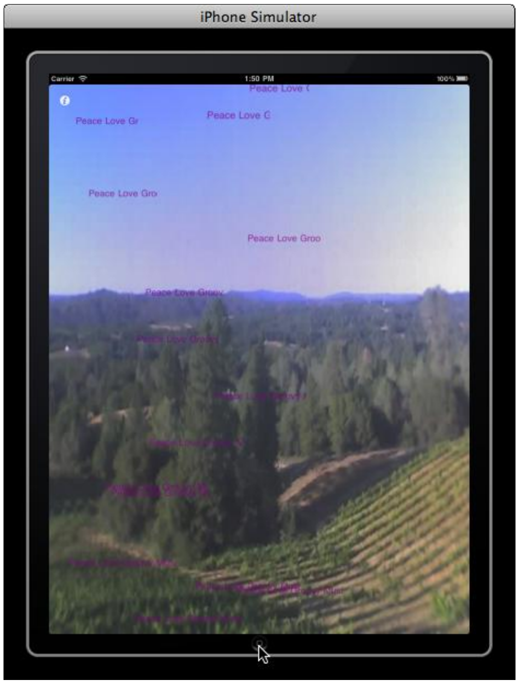


Figure 10-15: The view in the Simulator.

The animation is quite impressive, but now is not the time to sit on your laurels. There's more work to be done — setting up the modal controller, so that users can change the text and speed for the animation, and then saving these new preferences, for example. All that and more are covered in Chapter 11.

Chapter 11

Adding User Settings and Gestures

In This Chapter

- Loading and saving user preferences
- Adding outlets to the view controller
- Adding a modal view
- Connecting interface objects to your code in Interface Builder

One reason why it's easy to extend and enhance your iPad app is the fact that the template sets you up to take advantage of *delegation* — you're using a behavior-rich object supplied by the framework *as is,* and you're putting the code for program-specific behavior in a separate (delegate) object. You're basically using delegation to get the framework objects to do the work for you, as I describe in Chapter 7.

Government and military leaders know all about delegation. Ronald Reagan could have been talking about extending the functionality of an app's object-oriented programming when he said "Surround yourself with the best people you can find, delegate authority, and don't interfere." And General George S. Patton seemed to know all about combining delegation with encapsulation to enhance applications when he said, "Never tell people *how* to do things. Tell them *what* to do and they will surprise you with their ingenuity."

Encapsulation, you'll recall, is about keeping the details of how an object works hidden from the other objects that use it — you practice encapsulation in the code you add to animate the view in Chapter 10. That code doesn't know (or care) where the user's preference settings for text or animation speed come from; it simply does its job well.

In object-oriented programming, you can essentially copy all the characteristics of an existing class to make a new class — the new class *inherits* the methods and data structure of the existing class. When you combine delegation, encapsulation, and inheritance, changing or enhancing objects or their functionality becomes much easier because it reduces the impact of those changes on the rest of your application.

Inheritance allows you to do a number of things that make your programs more extensible: In a subclass, you can add new methods and instance variables to what is inherited from a superclass, refine or extend the behavior of an inherited method, and change the behavior of an inherited method. With encapsulation, you're hiding *how* things are being done from *what* is being done. Combining inheritance and encapsulation gives you *polymorphism* — using objects that do the same thing in different ways. (See *Objective-C For Dummies* for background info on these programming patterns.)

With DeepThoughts, the idea is to enable the user to enter his or her own words for the falling words animation, as well as change the speed of the animation. In this chapter, you enhance the DeepThoughts app to enable the user to change these preference settings using a *modal view* — a child window, such as a dialog in Mac OS X, that appears on top of the parent window (the main view) and requires the user to interact with it before returning to the parent window.

For these functions to work, you need to enable the app to save data entered by a user for the next time he or she fires up the app. In Chapter 10, you added code that uses these preference settings in your app to animate the view, but now you find out how to save data entered by the user using a modal controller that displays a view — a modal dialog — on top of the animated view. You create another view controller and use the inherited methods of the `UIViewController` superclass to implement the modal dialog that the user can use to enter text and change the animation speed. Because you've encapsulated the details of how to set the falling words and speed, it's a piece of cake to add code for setting and saving user preference settings.

Setting Up User Preference Settings

Most people these days have spent enough time around computers that they know what I mean when I throw the term *preferences* around. On your desktop, for example, you can set preferences at the system level for things like security, screen savers, printing, and file sharing — just to name a few. You can also set preferences for applications. For example, you can set all sorts of preferences in Xcode — not to mention all those preferences in your browser and word-processing programs.

The latter are application-specific settings used to configure the behavior or appearance of an application. You can create and save preference settings in your app, but you can also use the supplied Settings app to display and set your app-specific preferences. (The Settings app icon looks like a bunch of gears.) Whatever separate settings feature you come up with has to function within the framework of the Settings app; in effect, the Settings app makes you color within the lines.

What guidelines does the iPad impose for preference settings? Here's a short summary:

- **If you have preference settings that are typically configured once and then rarely changed:** Leave the task of setting preferences to the Settings app. On an iPad, this would apply to things like enabling/disabling Wi-Fi access, setting wallpaper displays, setting up Mail accounts, and any other preference settings you would set and then leave in place for a while.
- **If you have preference settings that the user might want to change regularly:** In this situation, you should consider having users set the options themselves in your app.

  TIP

  The iBooks app is a good example. Preferences that are configured once and rarely changed — in iBooks, the options to Sync Bookmarks and turn on Full Justification — are in the Settings app, while options you might change on the fly — such as brightness and font size in iBooks — are in the app itself.

With DeepThoughts, the idea is to change settings on the fly, whenever you feel like it, so it makes more sense to set up preferences from inside the app (because they are changed frequently). To find out how to set up your app to save preferences in a Settings bundle for the Settings app, see Chapter 19.

To save and read preference settings, you use a built-in, easy-to-use class that lets you read and set user preferences from your app — `NSUserDefaults`. In Chapter 10, you use `NSUserDefaults` to read and store preference data to a default database, using a key value — just as you would access keyed data from an `NSDictionary`. The difference here is that `NSUserDefaults` data is stored in the file system rather than in an object in memory — objects, after all, go away when the application terminates.

TIP

By the way, don't ask why the language experts put `Defaults` in the name rather than something to do with preference settings — fewer letters, maybe — but that's the way it is. Just don't let their naming idiosyncrasies confuse you.

Storing the data in the file system rather than in memory gives you an easy way to store application-specific information. With the help of `NSUserDefaults`, you can easily store the state the user was in when he or she quit the application — or store something simple like a text string — which just so happens to be precisely what you did in the code you added in Chapter 10 for DeepThoughts.

Identifying preference settings for NSUserDefaults

It's really easy to both access and update a preference — as long as you have NSUserDefaults by your side. The trick in this case is that you use the NSUserDefaults class to read and update the replacement text and speed. NSUserDefaults is implemented as a *singleton,* meaning there's only one instance of NSUserDefaults running in your application. To get access to that one instance, you invoke the class method standard UserDefaults:

```
  [NSUserDefaults  standardUserDefaults]
```

standardUserDefaults returns the NSUserDefaults object. As soon as you have access to the standard user defaults, you can store data there and then get it back when you need it. To store data, you simply give it a key and tell it to save the data using that key.

The way you tell it to save something is by using the setObject:forKey: method. In case your knowledge of Objective-C is a little rusty (or not there at all), that's the way any message that has two arguments is referred to.

The first argument, setObject:, is the object you want NSUserDefaults to save. This object must be NSData, NSString, NSNumber, NSDate, NSArray, or NSDictionary. In this case, savedData is an NSString, so you're in good shape.

The second argument is forKey:. In order to get the data back, and in order for NSUserDefaults to know where to save it, you have to be able to identify it to NSUserDefaults. You can, after all, have a number of preferences stored in the NSUserDefaults database, and the key tells NSUserDefaults which one you're interested in.

The keys you use are kWordsOfWisdom for the falling words replacement text, kSpeed for the animation speed, and kMaxSpeed for the maximum speed possible. You added them to the Constants.h file in Chapter 10.

Reading preferences into the app

To use the preference settings for the app's view, you link it up with the view controller — in this case, DeepThoughtsViewController. As Chapter 10 explains, the best place to do that is viewDidLoad, which is invoked right after the view has been loaded from the nib file.

After using the `NSTimer` class to create timers, the code checks to see whether the `kWordsOfWisdom` and `speed` settings have been moved into `NSUserDefaults`. The code moves the user's preferences into `NSUserDefaults` only *after* the application runs for the first time. If the settings haven't been moved into `NSUserDefaults` yet, the code uses the initial preference value (`"Peace Love Groovy Music"`) for `fallingWords`. If the settings *have* been moved into `NSUserDefaults`, the code reads them in and then sets `fallingWords` and `speed` to whatever the user's preference is. The rest of the code that animates the view can now use the preference settings.

Now that you've added the code to use the preference settings (in Chapter 10), you need to now decide how to enable the user to change these settings. One easy way for your app to offer the preference settings is in a modal dialog, which the user can use to enter the replacement text for `fallingWords` and change the `speed`.

Setting Up a Modal View Controller

You've encountered modal views before — whenever you had to click OK in a dialog to allow a system or application workflow to continue. A *modal view* provides self-contained functionality in the context of the current task or workflow. Think of it as a child window that requires the user to interact with it before returning to the parent.

A modal view interrupts the workflow, but if the context shift is clear and temporary (so that the user doesn't lose sight of the main task), a modal view can be the most agreeable way to offer the ability to change settings. As a design goal, keep tasks in a modal view fairly short and narrowly focused. You don't want your users to experience a modal view as a mini application within your application. Avoid creating a modal task that involves a hierarchy of modal views, because people can get lost and forget how to retrace their steps. And always provide an obvious and safe way to exit a modal view — such as a Done button.

The goal with DeepThoughts is to display a modal view over the animated view when the user taps the Light Info button (which you added to the user interface in Chapter 9). In that modal view, the user can enter text to replace the existing text for `fallingWords` and drag a speed slider for `speed`. The user can then tap a Done button to gracefully exit the modal view. You add these interface elements to the modal view using — you guessed it — Interface Builder.

First, though, you need to add a new view controller for the modal view, to be called `SettingsViewController`. That's the topic of the next section.

Adding a new view controller

The modal view, just like the animated main view, is accessed by a subclass of `UIViewController`.

To add the subclass, do the following:

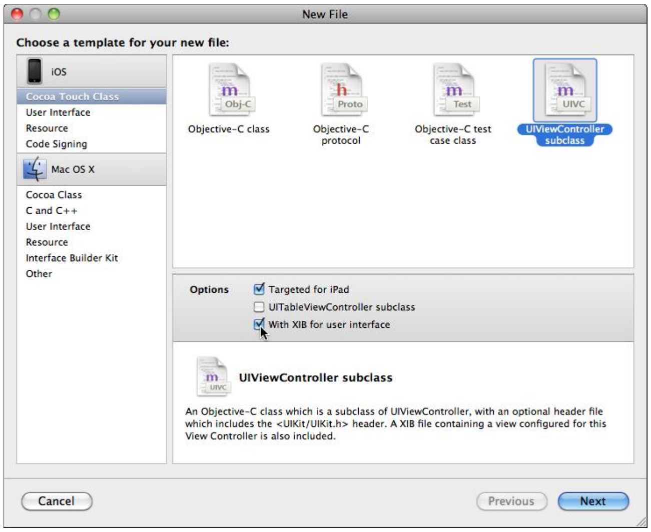
1. **Select the Classes group (if it's not already selected) in the Xcode project window's Groups & Files list and then choose File➪New File.**
2. **In the New File dialog that appears (as shown in Figure 11-1), select Cocoa Touch Class in the left column under iOS and select UIViewController Subclass in the row of icons on the right.**
3. **Select the Targeted for iPad check box (so that you get the appropriate subclass).**
4. **Select the With XIB for User Interface check box and click the Next button.**

   You select this option so that the `.xib` file is created along with the new view controller files. For this example, ignore the `UITableViewController` subclass option.
5. **In the next New File screen, enter the filename for the implementation file (`SettingsViewController.m`), as shown in Figure 11-2.**
6. **Select the Also Create "SettingsViewController.h" check box and click Finish.**

Xcode creates `SettingsViewController.h` and `SettingsViewController.m` in the Classes group. Xcode also creates `SettingsView.xib` in the Classes group. You may want to drag `SettingsView.xib` from that group into the Resources group, as I do in Figure 11-3, just to be consistent. (That's where the other nib files are located.)

You now have a bare-bones view controller for the modal view, called `SettingsViewController`.

Next, you need to add the code to the `SettingsViewController.h` (header) and `SettingsViewController.m` (implementation) files that connect to the interface elements (speed slider and text entry field) to offer the ability to change the speed and enter replacement text.

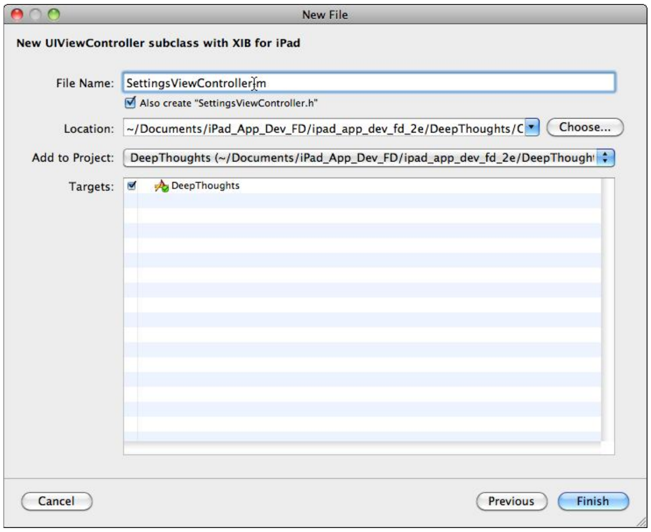


Options
- ☑ Targeted for iPad
- ☐ UITableViewController subclass
- ☑ With XIB for user interface

UIViewController subclass

An Objective-C class which is a subclass of UIViewController, with an optional header file which includes the <UIKit/UIKit.h> header. A XIB file containing a view configured for this View Controller is also included.

Figure 11-1: Create a new subclass of UIView-Controller.



New UIViewController subclass with XIB for iPad

File Name: SettingsViewController.m

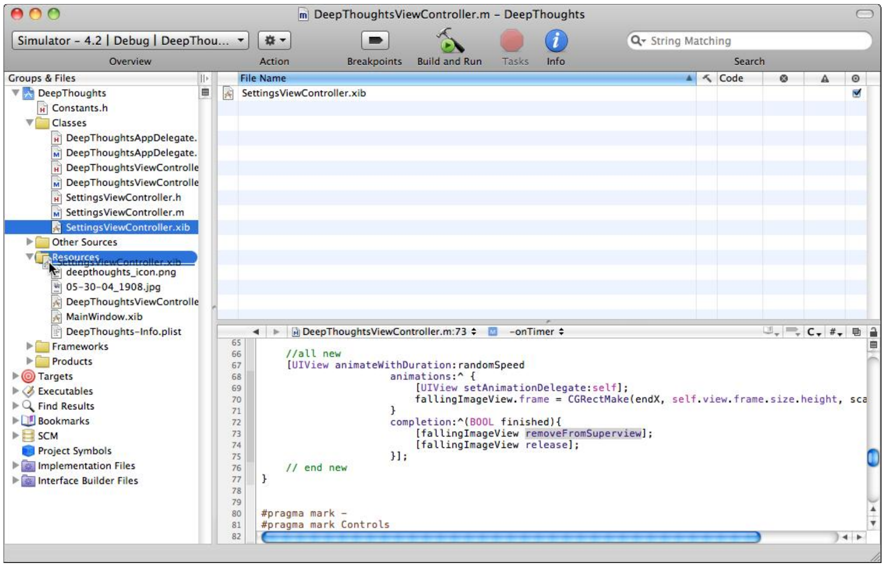
☑ Also create "SettingsViewController.h"

Location: ~/Documents/iPad_App_Dev_FD/ipad_app_dev_fd_2e/DeepThoughts/C

Add to Project: DeepThoughts (~/Documents/iPad_App_Dev_FD/ipad_app_dev_fd_2e/DeepThough

Targets: ☑ DeepThoughts

Figure 11-2: Here you name the subclass.



```
    //all new
    [UIView animateWithDuration:randomSpeed
                     animations:^ {
                         [UIView setAnimationDelegate:self];
                         fallingImageView.frame = CGRectMake(endX, self.view.frame.size.height, sca
                     }
                     completion:^(BOOL finished){
                         [fallingImageView removeFromSuperview];
                         [fallingImageView release];
                     }];
    // end new
}


#pragma mark -
#pragma mark Controls
```

Figure 11-3: Move the nib (.xib) file to the Resources group.

Adding outlets to the view controller

Before using Interface Builder to create the elements for the modal view, you should first put *outlets* in the code that will connect your methods to the Interface Builder interface objects.

The fact that a connection between an object and its outlets exists is actually stored in a nib file. When the nib file is loaded, each connection is reconstituted and reestablished, thus enabling you to send messages to the object. `IBOutlet` is the keyword that tags an instance-variable declaration so the Interface Builder application knows that a particular instance variable *is* an outlet — and can then enable the connection to it with Xcode.

In your code, it turns out that you need to create *two* outlets: one to point to the text entry field and one to point to the speed slider. To get this outlet business started, you need to *declare* each outlet, which you do with the help of the aforementioned `IBOutlet` keyword.

Add the bold lines of code in Listing 11-1 to the `SettingsViewController.h` file.

Listing 11-1: SettingsViewController.h

```
#import <UIKit/UIKit.h>
@protocol SettingsViewControllerDelegate;

@interface SettingsViewController : UIViewController
          <UITextFieldDelegate> {
<SettingsViewControllerDelegate>  delegate;
  float                            sliderValue;
  IBOutlet UITextField            *theTextField;
  IBOutlet UISlider               *slider;
  IBOutlet UILabel                *sliderDisplay;
}

- (IBAction) done;
- (IBAction) speedChanged:  (id) sender;
@property (nonatomic, assign) id
          <SettingsViewControllerDelegate> delegate;
@property (nonatomic, assign) UISlider* slider;

@end

@protocol SettingsViewControllerDelegate
- (void) settingsViewController:(SettingsViewController *)
          controller didFinishWithChange: (BOOL) changed;
- (void) changeSpeed: (double) newSpeed;
@property (nonatomic, retain) NSString    *fallingWords;

@end
```

Two action methods (`done` and `speedChanged`) for Interface Builder elements are declared (with `IBAction`), along with the `IBOutlet` statements, which declare the outlets that will be automatically initialized with a pointer to the `UITextField` (`theTextField`) and the `UISlider` (`slider`) when the application is launched. But while this will happen automatically, it won't *automatically* happen automatically. You have to help it out a bit.

In procedural programming — you know, all that Linux Kernel stuff — variables are generally fair game for all. But in object-oriented programming, a class's instance variables are tucked away inside an object and shouldn't be accessed directly. The only way for them to be initialized is for you to create what are called *accessor methods,* which allow the specific instance variable of an object to be read and (if you want) updated. Creating accessor methods is a two-step process that begins with an `@property` declaration, which tells the compiler that there are accessor methods. And that's what you did in Listing 11-1; you coded a corresponding `@property` declaration for the `IBOutlet` declaration for `UISlider`.

The methods that provide access to the instance variables of an object are called *accessor methods,* and they effectively *get* (using a *getter method*) and *set* (using a *setter method*) the values for an instance variable. Although you can code those methods yourself, it can be rather tedious. This is where properties come in. The Objective-C Declared Properties feature provides a simple way to declare and implement an object's accessor methods. The compiler can synthesize accessor methods according to the way you told it to in the property declaration. Objective-C creates the getter and setter methods for you by using an `@property` declaration in the interface file, combined with the `@synthesize` declaration in the implementation file.

All that being said, at the end of the day, you need to do three things in your code to have the compiler create accessors for you:

1. **Declare an instance variable in the interface file.**
2. **Add an `@property` declaration of that instance variable in the same interface file (usually with the `nonatomic` attribute).**

   The declaration specifies the name and type of the property as well as some attributes that provide the compiler with information about how exactly you want the accessor methods to be implemented.

   For example, the declaration

   ```
   @property (nonatomic, assign) UISlider* slider;
   ```

   declares a property named `slider`, which is a pointer to a `UISlider` object. As for the two attributes — `nonatomic` and `assign` — `nonatomic` tells the compiler to create an accessor to return the value directly, which is another way of saying that the accessors can be interrupted while in use. (This works fine for applications like this one.)

   The second value, `assign`, tells the compiler to create an accessor method that sends an `assign` message to any object that's assigned to this property.
3. **Use `@synthesize` in the implementation file so that Objective-C generates the accessors for you.**

   The `@property` declaration only declares that there *should be* accessors. It's the `@synthesize` statement that tells the compiler to *create* them for you. You add this statement in the next section, along with more code, to the `SettingsViewController.m` implementation file.

Using delegation

Delegation is a design pattern used extensively in the `UIKit` and `AppKit` frameworks to customize the behavior of an object without subclassing.

Instead of having to bother with subclassing, one object delegates the task of implementing one of its methods to another object. You can use Interface Builder to connect objects to their delegates; or you can set the connection programmatically through the delegating object's `setDelegate:` method or `delegate` property.

To implement a delegated method, you put the code for your application-specific behavior in a separate *(delegate)* object. When a request is made of the delegator, the delegate's method that implements the application-specific behavior is invoked by the delegator.

The methods that a class delegates are defined in a *protocol*. You declared protocols in Listing 11-1 with the `@protocol` directive:

```
@protocol SettingsViewControllerDelegate
- (void) settingsViewController:(SettingsViewController *)
          controller didFinishWithChange: (BOOL) changed;
- (void) changeSpeed: (double) newSpeed;
@property (nonatomic, retain) NSString    *fallingWords;

@end
```

Protocols declare methods that can be implemented by any class. They're useful for declaring methods that other delegate objects are expected to implement.

Adding methods for the interface objects

Next, you need to add the methods to the `SettingsViewController.m` (implementation) file for managing the modal view and performing actions connected to the slider, the text field, and the Done button (all of which get added in the "Connecting the Interface Objects in Interface Builder" section, later in this chapter).

Before adding your code to this view controller object, it helps to know what each section of the template-provided code does, and it's especially helpful if you use `# pragma mark` statements to mark off each section so you can quickly jump to the relevant section when needed. (For more on how to use `# pragma mark` statements, check out Chapter 10.) I added these statements along with the new code (in bold) in Listing 11-2.

TIP

The `# pragma mark` statement is simply a way to organize your methods in the Method list pop-up in the Xcode Text Editor's Navigation bar (to the left of the Bookmarks menu). You use it with a label (such as `View life cycle`) to add the label in bold to the Method list so that you can identify and keep separate the methods logically in the list.

Add the bold statements in Listing 11-2 to the skeletal view controller code in the new `SettingsViewController.m` file (you can delete the "designated initializer" and "Implement viewDidLoad" commented text blocks in the `View life cycle` section).

Listing 11-2: SettingsViewController.m

```
#import "SettingsViewController.h"
#import "DeepThoughtsViewController.h"
#import "Constants.h"

@implementation SettingsViewController

@synthesize delegate, slider ;

#pragma mark -
#pragma mark View life cycle

- (void)viewDidLoad {
  [super viewDidLoad];
  theTextField.backgroundColor = [UIColor whiteColor];
  theTextField.textColor = [UIColor blueColor];
  slider.value = + kMaxSpeed -
          ((DeepThoughtsViewController*) (self.
          parentViewController)).speed;
  sliderDisplay.text = [NSString stringWithFormat:
          @"%.2f",slider.value];
}

#pragma mark -
#pragma mark textField

-(BOOL)textFieldShouldBeginEditing:(UITextField *)
          textField {
  [textField setReturnKeyType:UIReturnKeyNext];
  return YES;
}

-(BOOL)textFieldShouldReturn:(UITextField *)textField {
  [textField resignFirstResponder];
          return YES;
}

#pragma mark -
#pragma mark Controls

- (IBAction) speedChanged: (id) sender {
  [delegate changeSpeed: [(UISlider *)sender value] ];
  sliderDisplay.text = [NSString stringWithFormat:
          @"%.2f",[(UISlider *)sender value]];
}

- (IBAction)done {
  if(! [theTextField.text isEqualToString: @"" ]) {
delegate.fallingWords= theTextField.text;
          [self.delegate settingsViewController:self
          didFinishWithChange:YES];
  }
  else
    [self.delegate settingsViewController:self
          didFinishWithChange:NO];
}


#pragma mark -
#pragma mark Orientation

- (BOOL)shouldAutorotateToInterfaceOrientation:
          (UIInterfaceOrientation)interfaceOrientation {
    // Overriden to allow any orientation.
    return YES;
}

#pragma mark -
#pragma mark Memory management

- (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 {
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    self.slider = nil;
}

- (void)dealloc {
    self.slider = nil;
    [super dealloc];
}

@end
```

Okay, let me walk you through this one. Although the `@property` declaration in the header file in Listing 11-1 tells the compiler that there are accessor methods, they still have to be created. Fortunately, Objective-C will create these accessor methods for you whenever you include an `@synthesize` statement for a property, which is what you did near the top of Listing 11-2:

```
@synthesize delegate , slider;
```

The `@synthesize` statement tells the compiler to create accessor methods for you — one for each `@property` declaration.

Next, you add the `viewDidLoad` method to set the background color and text color for the text field, to set the speed for the slider, and to set the value of the slider to display as text:

```
- (void)viewDidLoad {
  [super viewDidLoad];
  theTextField.backgroundColor = [UIColor whiteColor];
  theTextField.textColor = [UIColor blueColor]
  slider.value = + kMaxSpeed -
           ((DeepThoughtsViewController*) (self.
           parentViewController)).speed;
  sliderDisplay.text = [NSString stringWithFormat:
           @"%.2f",slider.value];
}
```

Following that code (in Listing 11-2), you add the methods to obtain the text for `textField`:

```
#pragma mark -
#pragma mark textField

-(BOOL)textFieldShouldBeginEditing:(UITextField *)
           textField {
  [textField setReturnKeyType:UIReturnKeyNext];
  return YES;
}
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
  [textField resignFirstResponder];
   return YES;
}
```

The `UITextFieldDelegate` protocol defines the messages sent to a text field delegate as part of the sequence of editing its text. When the user performs an action that would normally start an editing session, the text field calls the `textFieldShouldBeginEditing:` method first to see whether editing should actually proceed. In most circumstances, you would simply return `YES` from this method to allow editing to proceed.

The text field calls the `textFieldShouldReturn:` method whenever the user taps the Return button on the keyboard to find out whether it should process the Return. You can use this method to implement any custom behavior when the Return button is tapped, but for your purposes, you simply return `YES` (which is the default), although you could return `NO` to ignore the Return button.

Next, you provide a `speedChanged` method (of type `IBAction`) to handle a change in speed, which uses the delegate's `changeSpeed` method to immediately change the speed of the animation in the view as the user changes it in the modal view:

```
#pragma mark -
#pragma mark Controls

- (IBAction) speedChanged: (id) sender {
  [delegate changeSpeed: [(UISlider *)sender value] ];
  sliderDisplay.text = [NSString stringWithFormat:
           @"%.2f",[(UISlider *)sender value]];
}

- (IBAction)done {
  if(! [theTextField.text isEqualToString: @"" ])  {
           delegate.fallingWords= theTextField.text;
      [self.delegate settingsViewController:self
           didFinishWithChange:YES];
}
```

You also supply a `done` method that handles the possibility of a blank text field. The code assigns the text field's text to `delegate.fallingWords` *only* if the field is *not* `theTextField.text isEqualToString: @""`.

Finally, to make your app act like a good citizen and relinquish `slider` to free memory, you added the following bold code in Listing 11-2 to the `viewDidUnload` and `dealloc` methods:

```
- (void)viewDidUnload {
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    self.slider = nil;
}

- (void)dealloc {
    self.slider = nil;
    [super dealloc];
}
```

Initializing and setting the modal view style

Fine so far, but there are still a few problems: Where is this modal view initialized and set up? And how would DeepThoughtsViewController even know about the new speed setting and change it accordingly?

Here's the deal: You need to modify the code in the DeepThoughtsViewController.h (header) and DeepThoughtsViewController.m (implementation) files to add the method for initializing and setting up the modal view when the user taps the Light Info button. You also need to add the code to change the speed. Add the bold lines of code in Listing 11-3 to the DeepThoughtsViewController.h file.

Listing 11-3: DeepThoughtsViewController.h

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

@interface DeepThoughtsViewController : UIViewController
          <SettingsViewControllerDelegate > {

  UIImage     *fallingImage;
  NSString    *fallingWords;
  UIImageView *imageView;
  double       speed;
}
- (IBAction)settings;
- (void) changeSpeed: (double) newSpeed;
- (void) settingsViewController: (SettingsViewController
          *)controller didFinishWithChange: (BOOL)
          changed;

@property (readwrite)  double speed;
@property (nonatomic, retain) UIImageView *imageView;
@property (nonatomic, retain) NSString    *fallingWords;

@end
```

You modify the declarations to include the SettingsViewController.h declarations, and you add the SettingsViewControllerDelegate protocol so that you can use the delegate's methods. You then declare the changeSpeed and settingsViewController:didFinishWithChange: methods.

Next, you add the bold lines of code in Listing 11-4 to the DeepThoughtsViewController.m file in the Controls section (marked by #pragma mark Controls in Chapter 10).

Listing 11-4: DeepThoughtsViewController.m (Controls Section)

```
#pragma mark -
#pragma mark Controls

- (IBAction)settings {
    SettingsViewController *controller =
            [[SettingsViewController alloc] initWithNibName
            :@"SettingsViewController" bundle:nil];

controller.modalTransitionStyle =
            UIModalTransitionStyleFlipHorizontal;
controller.modalPresentationStyle =
            UIModalPresentationFormSheet;
controller.delegate = self;

  [self presentModalViewController:controller
            animated:YES];

  [controller release];
}

- (void) changeSpeed: (double) newSpeed {
  speed = kMaxSpeed-newSpeed;
  [[NSUserDefaults standardUserDefaults]setDouble:
            newSpeed forKey:kSpeed];
}

- (void) settingsViewController:(SettingsViewControl
            ler *)controller  didFinishWithChange: (BOOL)
            changed{

  if (changed) {
    [[NSUserDefaults standardUserDefaults]
            setObject:fallingWords  forKey:kWordsOfWisdom];

  }
    [self dismissModalViewControllerAnimated:YES];
}
```

The `settings` method (of type IBAction) initializes the modal view. The `UIViewController` class offers the `modalTransitionStyle` property to set the transition to use when the modal view appears. (I chose `UIModalTransitionStyleFlipHorizontal` to do a horizontal 3D flip from right to left, a fairly standard transition.)

TIP

You could choose other transitions, such as a *partial-curl* (`UIModalTransitionStylePartialCurl`), in which one corner of the current view curls up to reveal the modal view underneath. When the user leaves the modal view, the current view uncurls to its original position. (Of course, a modal view revealed by a partial-curl can't itself reveal *another* modal view with a partial-curl — that would be a wipe-out, in surfer terms.)

The UIViewController class also offers the modalPresentationStyle property that specifies the appearance of the modal view on the iPad. Options for this property let you present the modal view so that it fills the entire iPad display, or only part of the display:

- UIModalPresentationFullScreen: This option uses the entire display, which is good for something that is complex — such as choosing a Genius mix from your music playlists in the iPod app.
- UIModalPresentationPageSheet: This option offers a fixed width of 768 points; the sheet height is the current height of the display. In portrait, the Page Sheet view covers the entire display; in landscape orientation, the area of the display that is visible on both sides of the Page Sheet view is dimmed to prevent user interaction. Some apps use this style for composing a text message or note.
- UIModalPresentationFormSheet: I use this option (in Listing 11-4) for DeepThoughts. The form sheet is a fixed-dimension view of 540 x 620 points centered in the display. The area of the display that is visible outside the Form Sheet view is dimmed to prevent user interaction. When the keyboard is visible in landscape orientation, the Form Sheet view moves up to just below the status bar so that you can still see it.
- UIModalPresentationCurrentContext: This option uses the same size as its parent view. This style is good for displaying a modal view within a Split View pane, popover, or other view that doesn't fill the display.

After setting the transition and presentation style for the modal view, the code uses the presentModalViewController:animated: instance method to present the animated modal view (and attach it to the view hierarchy). At the end of the code (in Listing 11-4), you use the dismissModalViewControllerAnimated: method to animate the view as it's dismissed.

Saving the preference settings

The code in Listing 11-4 then implements the changeSpeed method to change the animation speed if the slider in the modal view changes. After this, you add the settingsViewController:didFinishWithChange: method to save the user preference settings. Although the user can change the speed and see the results on the fly, the speed setting is saved (for later use when the app runs again) only after the user taps Done in the modal view. (You connect the Done button in the "Adding the Done button" section, later in this chapter.)

As you may recall from earlier in this chapter in "Identifying preference settings for NSUserDefaults," you use standardUserDefaults (a NSUserDefaults class method) to gain access to the standard user defaults; you can store data there and then get it back when you need it. To store data, you use the `setObject:forKey:` method. The first argument, `setObject:`, is the object you want `NSUserDefaults` to save (`falling Words`); the second argument is `forKey:` (`kWordsOfWisdom`), which is how `NSUserDefaults` identifies it. For the slider value, you use `setDouble: newSpeed forKey:kSpeed`.

The next step is to create the interface elements for the Done button, speed slider, and text field elements of the modal view and then *connect* these interface elements — along with the Light Info button in the animated view — to the methods in your view controllers.

REMEMBER

Don't forget to save your changes in Xcode; otherwise, Interface Builder won't be able to find the new code. Choose File⇨Save.

Connecting the Interface Objects in Interface Builder

You've created the outlets and their accessor methods in your code. The next sections are about wiring the two nib (`.xib`) files to your code using Interface Builder — so that when the nib files are loaded, the nib loading code will create these connections automatically.

To start, open the modal view controller in Interface Builder: click Resources in the Groups & Files list and then double-click the `SettingsViewController.xib` file to launch Interface Builder.

You can then click the View icon in the `SettingsViewController.xib` window of Interface Builder so that you can add the user interface objects to it. If the Library window isn't already open, choose Tools⇨Library.

Adding the Done button

To add the Done button, click the Classes tab at the top of the Library window, and select `UIResponder` from the drop-down menu below the Classes tab (you can see the selection in Figure 11-4). (This narrows your search through the Library window so that you can find the class you need quickly.) You need the `UIButton` class, the same class you used for the Info button in Chapter 9.

Drag the `UIButton` class object from the Library onto the View window in the top-left corner (refer to Figure 11-4). Guides appear to help you place the object where you want it.

After dragging it, the `UIButton` class object appears under View in the `SettingsViewController.xib` window in List view as a Rounded Rect Button, which is its default setting.

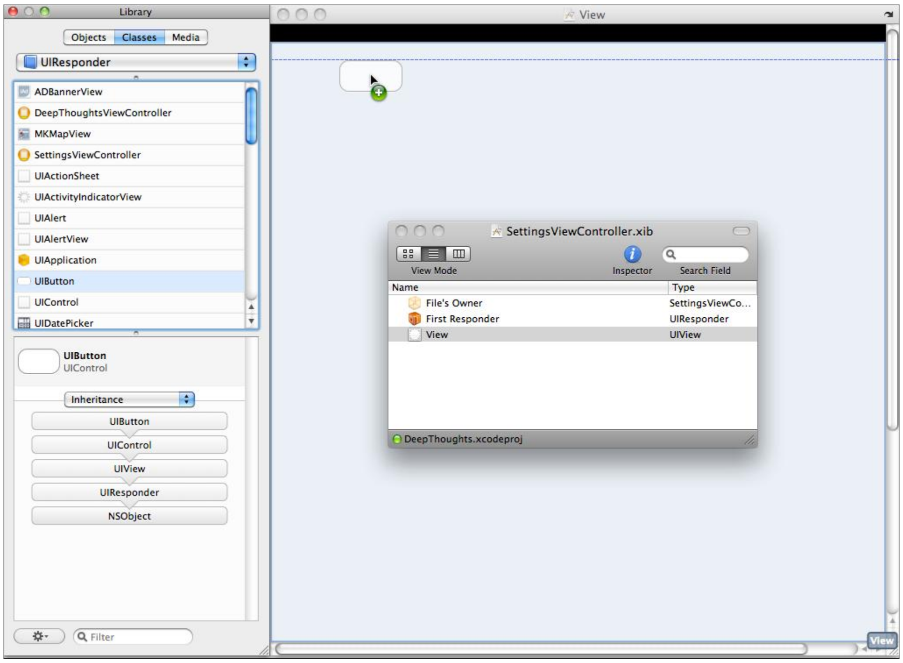
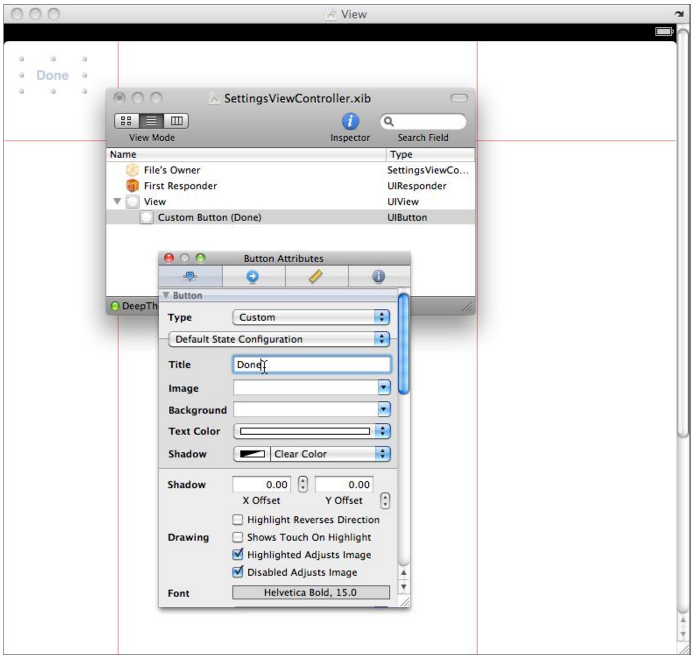


Figure 11-4: Drag the UIButton class object to the View window.

Select Rounded Rect Button in the `SettingsViewController.xib` window, and click the Attributes tab of the Inspector window (or choose Tools➪Attributes Inspector) if it isn't already selected. The Attributes Inspector shows Button Attributes, as shown in Figure 11-5. Choose Custom in the Type drop-down menu in Button Attributes and then click the Text Color tile to change it to white, and click the Clear Color button next to Shadow to remove the shadow. Don't forget to type **Done** for the button's title. (See Figure 11-5.)

To connect the Done button to your code, select Custom Button (Done) in the `SettingsViewController.xib` window, and click the Connections tab in the Inspector window (or choose Tools➪Connections Inspector) and scroll down to the bottom of the Events section (above Referencing Outlets). Drag from the connection point for a Touch Up Inside event to File's Owner, as shown in Figure 11-6. Then choose `done` from the pop-up menu that appears to connect the object to your code's outlet (`done`), as shown in Figure 11-7.

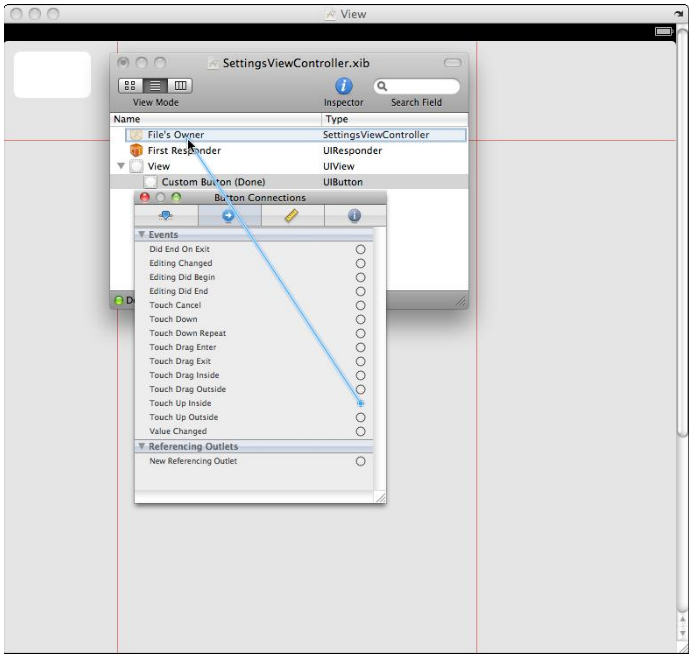
That's it; the Done button is done.



Name	Type
File's Owner	SettingsViewCo…
First Responder	UIResponder
View	UIView
Custom Button (Done)	UIButton

Type: Custom
Default State Configuration
Title: Done
Image:
Background:
Text Color:
Shadow: Clear Color
Shadow: 0.00 X Offset, 0.00 Y Offset
Drawing: ☐ Highlight Reverses Direction
☐ Shows Touch On Highlight
☑ Highlighted Adjusts Image
☑ Disabled Adjusts Image
Font: Helvetica Bold, 15.0

Figure 11-5: Change the button attributes.



Name	Type
File's Owner	SettingsViewController
First Responder	UIResponder
View	UIView
Custom Button (Done)	UIButton

Figure 11-6: Make a connection from the custom button to the File's Owner.

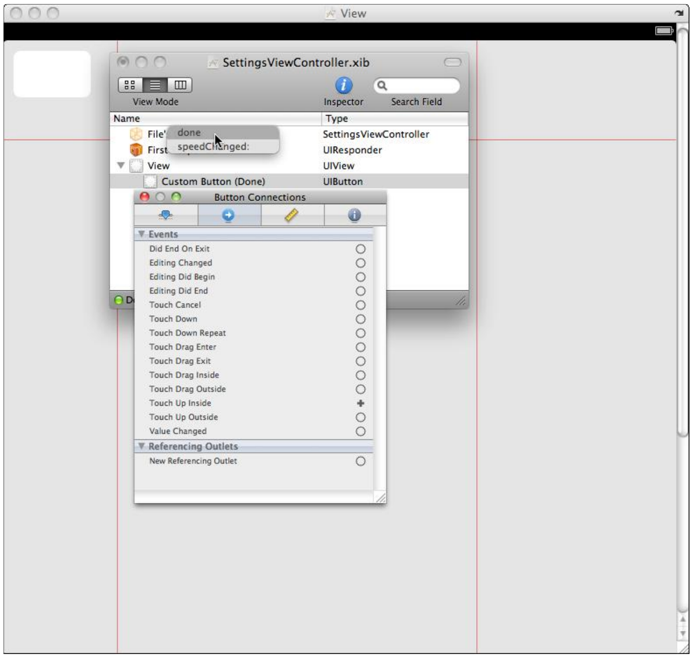


Figure 11-7: Connect the custom button to the done method.

Adding the slider and text field

Next, click the Objects tab at the top of the Library window, and then select Inputs & Values from the Interface Builder Library pop-up menu to see the input objects you can use. Information about each object appears in the lower portion of the Library window. The slider, for example, is of the `UISlider` class; it displays a horizontal bar representing a range of values.

Drag the slider from the Library window over to the View window, just like you did with the `UIButton` class object. (Refer to Figure 11-4.)

You can select the horizontal slider in the View window and then drag its edges to make it longer. To set the slider's values, click the Attributes tab in the Inspector window (or choose Tools➪Attributes Inspector) and then change the Minimum, Maximum, and Initial values, as shown in Figure 11-8.

To connect the horizontal slider to your code, first click the triangle next to the View icon in the `SettingsViewController.xib` window, if you haven't done this already, to reveal its contents (which now includes Horizontal Slider) and then select Horizontal Slider. Then click the Connections tab in the Inspector window (or choose Tools➪Connections Inspector) and scroll down to the Referencing Outlets section. Drag from the connection point for a new referencing outlet to File's Owner, as shown in Figure 11-9. Then choose `slider` from the pop-up menu that appears to connect the object to your code's outlet (`slider`), as shown in Figure 11-10.

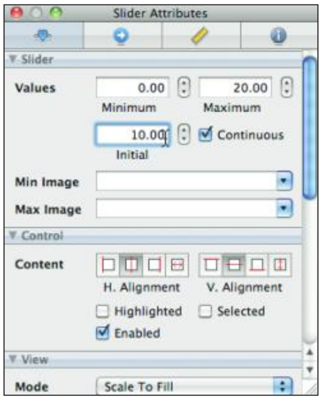


Figure 11-8: Set the slider's Minimum, Maximum, and Initial values.

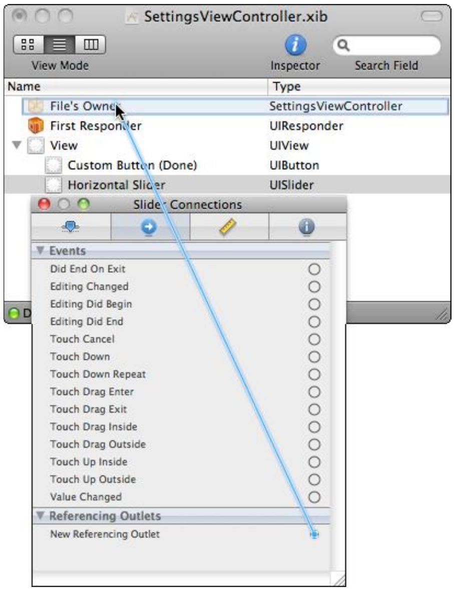


Figure 11-9: Make a connection from the Horizontal Slider to the File's Owner.

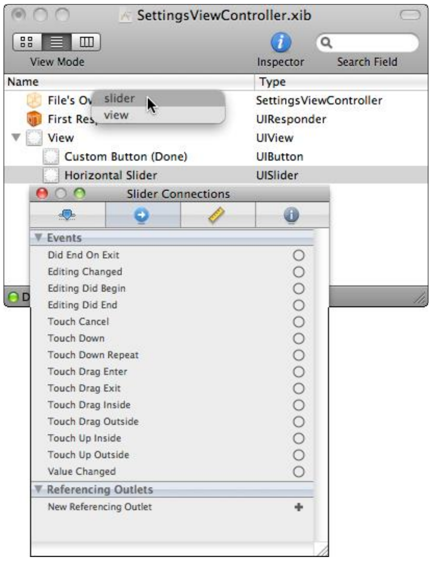


Figure 11-10: Connect the Horizontal Slider to slider in your code.

Because you want to capture the speed setting when the user slides the slider and then immediately use that new setting with the view controller, scroll down to the bottom of the Events section (above Referencing Outlets) in the Connections Inspector, and drag from the connection point for a Value Changed event to File's Owner, as shown in Figure 11-11. Then choose `speedChanged` from the pop-up menu that appears to connect the object to your code's outlet (`speedChanged`), as shown in Figure 11-12.

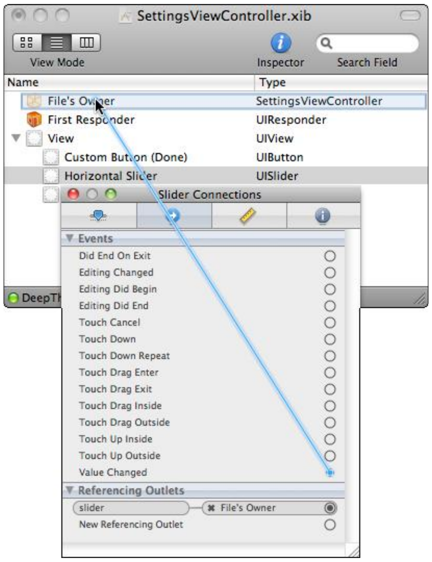


Figure 11-11: Connect a Value Changed event in the slider to the File's Owner.

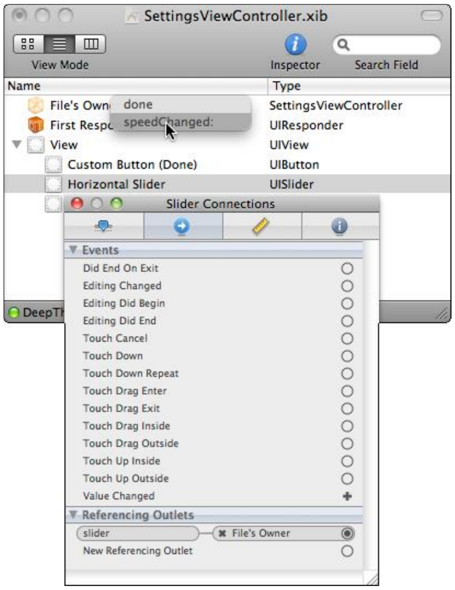
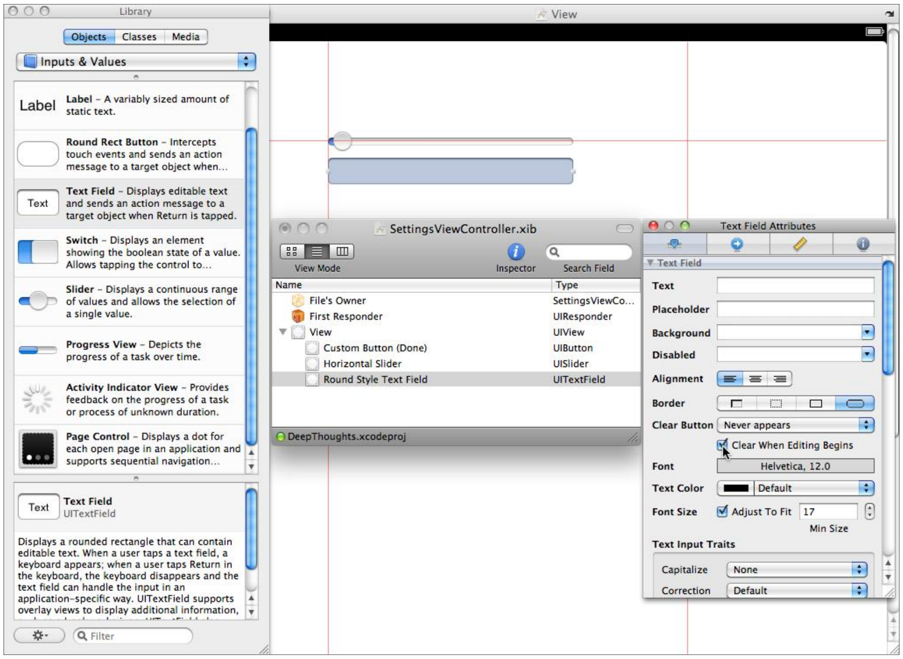


Figure 11-12: Connect the Value Changed event to the speed-Changed method.

Now, perform a similar procedure with the Text Field object in the Interface Builder Library window. (If the Library window isn't already open, choose Tools➪Library.) Drag the Text Field for text entry from the Library window to the View window, where it becomes the Round Style Text Field.

You can then select your new text field in the View window and then drag its edges to make it longer. To change its attributes, first select Round Style Text Field (which is now below Horizontal Slider) in the `SettingsViewController.xib` window. Then click the Attributes tab in the Inspector window (or choose Tools➪Attributes Inspector), as shown in Figure 11-13.

To connect the text field's Delegate connector to `SettingsViewController`, click the Connections tab in the Inspector window (or choose Tools➪Connections Inspector) and drag from the connection point for `delegate` (at the top of the Text Field Connections window) to File's Owner, as shown in Figure 11-14.



Label – A variably sized amount of static text.

Round Rect Button – Intercepts touch events and sends an action message to a target object when…

Text Field – Displays editable text and sends an action message to a target object when Return is tapped.

Switch – Displays an element showing the boolean state of a value. Allows tapping the control to…

Slider – Displays a continuous range of values and allows the selection of a single value.

Progress View – Depicts the progress of a task over time.

Activity Indicator View – Provides feedback on the progress of a task or process of unknown duration.

Page Control – Displays a dot for each open page in an application and supports sequential navigation…

Text Field
UITextField

Displays a rounded rectangle that can contain editable text. When a user taps a text field, a keyboard appears; when a user taps Return in the keyboard, the keyboard disappears and the text field can handle the input in an application-specific way. UITextField supports overlay views to display additional information,

SettingsViewController.xib

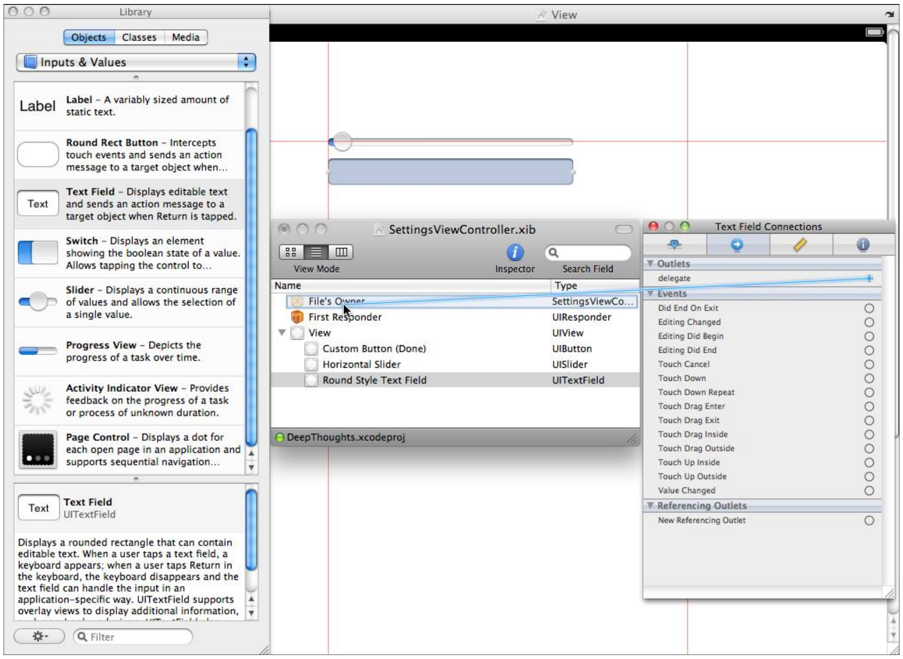
Name	Type
File's Owner	SettingsViewCo…
First Responder	UIResponder
View	UIView
Custom Button (Done)	UIButton
Horizontal Slider	UISlider
Round Style Text Field	UITextField

DeepThoughts.xcodeproj

Text Field Attributes

Text Field
Text
Placeholder
Background
Disabled
Alignment
Border
Clear Button Never appears
[✓] Clear When Editing Begins
Font Helvetica, 12.0
Text Color Default
Font Size [✓] Adjust To Fit 17 Min Size
Text Input Traits
Capitalize None
Correction Default

Figure 11-13: After dragging the Text Field you can change its attributes.



Label – A variably sized amount of static text.

Round Rect Button – Intercepts touch events and sends an action message to a target object when…

Text Field – Displays editable text and sends an action message to a target object when Return is tapped.

Switch – Displays an element showing the boolean state of a value. Allows tapping the control to…

Slider – Displays a continuous range of values and allows the selection of a single value.

Progress View – Depicts the progress of a task over time.

Activity Indicator View – Provides feedback on the progress of a task or process of unknown duration.

Page Control – Displays a dot for each open page in an application and supports sequential navigation…

Text Field
UITextField

Displays a rounded rectangle that can contain editable text. When a user taps a text field, a keyboard appears; when a user taps Return in the keyboard, the keyboard disappears and the text field can handle the input in an application-specific way. UITextField supports overlay views to display additional information,

SettingsViewController.xib

Name	Type
File's Owner	SettingsViewCo…
First Responder	UIResponder
View	UIView
Custom Button (Done)	UIButton
Horizontal Slider	UISlider
Round Style Text Field	UITextField

DeepThoughts.xcodeproj

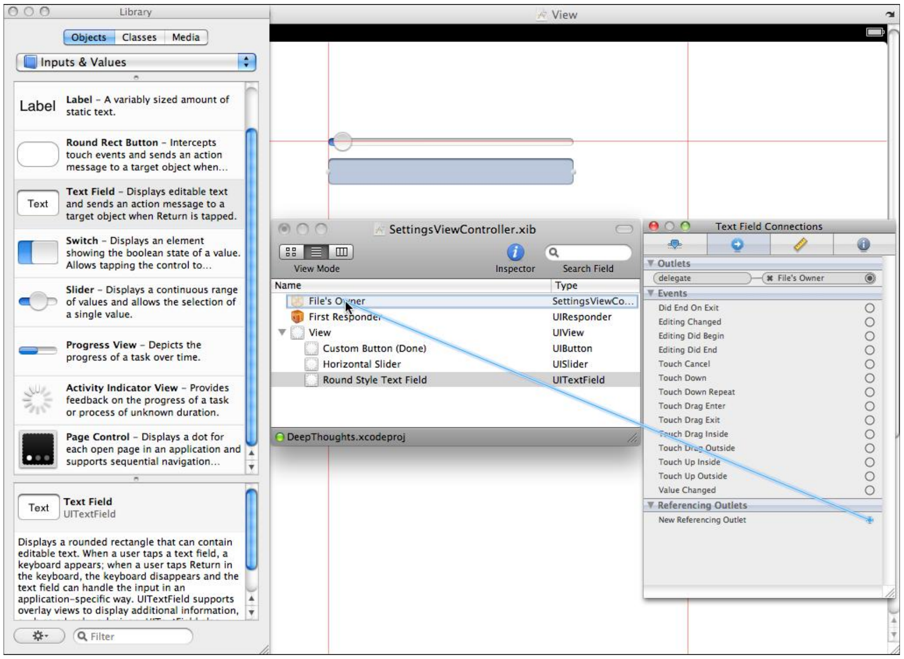
Text Field Connections

Outlets
delegate
Events
Did End On Exit
Editing Changed
Editing Did Begin
Editing Did End
Touch Cancel
Touch Down
Touch Down Repeat
Touch Drag Enter
Touch Drag Exit
Touch Drag Inside
Touch Drag Outside
Touch Up Inside
Touch Up Outside
Value Changed
Referencing Outlets
New Referencing Outlet

Figure 11-14: Connect the Text Field to the modal view controller.

Finally, with Round Style Text Field still selected, scroll down the Text Field Connections window to the Referencing Outlets section. Drag from the connection point for a new referencing outlet to File's Owner, as shown in Figure 11-15. Then choose `theTextField` from the pop-up menu that appears to connect the object to your code's outlet (`theTextField`), as shown in Figure 11-16.

To display the slider's value, use a Label object. Perform the same procedure by dragging the Label object in the Interface Builder Library window to the View window. You can then select your new label in the View window and change its attributes by clicking the Attributes tab in the Inspector window (or choose Tools➪Attributes Inspector). To connect the label to your code, click the Connections tab in the Inspector window (or choose Tools➪Connections Inspector). Drag from the connection point for a new referencing outlet (in the Referencing Outlets section) to File's Owner. Then choose `sliderDisplay` from the pop-up menu that appears to connect the object to your code's outlet.



Label – A variably sized amount of static text.

Round Rect Button – Intercepts touch events and sends an action message to a target object when...

Text Field – Displays editable text and sends an action message to a target object when Return is tapped.

Switch – Displays an element showing the boolean state of a value. Allows tapping the control to...

Slider – Displays a continuous range of values and allows the selection of a single value.

Progress View – Depicts the progress of a task over time.

Activity Indicator View – Provides feedback on the progress of a task or process of unknown duration.

Page Control – Displays a dot for each open page in an application and supports sequential navigation...

Text Field
UITextField

Displays a rounded rectangle that can contain editable text. When a user taps a text field, a keyboard appears; when a user taps Return in the keyboard, the keyboard disappears and the text field can handle the input in an application-specific way. UITextField supports overlay views to display additional information,

Figure 11-15: Connect the Text Field's referencing outlet to the File's Owner.

To complete the modal dialog, change the view's background color. Select View in the `SettingsViewController.xib` window, and change the Background Color option to View Flipside Background Color (see Figure 11-17), which returns the system color used for the back side of a view while it is being flipped. Now it's ready for use.

REMEMBER

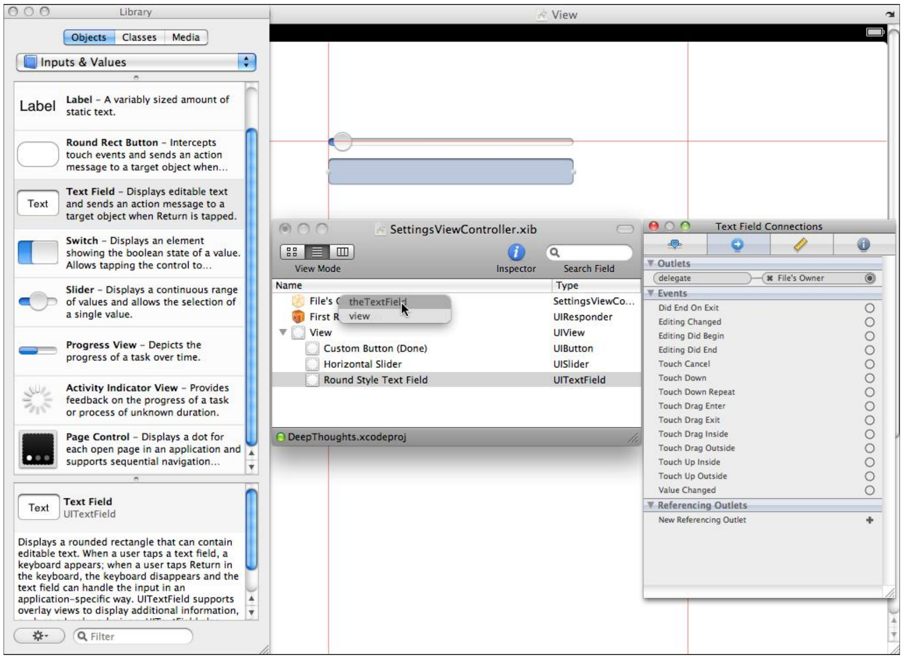
Don't forget to save your changes in Interface Builder — you can choose File➪Save if you are continuing to edit the same file, or you can click Save in the warning dialog if you close the file or quit Interface Builder.

For the next step, you need to close the `SettingsViewController.xib` file and double-click the `DeepThoughtsViewController.xib` file to open Interface Builder again.

Connecting the Info button

You have one more chore: to connect the Light Info button, which you added to the user interface in Chapter 9, to the method in `DeepThoughtsViewController` that initializes the modal view.

To connect the Light Info button to this method, double-click the `DeepThoughtsViewController.xib` file to launch Interface Builder (if you haven't done this already). Next, click the triangle next to View in the `DeepThoughtsViewController.xib` window (in List view) to open it, and select Light Info Button underneath View. To make the connection, click the Connections tab in the Inspector window (or choose Tools➪Connections Inspector), and drag the outlet for a Touch Up Inside event to the File's Owner, as shown in Figure 11-18. The `settings` method pops up so that you can select it for the Info button.



Label – A variably sized amount of static text.

Round Rect Button – Intercepts touch events and sends an action message to a target object when...

Text Field – Displays editable text and sends an action message to a target object when Return is tapped.

Switch – Displays an element showing the boolean state of a value. Allows tapping the control to...

Slider – Displays a continuous range of values and allows the selection of a single value.

Progress View – Depicts the progress of a task over time.

Activity Indicator View – Provides feedback on the progress of a task or process of unknown duration.

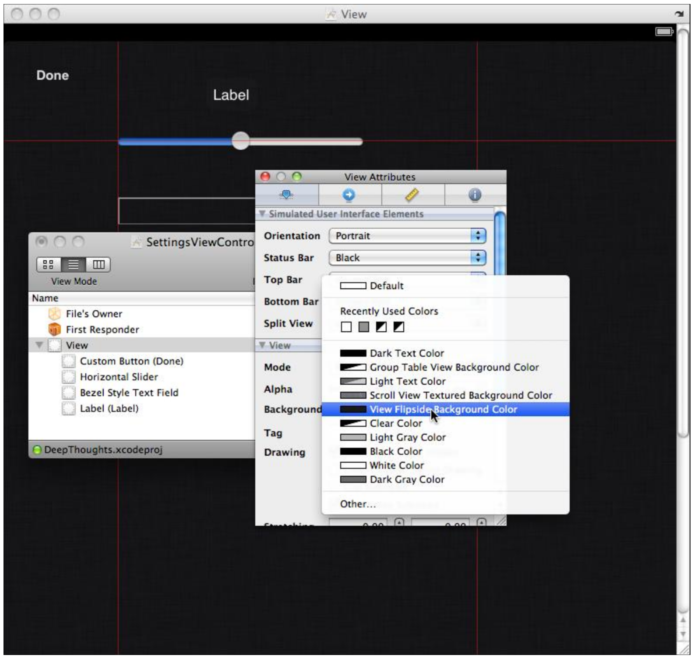
Page Control – Displays a dot for each open page in an application and supports sequential navigation...

Text Field
UITextField

Displays a rounded rectangle that can contain editable text. When a user taps a text field, a keyboard appears; when a user taps Return in the keyboard, the keyboard disappears and the text field can handle the input in an application-specific way. UITextField supports overlay views to display additional information,

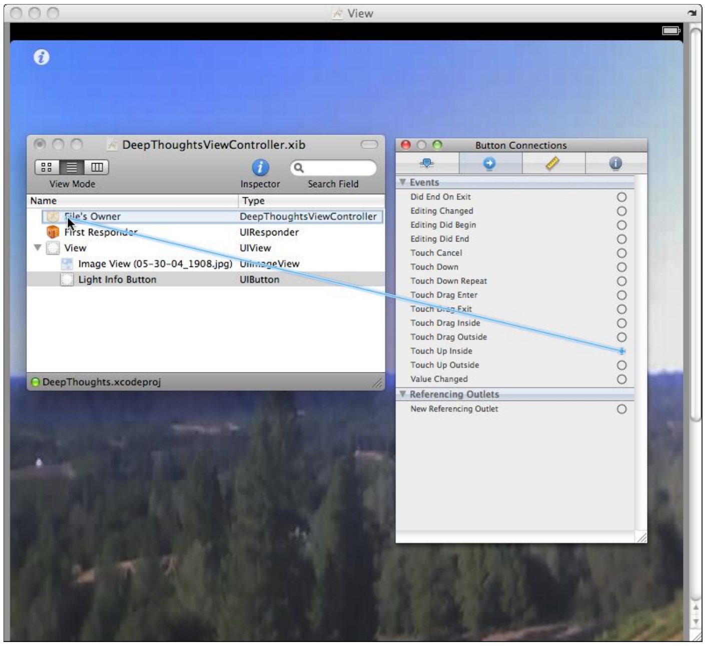
Name	Type
File's Owner	SettingsViewCo...
First Responder	UIResponder
View	UIView
Custom Button (Done)	UIButton
Horizontal Slider	UISlider
Round Style Text Field	UITextField

Figure 11-16: Connect the Text Field outlet to theTextField in your code.



Orientation: Portrait
Status Bar: Black

Figure 11-17: Change the view's background color attribute.



Name	Type
File's Owner	DeepThoughtsViewController
First Responder	UIResponder
View	UIView
Image View (05-30-04_1908.jpg)	UIImageView
Light Info Button	UIButton

Figure 11-18: Connect the Touch Up Inside event for the Light Info button.

Testing the new modal view

So it looks like you now have all the pieces in place for the DeepThoughts application. Save your Xcode project by choosing File➪Save. Then, to test the modal view, click the Build and Run button. You should see the Simulator launch, run the app, and display the falling words over the image.

Now click the Info button in the upper-left corner. The modal view should appear, as shown in Figure 11-19. Click inside the text box to make the keyboard appear, and type new words for the falling words. Drag the slider to change the animation speed, and the falling words should move slower or faster in the view behind the modal view. Cool!

In the Simulator, choose Hardware➪Rotate Right to see what DeepThoughts looks like in landscape orientation. After clicking inside the text box to edit the falling words, you should notice that in landscape orientation, the keyboard is much larger, and the modal view automatically slides up to accommodate it — thanks to the Form Sheet modal view style you chose in the "Initializing and setting the modal view style" section, earlier in this chapter.

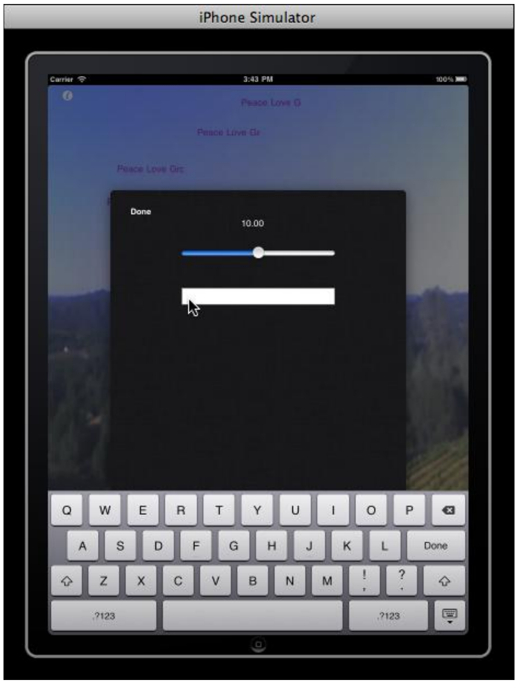


Figure 11-19: Run DeepThoughts and click the Info button to see the modal view.

As you experiment with code and build and run your project, you need to delete the application and its data from the Simulator if you change anything of significance — before building and running again. The consequences of not doing so will become obvious when things don't work like you would expect them to. See Chapter 5 for details on deleting specific apps from the Simulator. For a fast reset of all apps and all data in the simulator, choose iPhone Simulator➪Reset Contents and Settings, and then click Reset. Note that this removes *all* apps and data that you have installed in the simulator.

You can now enter a new phrase and speed up or slow down the animation. Your preferences are saved when you leave the app so that they're used when you launch the app again later.

Ah, but there's a bit more you can do with this little app, as you see in the next section.

Adding Tap and Swipe Recognizers

To put a finer point on touch events, you need to examine gestures. No sample iPad application would be complete without some gesture control, because with an iPad you have a large display that begs to be tapped, pinched, and even swiped.

To offer a finer grain of control over what happens when fingers touch the display, you can use a delegate of the `UIGestureRecognizer` class that recognizes gestures and customizes their actions. This class defines a set of common behaviors that can be configured for all gesture recognizers, and it can also communicate with its delegate (an object that adopts the `UIGestureRecognizerDelegate` protocol) for even finer control. With a Gesture-Recognizer object, you can separate the logic for recognizing a gesture from the action that should occur. When one of these objects recognizes a common gesture or, in some cases, a change in the gesture, it sends an action message to each designated target object.

To improve DeepThoughts, you can add a Tap recognizer to the view that brings up the modal view for changing settings (and thereby dispense with the Info button if you want). You can also create Swipe Gesture recognizers to recognize right and left swipes. You add this code to the view controller for the animated view, which means the files to be modified are `DeepThoughtsViewController.h` and `DeepThoughtsViewController.m`.

Add the bold lines of code in Listing 11-5 to the `DeepThoughtsViewController.h` file. First, add `UIGestureRecognizer` within the angle brackets of the `@interface` statement (be sure to include the comma before it) to declare the `UIGestureRecognizer` delegate. Then add a declaration for a new `addGestures` method, which you'll set up later in `DeepThoughtsController.m` (in Listing 11-7) to recognize gestures.

Listing 11-5: DeepThoughtsViewController.h

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

@interface DeepThoughtsViewController : UIViewController
           <SettingsViewControllerDelegate ,
           UIGestureRecognizerDelegate>  {

  UIImage     *fallingImage;
  NSString    *fallingWords;
  UIImageView *imageView;
  double       speed;
}
- (IBAction)settings;
- (void) changeSpeed: (double) newSpeed;
- (void) settingsViewController: (SettingsViewController
           *)controller didFinishWithChange: (BOOL)
           changed;

- (void) addGestures;

@property (readwrite)  double speed;
@property (nonatomic, retain) UIImageView *imageView;
@property (nonatomic, retain) NSString    *fallingWords;

@end
```

Next, add the bold line of code in Listing 11-6 to the `DeepThoughtsViewController.m` file at the end of the `viewDidLoad` method, which is at the end of the `View life cycle` section (the section you marked using `#pragma mark View life cycle` in Chapter 10).

Listing 11-6: DeepThoughtsViewController.m (View life cycle Section)

(The `View life cycle` section marked by `#pragma mark View life cycle` appears here. See Chapter 10 for the complete code.)

```
- (void)viewDidLoad {

  [super viewDidLoad];
  [NSTimer scheduledTimerWithTimeInterval:.5 target:self
          selector:@selector(onTimer) userInfo:nil
          repeats:YES];
  if (![[NSUserDefaults standardUserDefaults]
          objectForKey:kWordsOfWisdom]) {
    [[NSUserDefaults standardUserDefaults]
          setObject:@"Peace Love Groovy Music"
          forKey:kWordsOfWisdom];
    fallingWords = @"Peace Love Groovy Music";
  }
  else {
    fallingWords = [[NSUserDefaults standardUserDefaults]
          stringForKey:kWordsOfWisdom];
  }
  if (![[NSUserDefaults standardUserDefaults]
          objectForKey:kSpeed] ){
    [[NSUserDefaults standardUserDefaults]setDouble:10.0
          forKey:kSpeed];
    speed = kMaxSpeed-10.0;}
  else {
    speed = kMaxSpeed-[[NSUserDefaults
          standardUserDefaults] doubleForKey:kSpeed] ;
  }

  [self addGestures];
}
```

(The rest of the code would appear here. See Chapter 10 for the complete code.)

You have now added the `addGestures` method to the view, so it's time to specify what that method actually does.

Add the bold lines of code in Listing 11-7 to the `DeepThoughtsViewController.m` file — between the end of the `Controls` section (marked by `#pragma mark Controls` in Chapter 10) and the beginning of the Orientation section (marked by `#pragma mark Orientation` in Chapter 10).

Listing 11-7: DeepThoughtsViewController.m (View life cycle Section)

(The `Controls` section marked by `#pragma mark Controls` appears here. See Listing 11-4 for the Controls section, and see Chapter 10 for the complete code.)

```
    [self dismissModalViewControllerAnimated:YES];
}
```

(continued)

Listing 11-7 *(continued)*

```
#pragma mark -
#pragma mark Responding to gestures

- (void) addGestures {

  /*
   Create and configure the gesture recognizers. Add each
          to the view as a gesture recognizer.
   */
  UIGestureRecognizer *recognizer;
  /*
   Create a tap recognizer and add it to the view.
   Keep a reference to the recognizer to test in gestureRe
          cognizer:shouldReceiveTouch:.
   */
  recognizer = [[UITapGestureRecognizer
          alloc] initWithTarget:self action:@
          selector(handleTapFrom:)];
  [self.view addGestureRecognizer:recognizer];
  recognizer.delegate = self;
  [recognizer release];

  /*
   Create a swipe gesture recognizer to recognize right
          swipes (the default).
   */
  recognizer = [[UISwipeGestureRecognizer
          alloc] initWithTarget:self action:@
          selector(handleSwipeFrom:)];
  [self.view addGestureRecognizer:recognizer];
  [recognizer release];
  /*
   Create a swipe gesture recognizer to recognize left
          swipes.
   */
  recognizer = [[UISwipeGestureRecognizer
          alloc] initWithTarget:self action:@
          selector(handleSwipeFrom:)];
  ((UISwipeGestureRecognizer *)recognizer).direction =
          UISwipeGestureRecognizerDirectionLeft;
  [self.view addGestureRecognizer:recognizer];
  [recognizer release];
}
```

```
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)
           gestureRecognizer shouldReceiveTouch:(UITouch
           *)touch {
  return YES;
}

/*
 In response to a tap gesture, show settings modal view.
 */
- (void)handleTapFrom:(UITapGestureRecognizer *)recognizer
           {
  [self settings];
}

/*
 In response to a swipe gesture, change the speed.
 */
- (void)handleSwipeFrom:(UISwipeGestureRecognizer *)
           recognizer {

  if (recognizer.direction ==
           UISwipeGestureRecognizerDirectionLeft) {
    if (speed <= kMaxSpeed) speed = speed+2;
  }
  else {
    if (speed > 0) speed = speed-2;
  }
}

#pragma mark -
#pragma mark Orientation
```

(The `Orientation` section appears here. See Chapter 10 for the complete code.)

The code initializes a Tap recognizer and adds it to the view using the `addGestureRecognizer:` instance method that's part of `UIGestureRecognizer`. (`UITapGestureRecognizer` is a subclass of `UIGestureRecognizer` that looks for single or multiple taps.)

The code then initializes two Swipe recognizers to handle left and right swipes and adds them to the view. (`UISwipeGestureRecognizer` is a concrete subclass of `UIGestureRecognizer` that looks for swiping gestures in one or more directions.) The default direction is a right swipe, so you don't have to specify the direction. For a left swipe, you specify a `direction` property with `UISwipeGestureRecognizerDirectionLeft`.

Through this code, you also keep a reference to the `recognizer` so that you can test it using `gestureRecognizer:shouldReceiveTouch:`, which asks the delegate if a Gesture recognizer should receive an object representing a touch. `YES` (the default) lets the Gesture recognizer examine the Touch object.

Finally, you specify what the `handleTapFrom` method does: In response to a Tap gesture (from `UITapGestureRecognizer`), the method uses `settings` to display the modal view. You also specify what the `handleSwipeFrom` method does: If the direction is to the left (`recognizer.direction == UISwipeGestureRecognizerDirectionLeft`), the method reduces the animation speed; if not (which means the direction must be to the right, which is the default), the method increases the speed.

Save your Xcode project by choosing File➪Save. Then, to test the gestures, click the Build and Run button. You should see the Simulator launch, run the app, and display the falling words over the image.

Now mimic a tap by clicking anywhere in the animated view. The modal view should appear, and you can click inside the text box and type new words for the falling words, and you can drag the slider to change the animation speed. Click the Done button to return to the animated view. Now drag across the view to the right, and the animation should speed up — dragging is the equivalent of a swipe. Drag across to the left, and the animation should slow down. Very cool!

A Lot Accomplished Very Quickly

It looks like you have all the pieces in place for the DeepThoughts application. The user can now tap anywhere to bring up a modal view and enter a new phrase for the flowing words (so you can either keep or get rid of the Light Info button, which does the same thing) and control the animation speed. The user can also control the speed by swiping the animated view left or right.

Appearances can be deceiving, though.

Reality check: Some how-to books on software development should really be housed in the Fiction section of your local bookstore because all their examples work flawlessly. In the real world, everything doesn't always go as planned; occasionally your software program blows up on you. That's why an essential part of software development is the debugging phase — teasing as many flaws out of your app as possible so you can squash ’em. The next chapter shows you how to work through the debugging phase of your project and introduces you to the SDK’s very own debugging tool, something that’s sure to make your software-development life a lot easier.

Chapter 12
Getting the Bugs Out

In This Chapter

- ▶ Understanding the kinds of errors that may come up
- ▶ Using Xcode's Debugger and Build windows
- ▶ Setting breakpoints and examining objects and variables
- ▶ Using the Static Analyzer to analyze your code for memory leaks

When you're developing an application, sometimes things don't work out quite the way you planned — especially when you knock over a can of Jolt Cola on the keyboard and fry it out of existence.

Murphy was an optimist about computer programming with his law that there's always one more bug. It took Weinberg's Second Law to put debugging into perspective: If builders built buildings the way that programmers program programs, the first woodpecker to come along would destroy civilization.

As I learned the hard way (indeed, I wrote *Murphy's Computer Laws* in 1980 for Celestial Arts, only to violate most of them in subsequent projects), debugging is not something to put off until later, after the warnings and error messages pile up. Keep in mind Bove's Theorem: The remaining work required to finish a project increases as the deadline approaches. It's best to tackle any errors and warnings you get immediately.

So, what does it take to tackle the inevitable errors that will find their way into your code? In a word, *debugging:* the process of analyzing your code line by line to view your program's state at a particular stage of execution. To debug a program, you run it under the control of a debugger, which lets you pause the program and examine its state. In this chapter, I show you how to use the Xcode Debugger to understand, locate, and fix bugs.

Understanding Bugs

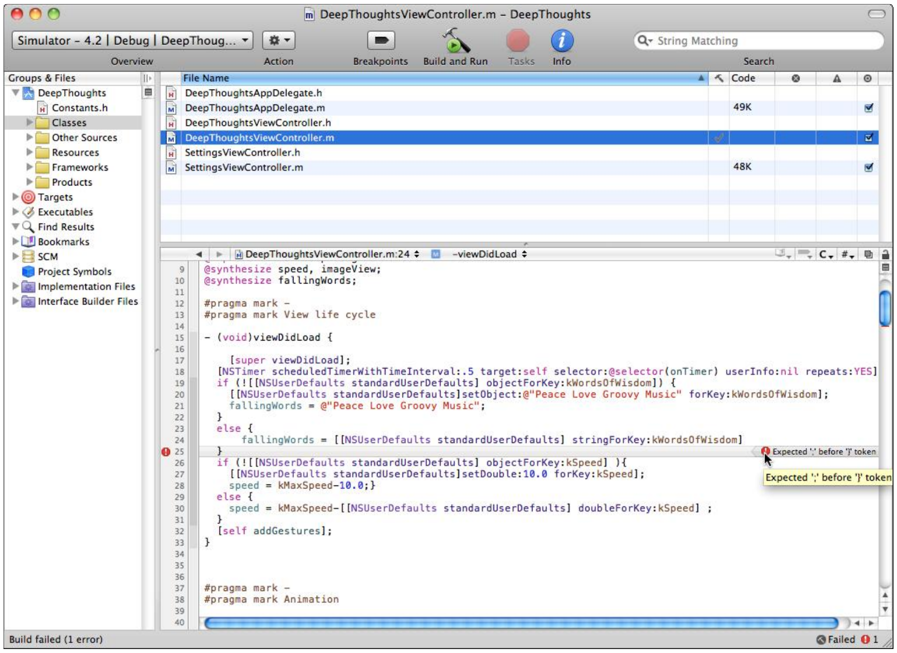
"Stuff happens," in the immortal words of a famous ex-U.S. Secretary of Defense. When it comes to developing your own programs, that "stuff" comes in three categories:

- **Syntax errors:** Compilers — the Objective-C compiler in Xcode is a case in point — expect you to use a certain set of instructions in your code; those instructions make up the language it understands. When you type `If` instead of `if`, or the subtler `[view release}` (with a curly closing bracket) instead of `[view release]` (with a straight closing bracket), the compiler suddenly has no idea what you're talking about and generates a syntax error. Objective-C is case sensitive, which means that `Speedchanged` and `speedChanged` are treated differently. Class, category, and protocol names generally begin with an uppercase letter (such as `NSUserDefaults`); the names of methods and instance variables typically begin with a lowercase letter (such as `speedChanged`).

  Syntax errors are the most obvious, simply because your program won't compile (and therefore won't run) until all of them are fixed. Generally, syntax errors spring from typographical errors. (And yes, the errors can be pretty penny-ante stuff — an *I* for an *i,* for goodness sake — it doesn't take much to stump a compiler.)

  In Figure 12-1, you can see an example of a syntax error — simply forgetting to put a semicolon at the end of the `fallingWords` statement. This one is kindly pointed out by Xcode's friendly Debugger feature. After choosing Build⇨Build and Run to build and run the application (and saving all changes in the process), the build fails — a tiny hammer icon and Failed appears in the notification section (in the bottom-right corner of the Xcode window), and the compiler highlights the statement after the syntax error with three (count 'em, three) red exclamation marks: one in the gutter to the left of the statement after the syntax error, one in the strip of debugging information that appears around the statement after the syntax error, and one next to Failed in the notifications section of the Xcode window.

  A syntax error in a different place in the code might not be explained so easily by Xcode and might therefore wreak havoc with subsequent code, thus causing the build to fail. For example, in Figure 12-2, I forget to include an asterisk in front of `recognizer` in the `UIGestureRecognizer recognizer` statement (which should be `UIGestureRecognizer *recognizer`). As a result, the compiler found a problem with subsequent code. Click the exclamation mark, and Xcode brings up the Build Results window, as shown in the upper part of Figure 12-3, with a long list of errors starting with `Statically allocated instance of Objective-C class 'UIGestureRecognizer'`. The Build Results window can show a more detailed view of the consequences of an error.



```
@synthesize speed, imageView;
@synthesize fallingWords;

#pragma mark -
#pragma mark View life cycle

- (void)viewDidLoad {

    [super viewDidLoad];
  [NSTimer scheduledTimerWithTimeInterval:.5 target:self selector:@selector(onTimer) userInfo:nil repeats:YES]
  if (![[NSUserDefaults standardUserDefaults] objectForKey:kWordsOfWisdom]) {
    [[NSUserDefaults standardUserDefaults]setObject:@"Peace Love Groovy Music" forKey:kWordsOfWisdom];
    fallingWords = @"Peace Love Groovy Music";
  }
  else {
      fallingWords = [[NSUserDefaults standardUserDefaults] stringForKey:kWordsOfWisdom]
  }
  if (![[NSUserDefaults standardUserDefaults] objectForKey:kSpeed] ){
    [[NSUserDefaults standardUserDefaults]setDouble:10.0 forKey:kSpeed];
    speed = kMaxSpeed-10.0;}
  else {
    speed = kMaxSpeed-[[NSUserDefaults standardUserDefaults] doubleForKey:kSpeed] ;
  }
  [self addGestures];
}



#pragma mark -
#pragma mark Animation

```

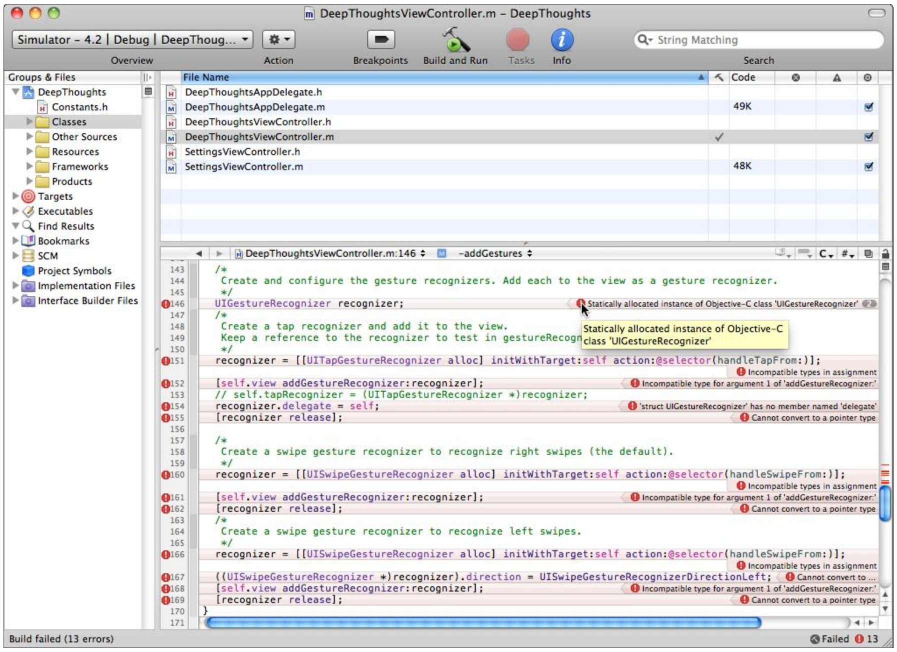
```
Expected ';' before '}' token
Expected ';' before '}' token
```

Figure 12-1: A syntax error. Oops.

TIP

It's generally better to ignore the subsequent errors after a syntax error because they may be the result of that first error.

- **Runtime errors:** *Runtime errors* cause your program to stop executing — it *crashes,* in other words, as in "crash and burn to much wailing and gnashing of teeth." Something might have come up in the data that you hadn't expected (a division-by-zero error, for example), or the result of a method dealt a nasty surprise to your logic, or you sent a message to an object that doesn't have that message implemented. Sometimes you even get some build warnings for these errors; often the application simply stops working or *hangs* (stops and does nothing), or shuts down.
- **Logic errors:** Your literal-minded application does exactly what you tell it to do, but sometimes you unintentionally tell it to do the wrong thing, and it coughs up a *logic error.* For example, in Figure 12-4, I deliberately created a logic error by dividing by zero. Xcode warns you about the divide-by-zero error but goes ahead anyway and builds and runs the app.



DeepThoughtsViewController.m – DeepThoughts

Simulator – 4.2 | Debug | DeepThoug… · Overview · Action · Breakpoints · Build and Run · Tasks · Info · Search

File Name	Code	
DeepThoughtsAppDelegate.h		
DeepThoughtsAppDelegate.m	49K	✓
DeepThoughtsViewController.h		
DeepThoughtsViewController.m		✓
SettingsViewController.h		
SettingsViewController.m	48K	✓

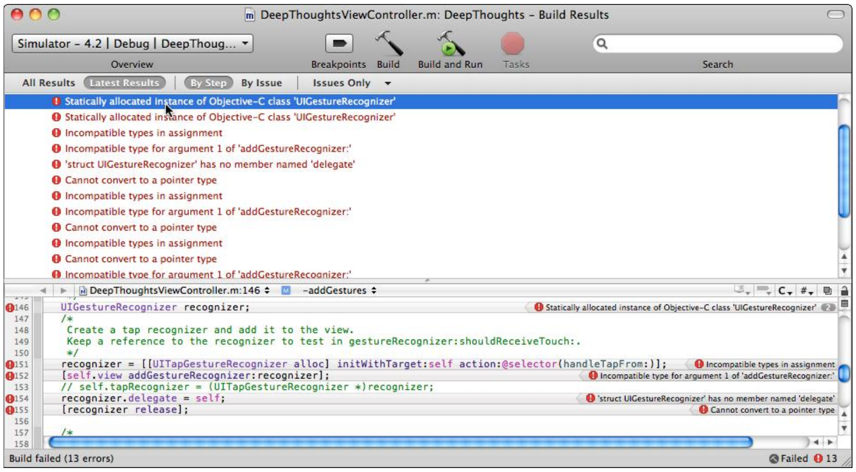
DeepThoughtsViewController.m:146 · -addGestures

```
/*
 Create and configure the gesture recognizers. Add each to the view as a gesture recognizer.
 */
UIGestureRecognizer recognizer;                 Statically allocated instance of Objective-C class 'UIGestureRecognizer'
/*
 Create a tap recognizer and add it to the view.
 Keep a reference to the recognizer to test in gestureRecogn
 */
recognizer = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(handleTapFrom:)];
                                                Incompatible types in assignment
[self.view addGestureRecognizer:recognizer];    Incompatible type for argument 1 of 'addGestureRecognizer:'
// self.tapRecognizer = (UITapGestureRecognizer *)recognizer;
recognizer.delegate = self;                     'struct UIGestureRecognizer' has no member named 'delegate'
[recognizer release];                           Cannot convert to a pointer type

/*
 Create a swipe gesture recognizer to recognize right swipes (the default).
 */
recognizer = [[UISwipeGestureRecognizer alloc] initWithTarget:self action:@selector(handleSwipeFrom:)];
                                                Incompatible types in assignment
[self.view addGestureRecognizer:recognizer];    Incompatible type for argument 1 of 'addGestureRecognizer:'
[recognizer release];                           Cannot convert to a pointer type
/*
 Create a swipe gesture recognizer to recognize left swipes.
 */
recognizer = [[UISwipeGestureRecognizer alloc] initWithTarget:self action:@selector(handleSwipeFrom:)];
                                                Incompatible types in assignment
((UISwipeGestureRecognizer *)recognizer).direction = UISwipeGestureRecognizerDirectionLeft;  Cannot convert to ...
[self.view addGestureRecognizer:recognizer];    Incompatible type for argument 1 of 'addGestureRecognizer:'
[recognizer release];                           Cannot convert to a pointer type
}
```

Statically allocated instance of Objective-C class 'UIGestureRecognizer'

Build failed (13 errors) · Failed 13

Figure 12-2: A subtle but more damaging syntax error.



DeepThoughtsViewController.m: DeepThoughts – Build Results

Simulator – 4.2 | Debug | DeepThoug… · Overview · Breakpoints · Build · Build and Run · Tasks · Search

All Results · Latest Results · By Step · By Issue · Issues Only

```
Statically allocated instance of Objective-C class 'UIGestureRecognizer'
Statically allocated instance of Objective-C class 'UIGestureRecognizer'
Incompatible types in assignment
Incompatible type for argument 1 of 'addGestureRecognizer:'
'struct UIGestureRecognizer' has no member named 'delegate'
Cannot convert to a pointer type
Incompatible types in assignment
Incompatible type for argument 1 of 'addGestureRecognizer:'
Cannot convert to a pointer type
Incompatible types in assignment
Cannot convert to a pointer type
Incompatible type for argument 1 of 'addGestureRecognizer:'
```

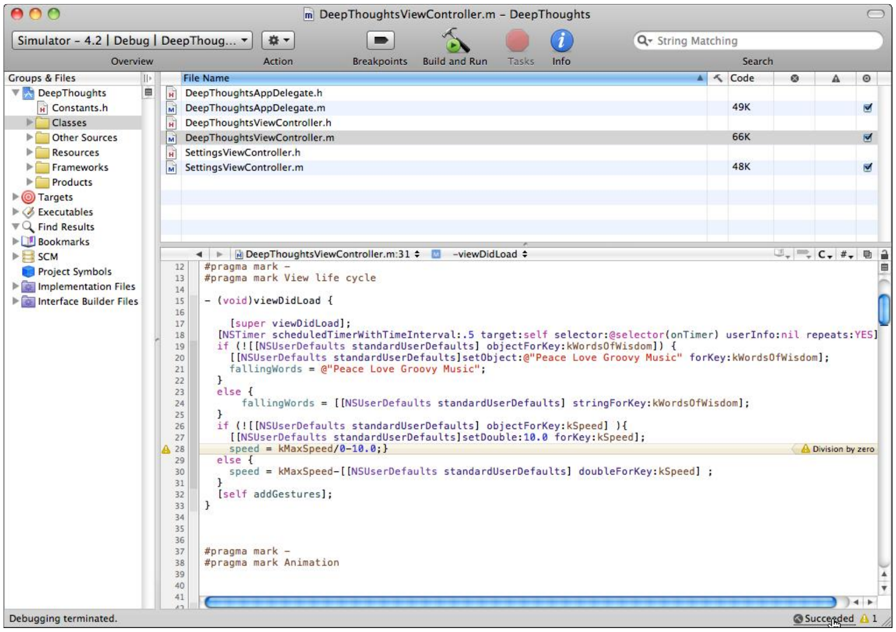
DeepThoughtsViewController.m:146 · -addGestures

```
UIGestureRecognizer recognizer;                 Statically allocated instance of Objective-C class 'UIGestureRecognizer'
/*
 Create a tap recognizer and add it to the view.
 Keep a reference to the recognizer to test in gestureRecognizer:shouldReceiveTouch:.
 */
recognizer = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(handleTapFrom:)];  Incompatible types in assignment
[self.view addGestureRecognizer:recognizer];    Incompatible type for argument 1 of 'addGestureRecognizer:'
// self.tapRecognizer = (UITapGestureRecognizer *)recognizer;
recognizer.delegate = self;                     'struct UIGestureRecognizer' has no member named 'delegate'
[recognizer release];                           Cannot convert to a pointer type

/*
```

Build failed (13 errors) · Failed 13

Figure 12-3: The Build Results window lists the damage from this one error.

I typed the divide-by-zero error (`speed = kMaxSpeed/0-10.0`) to make the point that you may be able to build and run your app, but it may not work as intended.



DeepThoughtsViewController.m – DeepThoughts

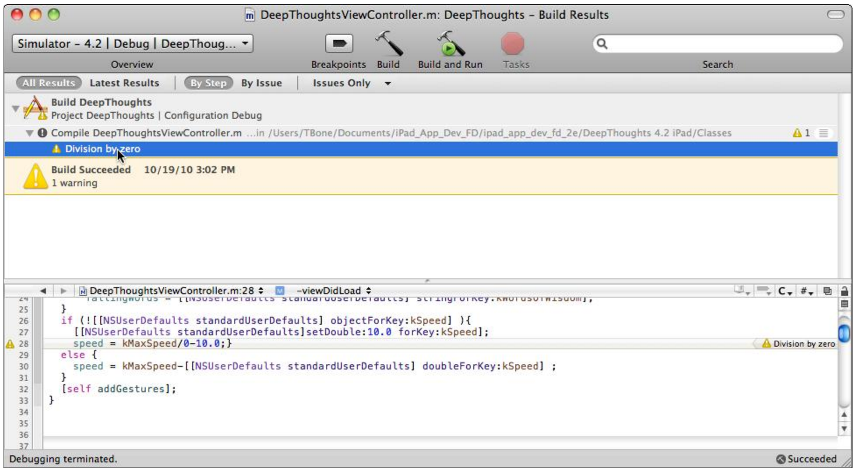
File Name	Code
DeepThoughtsAppDelegate.h	
DeepThoughtsAppDelegate.m	49K
DeepThoughtsViewController.h	
DeepThoughtsViewController.m	66K
SettingsViewController.h	
SettingsViewController.m	48K

```
#pragma mark -
#pragma mark View life cycle

- (void)viewDidLoad {

    [super viewDidLoad];
  [NSTimer scheduledTimerWithTimeInterval:.5 target:self selector:@selector(onTimer) userInfo:nil repeats:YES]
  if (![[NSUserDefaults standardUserDefaults] objectForKey:kWordsOfWisdom]) {
    [[NSUserDefaults standardUserDefaults]setObject:@"Peace Love Groovy Music" forKey:kWordsOfWisdom];
    fallingWords = @"Peace Love Groovy Music";
  }
  else {
      fallingWords = [[NSUserDefaults standardUserDefaults] stringForKey:kWordsOfWisdom];
  }
  if (![[NSUserDefaults standardUserDefaults] objectForKey:kSpeed] ){
    [[NSUserDefaults standardUserDefaults]setDouble:10.0 forKey:kSpeed];
    speed = kMaxSpeed/0-10.0;}                                                   Division by zero
  else {
    speed = kMaxSpeed-[[NSUserDefaults standardUserDefaults] doubleForKey:kSpeed] ;
  }
  [self addGestures];
}



#pragma mark -
#pragma mark Animation
```

Debugging terminated. Succeeded 1

Figure 12-4: Oh, great — it builds but doesn't work.

You can see in Figure 12-4 the yellow exclamation point, which is a warning (rather than a red one, which is an error), and the message `Division by zero`. Clicking the exclamation point brings up the Build Results window, as shown in Figure 12-5, with the `Division by zero` warning and the steps of compiling and building the app.



DeepThoughtsViewController.m: DeepThoughts – Build Results

```
Build DeepThoughts
Project DeepThoughts | Configuration Debug
  Compile DeepThoughtsViewController.m ...in /Users/TBone/Documents/iPad_App_Dev_FD/ipad_app_dev_fd_2e/DeepThoughts 4.2 iPad/Classes   1
    Division by zero
Build Succeeded   10/19/10 3:02 PM
1 warning
```

```
  }
  if (![[NSUserDefaults standardUserDefaults] objectForKey:kSpeed] ){
    [[NSUserDefaults standardUserDefaults]setDouble:10.0 forKey:kSpeed];
    speed = kMaxSpeed/0-10.0;}                                                   Division by zero
  else {
    speed = kMaxSpeed-[[NSUserDefaults standardUserDefaults] doubleForKey:kSpeed] ;
  }
  [self addGestures];
}
```

Debugging terminated. Succeeded

Figure 12-5: The Build Results window shows what happened.

TIP

With a complex app, you might be pelted with compiler warnings that you don't have time to take care of because they have no impact on the execution of the program. One reason to set your preferences so that Xcode always opens the Build Results window is that you'll be continually reminded about these warnings if you haven't fixed them. To set this preference, choose Xcode➪Preferences➪Building and choose Always from the Open During Builds pop-up menu.

Syntax errors, runtime errors, and logic errors can all be pains in the behind, but there's no need to think of them as insurmountable roadblocks. You're still on your way to a cool iPad app.

Using the Debugger

The Debugger can be really useful when your program isn't doing what you expect. For the blatant errors, the Debugger can show you exactly what was going on when the error occurred. It provides you with a trail of how you got to where you are, highlights the problem instruction, and shows you your application's variables and their values at that point.

TIP

If you've been following the examples in Chapters 9 through 11 for developing the DeepThoughts app, you're ready to debug the app, and your configuration should still be set to `Simulator-4.2 | Debug` in the pop-up menu in the upper-left corner of the Xcode Project window. (Refer to Figure 12-4.) If you've been developing a project with a different configuration, you must change it to the Debug build configuration. (Before you can take advantage of the Debugger, the compiler must collect information *for* the Debugger, and the Debug build configuration generates the debugging symbols for that purpose.)

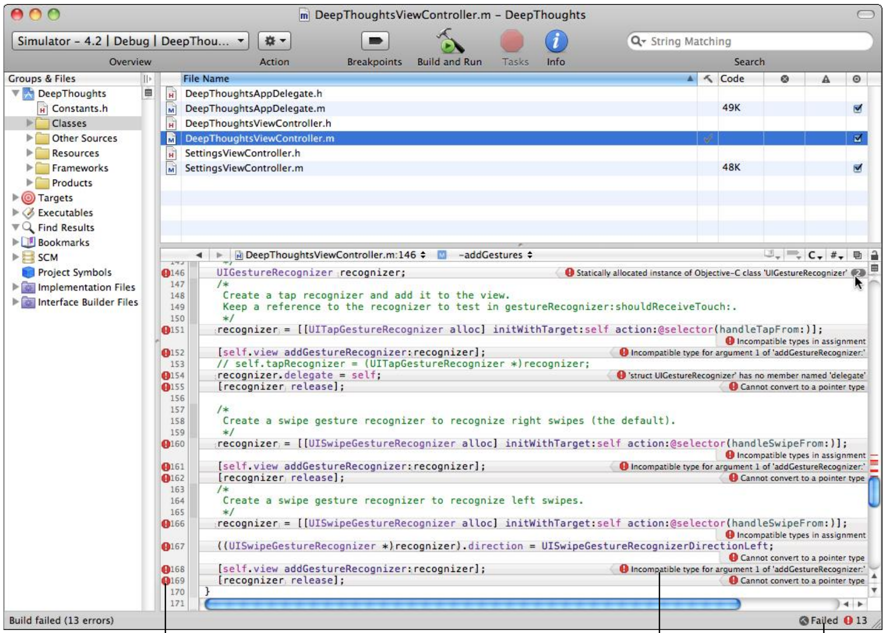
You can tap the Debugger from the Xcode Text Editor, as I show in the next section, and set breakpoints that stop execution at any point and trace the messages sent up to that point (as I describe in "Setting breakpoints" in the next section), so that you can step through the program's execution and view the contents of variables. The Debugger window offers even more control over the process and provides detailed information. You can also use the Mini Debugger — a floating window — that offers many of the functions of the Debugger window, as I show later in this chapter.

You can even use the Mac OS X Console utility application to view messages and interact with the GNU Source-Level Debugger with typed commands, as I explain in "Using the Console Application" in this chapter.

Debugging in the Text Editor

As shown earlier, a syntax error can stop a build in its tracks, and you can see both the compiler and the Debugger at work behind the scenes in the Text Editor. A red exclamation point in the Text Editor, as shown in Figure 12-6, points to the instruction that caused the program to stop building — that's the Debugger pointing out the problem.

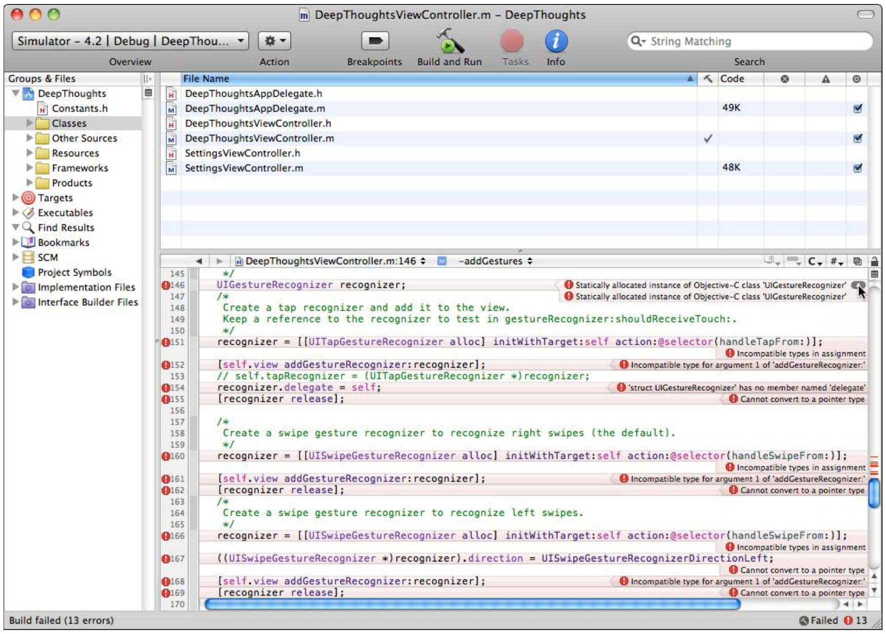
There's even some information about the error. The Debugger offers a strip of information called a *datatip,* which you can see in Figure 12-6 right next to the offending line. The datatip says `Statically allocated instance of Objective-C class 'UIGestureRecognizer'` and ends with a `2`, which means another error or warning is there. In Figure 12-7, I click the `2`, and it reveals a second warning, which happens to be the same error again. Other datatips show errors about incompatible types and that `'struct UIGestureRecognizer' has no member named 'delegate'`.



```
UIGestureRecognizer recognizer;        Statically allocated instance of Objective-C class 'UIGestureRecognizer' 2
/*
 Create a tap recognizer and add it to the view.
 Keep a reference to the recognizer to test in gestureRecognizer:shouldReceiveTouch:.
 */
recognizer = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(handleTapFrom:)];
                                                                  Incompatible types in assignment
[self.view addGestureRecognizer:recognizer];        Incompatible type for argument 1 of 'addGestureRecognizer:'
// self.tapRecognizer = (UITapGestureRecognizer *)recognizer;
recognizer.delegate = self;        'struct UIGestureRecognizer' has no member named 'delegate'
[recognizer release];                                             Cannot convert to a pointer type

/*
 Create a swipe gesture recognizer to recognize right swipes (the default).
 */
recognizer = [[UISwipeGestureRecognizer alloc] initWithTarget:self action:@selector(handleSwipeFrom:)];
                                                                  Incompatible types in assignment
[self.view addGestureRecognizer:recognizer];        Incompatible type for argument 1 of 'addGestureRecognizer:'
[recognizer release];                                             Cannot convert to a pointer type
/*
 Create a swipe gesture recognizer to recognize left swipes.
 */
recognizer = [[UISwipeGestureRecognizer alloc] initWithTarget:self action:@selector(handleSwipeFrom:)];
                                                                  Incompatible types in assignment
((UISwipeGestureRecognizer *)recognizer).direction = UISwipeGestureRecognizerDirectionLeft;
                                                                  Cannot convert to a pointer type
[self.view addGestureRecognizer:recognizer];        Incompatible type for argument 1 of 'addGestureRecognizer:'
[recognizer release];                                             Cannot convert to a pointer type
}
```

Figure 12-6: Xcode highlights an error and displays a datatip.

What this all means is that the compiler “thinks” I’m trying to allocate the object (`recognizer`) statically, rather than as a pointer — and that’s because I forgot to include an asterisk in front of `recognizer`.



```
  */
UIGestureRecognizer recognizer;                         Statically allocated instance of Objective-C class 'UIGestureRecognizer'
                                                        Statically allocated instance of Objective-C class 'UIGestureRecognizer'
/*
 Create a tap recognizer and add it to the view.
 Keep a reference to the recognizer to test in gestureRecognizer:shouldReceiveTouch:.
 */
recognizer = [[UITapGestureRecognizer alloc] initWithTarget:self action:@selector(handleTapFrom:)];
                                                        Incompatible types in assignment
[self.view addGestureRecognizer:recognizer];            Incompatible type for argument 1 of 'addGestureRecognizer:'
// self.tapRecognizer = (UITapGestureRecognizer *)recognizer;
recognizer.delegate = self;                             'struct UIGestureRecognizer' has no member named 'delegate'
[recognizer release];                                   Cannot convert to a pointer type

/*
 Create a swipe gesture recognizer to recognize right swipes (the default).
 */
recognizer = [[UISwipeGestureRecognizer alloc] initWithTarget:self action:@selector(handleSwipeFrom:)];
                                                        Incompatible types in assignment
[self.view addGestureRecognizer:recognizer];            Incompatible type for argument 1 of 'addGestureRecognizer:'
[recognizer release];                                   Cannot convert to a pointer type
/*
 Create a swipe gesture recognizer to recognize left swipes.
 */
recognizer = [[UISwipeGestureRecognizer alloc] initWithTarget:self action:@selector(handleSwipeFrom:)];
                                                        Incompatible types in assignment
((UISwipeGestureRecognizer *)recognizer).direction = UISwipeGestureRecognizerDirectionLeft;
                                                        Cannot convert to a pointer type
[self.view addGestureRecognizer:recognizer];            Incompatible type for argument 1 of 'addGestureRecognizer:'
[recognizer release];                                   Cannot convert to a pointer type
```

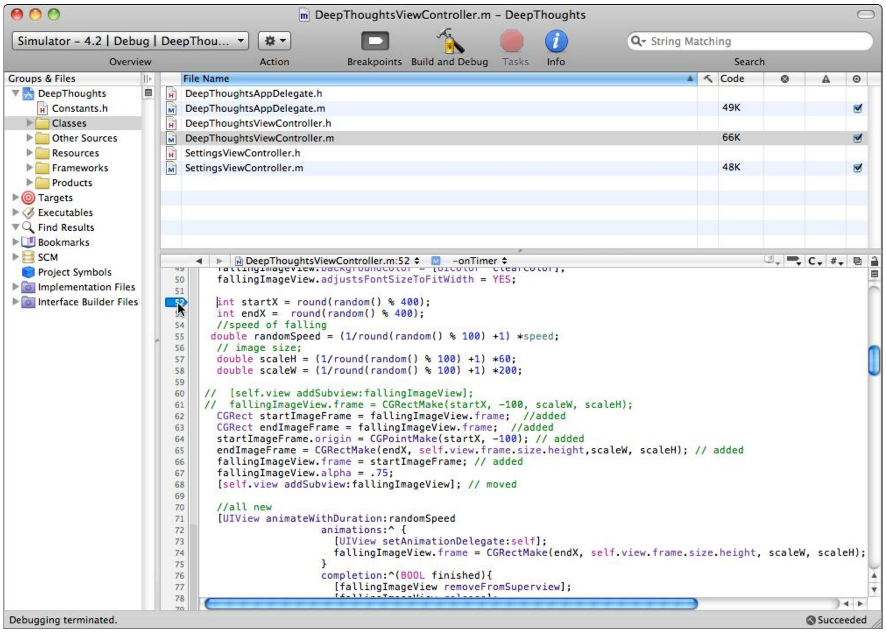
Figure 12-7: The datatip shows a second warning.

If your app manages to build and run (which can happen even with a warning, as you can see back in Figure 12-4), that means it has passed through the compiler without syntax errors. But you aren’t out of the woods yet — even if you don’t see evidence of runtime errors that crash the app, you certainly haven’t tried all the app’s functions yet. You also don’t know whether there are logic errors. But don’t despair; you have options.

Setting breakpoints

Breaking down may be a bad situation in real life, but in the life of your app, getting a break is a good thing. A *breakpoint* is an instruction to the Debugger to stop execution at that instruction and wait for further instructions (no pun intended). By setting breakpoints at various methods in your program, you can step through its execution — at the instruction level — to see exactly what it’s doing. You can also examine the variables the program is setting and using. If you’re stymied by a logic error, setting breakpoints is a great way to break that logjam.

To set a breakpoint in the Xcode Text Editor, click inside the *gutter* — the far-left column of the Editor pane, as shown in Figure 12-8. I set a breakpoint to stop execution right before executing the `int startX = round(random() % 400)` statement.



File Name	Code
DeepThoughtsAppDelegate.h	
DeepThoughtsAppDelegate.m	49K
DeepThoughtsViewController.h	
DeepThoughtsViewController.m	66K
SettingsViewController.h	
SettingsViewController.m	48K

```
  fallingImageView.adjustsFontSizeToFitWidth = YES;

  int startX = round(random() % 400);
  int endX =  round(random() % 400);
  //speed of falling
 double randomSpeed = (1/round(random() % 100) +1) *speed;
  // image size;
  double scaleH = (1/round(random() % 100) +1) *60;
  double scaleW = (1/round(random() % 100) +1) *200;

//  [self.view addSubview:fallingImageView];
//  fallingImageView.frame = CGRectMake(startX, -100, scaleW, scaleH);
  CGRect startImageFrame = fallingImageView.frame;  //added
  CGRect endImageFrame = fallingImageView.frame;  //added
  startImageFrame.origin = CGPointMake(startX, -100); // added
  endImageFrame = CGRectMake(endX, self.view.frame.size.height,scaleW, scaleH); // added
  fallingImageView.frame = startImageFrame; // added
  fallingImageView.alpha = .75;
  [self.view addSubview:fallingImageView]; // moved

  //all new
  [UIView animateWithDuration:randomSpeed
                 animations:^ {
                   [UIView setAnimationDelegate:self];
                   fallingImageView.frame = CGRectMake(endX, self.view.frame.size.height, scaleW, scaleH);
                 }
                 completion:^(BOOL finished){
                   [fallingImageView removeFromSuperview];
```

Figure 12-8: Setting a breakpoint in the Text Editor.

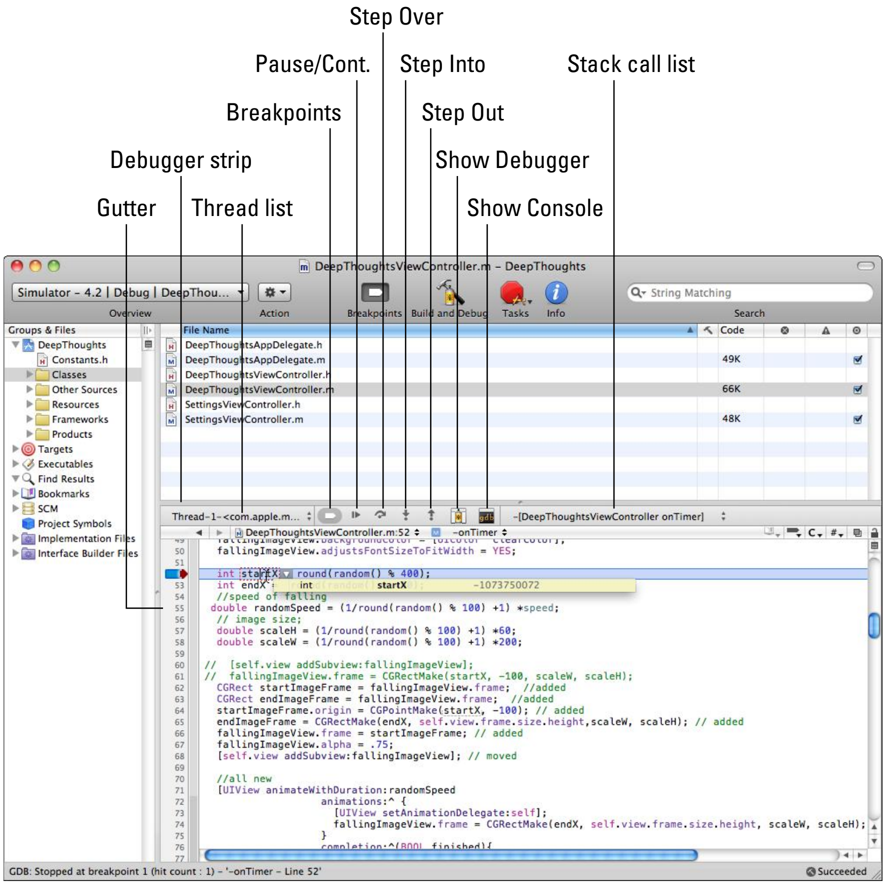
TIP

To get rid of a breakpoint, simply drag it off to the side. You can also right-click (or Control-click) the breakpoint and choose Remove Breakpoint from the pop-up menu that appears.

You can set the Xcode Text Editor to recognize breakpoints by clicking the Breakpoints button in the Project window toolbar — the Build and Run button changes to Build and Debug. Click Build and Debug to build and run the program.

Using the Debugger strip

When you build and run the program with breakpoints, the Debugger strip appears in the Text Editor as the program runs in the Simulator. The program stops executing at the first breakpoint. The process counter (PC) red arrow points to the line of code in the Text Editor immediately following the breakpoint. The Debugger strip appears just above the Text Editor, as shown in Figure 12-9, while the app is running in the Simulator but stopped at the breakpoint.



```
    fallingImageView.adjustsFontSizeToFitWidth = YES;

    int startX = round(random() % 400);
    int endX =
    //speed of falling
   double randomSpeed = (1/round(random() % 100) +1) *speed;
    // image size;
    double scaleH = (1/round(random() % 100) +1) *60;
    double scaleW = (1/round(random() % 100) +1) *200;

//  [self.view addSubview:fallingImageView];
//  fallingImageView.frame = CGRectMake(startX, -100, scaleW, scaleH);
    CGRect startImageFrame = fallingImageView.frame;  //added
    CGRect endImageFrame = fallingImageView.frame;  //added
    startImageFrame.origin = CGPointMake(startX, -100); // added
    endImageFrame = CGRectMake(endX, self.view.frame.size.height,scaleW, scaleH); // added
    fallingImageView.frame = startImageFrame; // added
    fallingImageView.alpha = .75;
    [self.view addSubview:fallingImageView]; // moved

    //all new
    [UIView animateWithDuration:randomSpeed
                     animations:^ {
                       [UIView setAnimationDelegate:self];
                       fallingImageView.frame = CGRectMake(endX, self.view.frame.size.height, scaleW, scaleH);
                     }
                     completion:^(BOOL finished){
```

```
GDB: Stopped at breakpoint 1 (hit count : 1) - '-onTimer - Line 52'
```

Figure 12-9: Xcode displays the Debugger strip as the app runs in the Simulator.

TIP When you move your pointer over a variable in a datatip (refer to Figure 12-9), its contents are revealed. You can even modify the contents of mutable variables. This is a powerful way to find out the value of variables at any given point during execution. (And yes, a slip of the datatip can sink a shipping app.)

The Debugger strip offers several buttons for your pushing pleasure:

- **Thread list:** Displays a list of the threads in your program. I explain this in “Using the Debugger Window,” later in this chapter.
- **Breakpoints:** Activates or deactivates breakpoints, which I describe in the preceding section, “Setting breakpoints.”
- **Continue:** Continues execution of a paused process in your program.

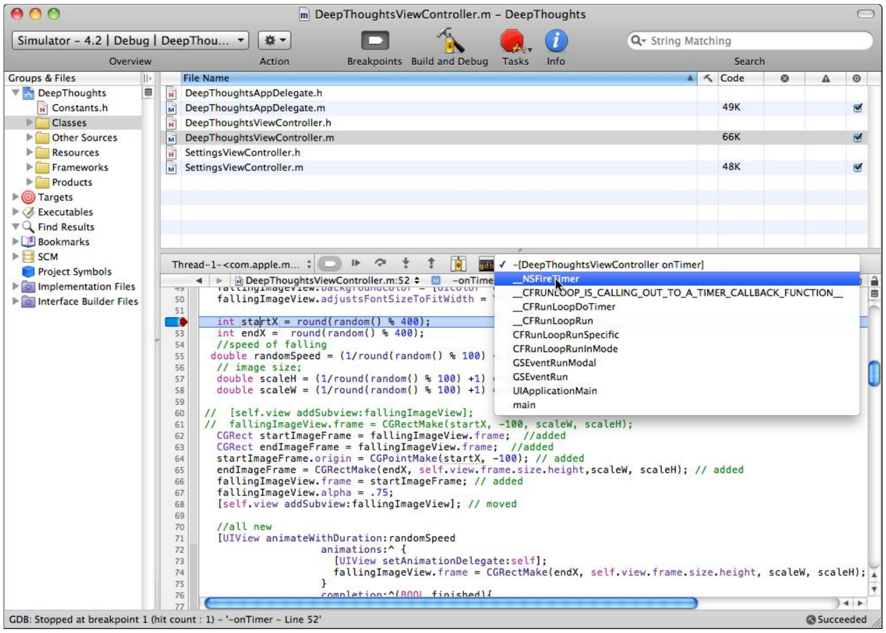
- **Step Over:** Steps over the current line of code. The *process counter* (PC), which is identified by the red arrow in the gutter (refer to Figure 12-9), moves to the next line of code to be executed in the current file.
- **Step Into:** Steps into a function or method in the current line of code. If possible, the Text Editor shows the source file with the called routine. The PC (red arrow) points to the line of code to be executed next.
- **Step Out:** Steps out of the current function or method. The Text Editor shows the source file with the function's caller.
- **Show Debugger:** Opens the Debugger proper.
- **Show Console:** Opens the Mac OS X Console, which I describe in "Using the Console Application," later in this chapter.
- **Call list:** Displays a list of the called functions or methods in the *stack,* which I explain next.

Click the up and down arrows next to `DeepThoughtsViewController onTimer` in the Debugger strip (refer to Figure 12-9), or whatever else is displayed in that section of the Debugger strip, so that you can see the *stack* — a trace of the objects and methods that got you to where you are now, as shown in Figure 12-10.

Although the stack is about as useful as a stack of pancakes in this particular context, the stack *can* be very useful in a more complex application — it can help you understand the path that you took to get where you are. Seeing how one object sent a message to another object — which sent a message to a third object — can be really helpful, especially if you didn't expect the program flow to work that way.

Getting a look at the stack can also be useful if you're trying to understand how the framework does its job, and in what order messages are sent. You can stop the execution of your program at a breakpoint and trace the messages sent up to that point.

You can play with your app in the Simulator and then switch back to the Text Editor to launch the Debugger window — click the Show Debugger button in the Debugger strip (refer to Figure 12-9), or choose Run➪Debugger, to bring up the Debugger window.



```
-[DeepThoughtsViewController onTimer]
__NSFireTimer
__CFRUNLOOP_IS_CALLING_OUT_TO_A_TIMER_CALLBACK_FUNCTION__
__CFRunLoopDoTimer
__CFRunLoopRun
CFRunLoopRunSpecific
CFRunLoopRunInMode
GSEventRunModal
GSEventRun
UIApplicationMain
main
```

```
  fallingImageView.adjustsFontSizeToFitWidth = 

  int startX = round(random() % 400);
  int endX =  round(random() % 400);
  //speed of falling
 double randomSpeed = (1/round(random() % 100) 
  // image size;
  double scaleH = (1/round(random() % 100) +1) 
  double scaleW = (1/round(random() % 100) +1) 

//  [self.view addSubview:fallingImageView];
//  fallingImageView.frame = CGRectMake(startX, -100, scaleW, scaleH);
  CGRect startImageFrame = fallingImageView.frame;  //added
  CGRect endImageFrame = fallingImageView.frame;  //added
  startImageFrame.origin = CGPointMake(startX, -100); // added
  endImageFrame = CGRectMake(endX, self.view.frame.size.height,scaleW, scaleH); // added
  fallingImageView.frame = startImageFrame; // added
  fallingImageView.alpha = .75;
  [self.view addSubview:fallingImageView]; // moved

  //all new
  [UIView animateWithDuration:randomSpeed
                   animations:^ {
                     [UIView setAnimationDelegate:self];
                     fallingImageView.frame = CGRectMake(endX, self.view.frame.size.height, scaleW, scaleH);
                   }
```

```
GDB: Stopped at breakpoint 1 (hit count : 1) - '-onTimer - Line 52'
```

Figure 12-10: The stack in the Debugger strip in the Text Editor.

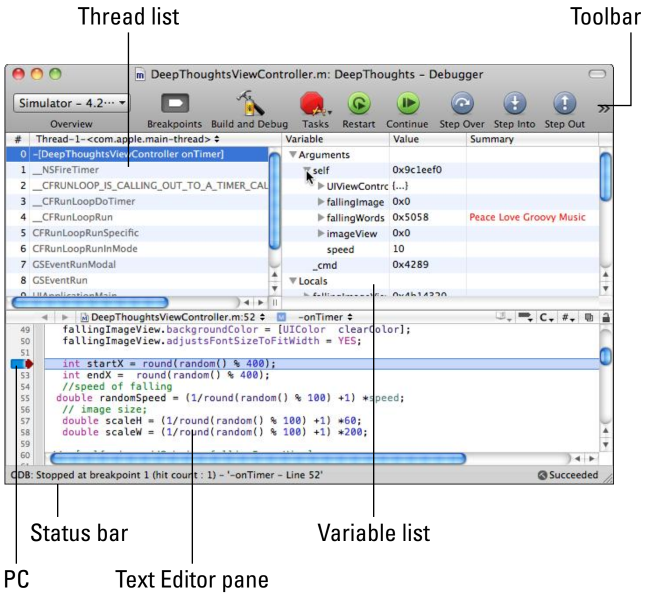
Using the Debugger Window

After clicking the Show Debugger button in the Debugger strip, or choosing Run➪Debugger (or pressing ⌘+Shift+Y), the Debugger window appears. (Even though the Debugger is officially running, you have to open the Debugger window explicitly.) You can then click the Pause button along the top of the Debugger window to stop execution, unless execution is already stopped at a breakpoint. (The Restart button replaces the Pause button after clicking Pause or stopping at a breakpoint, as shown in Figure 12-11.)

The Debugger window has everything the Text Editor has, but you can also see your stack and the variables in scope at a glance.

Here's what you see in the Debugger window:

- **Toolbar:** Offers buttons for controlling the program's execution, including Pause/Restart, Continue, Step Over, Step Into, and Step Out. (Restart starts execution from the beginning, whereas Continue continues execution from a breakpoint.)



Thread-1-<com.apple.main-thread>

#	
0	-[DeepThoughtsViewController onTimer]
1	__NSFireTimer
2	__CFRUNLOOP_IS_CALLING_OUT_TO_A_TIMER_CAL
3	__CFRunLoopDoTimer
4	__CFRunLoopRun
5	CFRunLoopRunSpecific
6	CFRunLoopRunInMode
7	GSEventRunModal
8	GSEventRun

Variable	Value	Summary
Arguments		
self	0x9c1eef0	
UIViewContrc	{...}	
fallingImage	0x0	
fallingWords	0x5058	Peace Love Groovy Music
imageView	0x0	
speed	10	
_cmd	0x4289	
Locals		

DeepThoughtsViewController.m:52 -onTimer

```
  fallingImageView.backgroundColor = [UIColor  clearColor];
  fallingImageView.adjustsFontSizeToFitWidth = YES;

  int startX = round(random() % 400);
  int endX =  round(random() % 400);
  //speed of falling
 double randomSpeed = (1/round(random() % 100) +1) *speed;
  // image size;
  double scaleH = (1/round(random() % 100) +1) *60;
  double scaleW = (1/round(random() % 100) +1) *200;
```

GDB: Stopped at breakpoint 1 (hit count : 1) – '-onTimer – Line 52'   Succeeded

Figure 12-11: The Debugger window.

- **Thread list:** Shows the call stack of the current thread. For each function or method call that your program makes, the Debugger stores information about it in a stack frame. These stack frames are stored in the call stack. When you pause execution at a breakpoint or when you click the Pause button on the toolbar, Xcode displays the call stack for the currently running process in the Thread list and puts the most recent call at the top. The pop-up menu above this view lets you select different threads to view when debugging a multi-threaded application.
- **Variable list:** Shows information — such as name, type, and value — about the variables for the selected stack frame. To see the contents of a structured variable (including arrays and vectors) or an object, click the triangle next to the variable.
- **Text Editor pane:** Displays the source code you're debugging. When you pause execution by clicking the Pause button in the Toolbar, the Debugger highlights the line of source code where execution paused and displays the PC red arrow indicator.
- **Status bar:** Displays the current status of the debugging session. For example, in Figure 12-11, Xcode indicates that GDB (the GNU Source-Level Debugger) is stopped at breakpoint 1.

Your window may not look exactly like Figure 12-11 — that's because Xcode gives you lots of different ways to customize the look of the Debugger window. You can, for example, choose Run➪Debugger Display from the main menu and then choose Horizontal Layout or Vertical Layout to change the window's layout.

TIP

You might want to choose Run➪Debugger Display➪Source and Disassembly if you have a hankering for checking both the source code *and* the assembly language (if you really care about assembly language); in that case, the Text Editor pane divides down the center into two panes, with the source code on the left and the assembly code on the right. The option I chose for Figure 12-11 is Source Only — so that only the source code appears in the Text Editor pane.

You can click the Step Into button in the Debugger window to go through your code line by line. The Debugger window also gives you other options for making your way through your program:

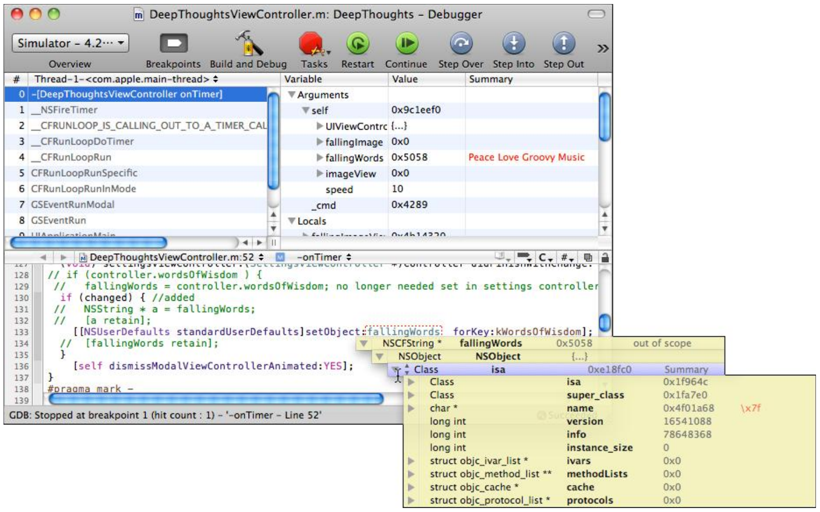
- **Step Over** gives you the opportunity to skip over a line of code.
- **Step Into** takes you step by step into a function or method in the current line of code.
- **Step Out** takes you out of the current method.
- **Continue** tells the program to keep on with its execution.
- **Restart** restarts the program. (You were hoping maybe if you tried it again it would work?)

Showing datatips for variables and objects

In the Debugger window, as shown in Figure 12-12, you can move your pointer over an object or variable in the Text Editor pane to show its contents in a datatip, and you can move your pointer over other disclosure triangles to see even more information in the datatip. In Figure 12-12, I move the pointer over `fallingWords` to show that it's an `NSObject`, and then I move the pointer over the triangle to reveal its class (`NSObject`) information.

TIP

Expanding the view of objects not only helps you check variables, but also checks messages sent to object reference instance variables. Objective-C, unlike some other languages, allows you to send a message to a `nil` object *without* generating a runtime error. If you do that, you should expect to subsequently see some sort of logic error because a message to a `nil` object simply does nothing. But it's possible that an object reference hasn't been set, and you're sending the message into the ether. If you look at an object reference instance variable and its value is `0x0`, any messages to that object are simply ignored. So when you get a logic error, the first thing you may want to check is whether any of the object references you’re using have `0x0` as their values, informing you that the reference was never initialized.



```
#  Thread-1-<com.apple.main-thread>
0  -[DeepThoughtsViewController onTimer]
1  __NSFireTimer
2  __CFRUNLOOP_IS_CALLING_OUT_TO_A_TIMER_CAL
3  __CFRunLoopDoTimer
4  __CFRunLoopRun
5  CFRunLoopRunSpecific
6  CFRunLoopRunInMode
7  GSEventRunModal
8  GSEventRun
```

```
Variable            Value        Summary
▼ Arguments
  ▼ self            0x9c1eef0
    ▶ UIViewContrc  {...}
    ▶ fallingImage  0x0
    ▶ fallingWords  0x5058       Peace Love Groovy Music
    ▶ imageView     0x0
      speed         10
    _cmd            0x4289
▼ Locals
```

```
DeepThoughtsViewController.m:52   -onTimer
// if (controller.wordsOfWisdom ) {
 //   fallingWords = controller.wordsOfWisdom; no longer needed set in settings controller
  if (changed) { //added
 //   NSString * a = fallingWords;
 //   [a retain];
    [[NSUserDefaults standardUserDefaults]setObject:fallingWords  forKey:kWordsOfWisdom];
  //  [fallingWords retain];
  }
    [self dismissModalViewControllerAnimated:YES];
}
#pragma mark -
```

```
NSCFString *   fallingWords   0x5058      out of scope
  NSObject     NSObject       {...}
    Class      isa            0xe18fc0    Summary
      Class                      isa            0x1f964c
      Class                      super_class    0x1fa7e0
      char *                     name           0x4f01a68   \x7f
      long int                   version        16541088
      long int                   info           78648368
      long int                   instance_size  0
      struct objc_ivar_list *    ivars          0x0
      struct objc_method_list ** methodLists    0x0
      struct objc_cache *        cache          0x0
      struct objc_protocol_list * protocols     0x0
```

```
GDB: Stopped at breakpoint 1 (hit count : 1) - '-onTimer - Line 52'
```

Figure 12-12: Show the contents of an object or variable.

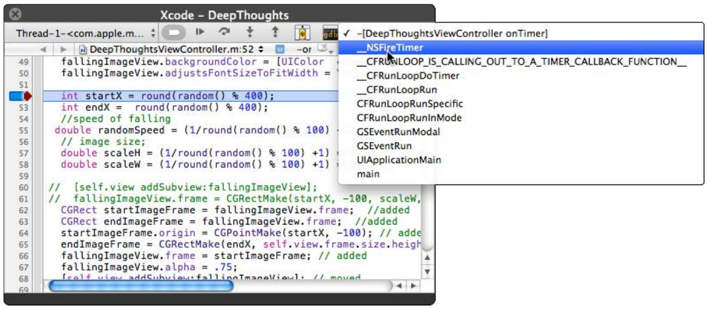
As you can see, the Debugger can be really useful when your program isn’t doing what you expect. For the blatant errors, the Debugger can show you exactly what was going on when the error occurred. It provides you with a trail of how you got to where you are, highlights the problem instruction, and shows you your application’s variables and their values at that point.

What’s just as valuable is how the Debugger can help you with logic errors. You may have mistakenly attached the slider interface object in Interface Builder to `theTextField` rather than `slider`. Sending a message to `nil` is not uncommon, especially when you’re making changes to the user interface and forget to set up an outlet, for example. In such situations, the ability to look at the object references can really help.

Using the Mini Debugger

The Mini Debugger is a floating window that provides debugging controls similar to those of the Xcode Text Editor. It can make debugging a bit easier, because you don’t have to switch back and forth between your running application and your Xcode Project window and Debugging window.

To show the Mini Debugger while running your program, choose Run➪Mini Debugger. The Mini Debugger appears as shown in Figure 12-13, with buttons to stop or pause the program, open the Xcode project, or activate or deactivate breakpoints.



```
  fallingImageView.backgroundColor = [UIColor
  fallingImageView.adjustsFontSizeToFitWidth =

  int startX = round(random() % 400);
  int endX =  round(random() % 400);
  //speed of falling
 double randomSpeed = (1/round(random() % 100)
  // image size;
  double scaleH = (1/round(random() % 100) +1)
  double scaleW = (1/round(random() % 100) +1)

//  [self.view addSubview:fallingImageView];
//  fallingImageView.frame = CGRectMake(startX, -100, scaleW,
  CGRect startImageFrame = fallingImageView.frame;  //added
  CGRect endImageFrame = fallingImageView.frame;  //added
  startImageFrame.origin = CGPointMake(startX, -100); // adde
  endImageFrame = CGRectMake(endX, self.view.frame.size.heigh
  fallingImageView.frame = startImageFrame; // added
  fallingImageView.alpha = .75;
```

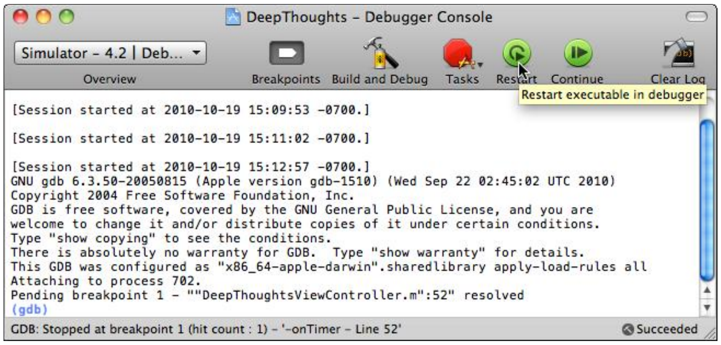
Figure 12-13: Use the Mini Debugger to pause program execution and check out the code.

After pausing or stopping the program (or reaching a breakpoint), the Mini Debugger displays the same information you would see when debugging in the Text Editor. As you can see in Figure 12-13, you can click the rightmost pop-up menu along the top of the window to see the call stack.

Using the Console Application

The Console utility application, supplied with Mac OS X, lets you watch error and status messages as they appear. If your computer appears to be stalled or is acting in an unusual manner, Console might be producing information that can help debug the problem. While the Xcode Debugger provides a graphical interface for GDB (the GNU Source-Level Debugger), Console lets you interact directly with GDB using a command line. You can type commands using Console to perform simple debugging tasks, and you can include code in your app to use `NSLog` statements to log messages to Console before and after variables are set.

To open the Console window, choose Run➪Console. After building and running the Xcode project, the messages appear in the Console window, as you can see in Figure 12-14.



```
[Session started at 2010-10-19 15:09:53 -0700.]

[Session started at 2010-10-19 15:11:02 -0700.]

[Session started at 2010-10-19 15:12:57 -0700.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1510) (Wed Sep 22 02:45:02 UTC 2010)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".sharedlibrary apply-load-rules all
Attaching to process 702.
Pending breakpoint 1 - ""DeepThoughtsViewController.m":52" resolved
(gdb)
```

GDB: Stopped at breakpoint 1 (hit count : 1) – '-onTimer – Line 52'

Figure 12-14: Use the Console window to monitor error and status messages.

You can use the Console window to see the commands that Xcode sends to GDB or the Java command-line debugger, to actually send commands directly to GDB or the Java command-line debugger, and to look at the debugger output for those commands. To enter commands, click in the Console window and type at the `gdb` or `JavaBug` prompt. To get help with GDB and Java debugging commands, type **help**. (To get the `gdb` or `JavaBug` prompt, the program you're debugging must be paused.)

Using the Static Analyzer

Xcode offers the Build and Analyze feature (the Static Analyzer) that analyzes your code for memory leaks. (Memory leaks are situations in which parts of memory become unusable or hidden, or the app is unable to release memory it has acquired. Memory leaks can cause apps to fail.) The results show up like warnings and errors, with explanations of where and what the issue is. You can also see the flow of control of the (potential) problem.

To show how this works, I deliberately created a memory leak in `DeepThoughtsAppDelegate`. I copied the following line of code in `DeepThoughtsAppDelegate.h`:

```
DeepThoughtsViewController *viewController;
```

I then added the following statement below the statement you see above, changing `viewController` to `viewController2`:

```
DeepThoughtsViewController *viewController2;
```

And a few lines down, I did the same thing — I copied the `@property` statement for `viewController` to make one for `viewController` 2:

```
@property (nonatomic, retain) IBOutlet
          DeepThoughtsViewController *viewController2;
```

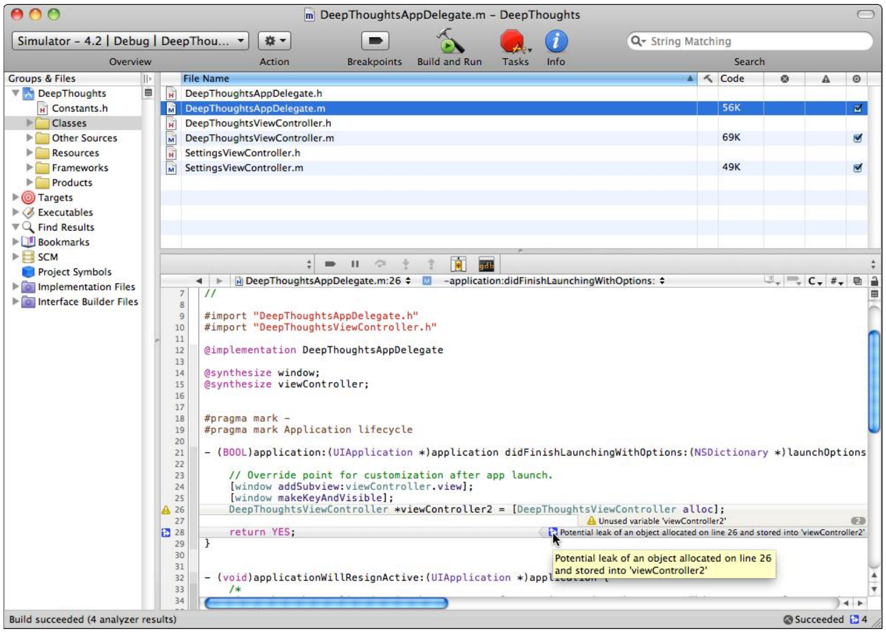
Then, in `DeepThoughtsAppDelegate.m`, I added the following line of code after setting up the view with the view controller:

```
DeepThoughtsViewController *viewController2 =
          [DeepThoughtsViewController alloc];
```

Allocating a new object without doing anything with it is sure to cause a memory leak warning.

To run the Static Analyzer, choose Build➪Build and Analyze. Sure enough, the changes I made to the code cause the warning shown in Figure 12-15. I get a warning (ignore the unused variable warning) with a little blue icon that says

```
Potential leak of an object allocated on line 23 and
          stored into 'viewController2'
```



File Name	Code	✓
DeepThoughtsAppDelegate.h		
DeepThoughtsAppDelegate.m	56K	✓
DeepThoughtsViewController.h		
DeepThoughtsViewController.m	69K	✓
SettingsViewController.h		
SettingsViewController.m	49K	✓

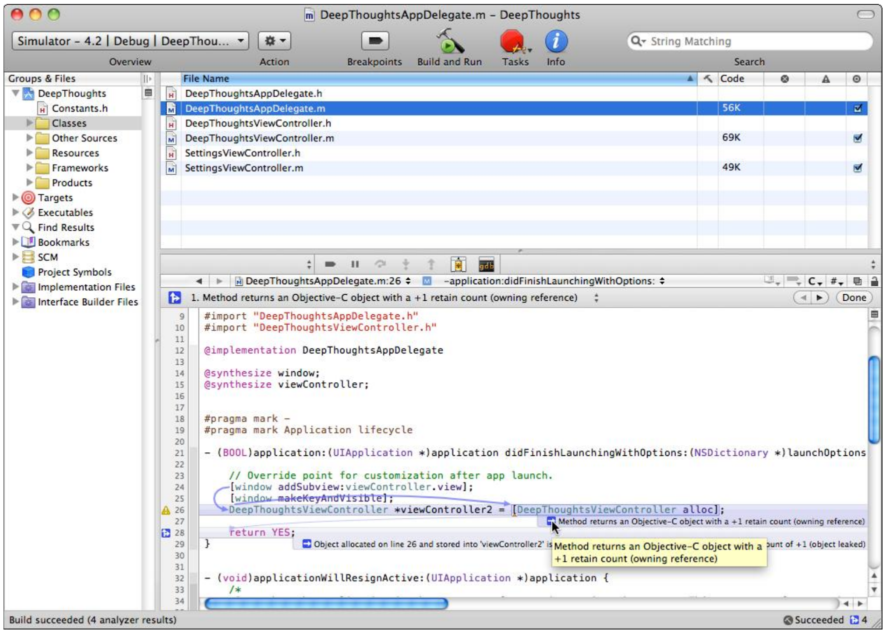
```
 7  //
 8
 9  #import "DeepThoughtsAppDelegate.h"
10  #import "DeepThoughtsViewController.h"
11
12  @implementation DeepThoughtsAppDelegate
13
14  @synthesize window;
15  @synthesize viewController;
16
17
18  #pragma mark -
19  #pragma mark Application lifecycle
20
21  - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
22
23      // Override point for customization after app launch.
24      [window addSubview:viewController.view];
25      [window makeKeyAndVisible];
26      DeepThoughtsViewController *viewController2 = [DeepThoughtsViewController alloc];
27                                                        Unused variable 'viewController2'
28      return YES;                     Potential leak of an object allocated on line 26 and stored into 'viewController2'
29  }
30
31
32  - (void)applicationWillResignActive:(UIApplication *)appl
33      /*
34
```

Potential leak of an object allocated on line 26 and stored into 'viewController2'

Build succeeded (4 analyzer results)

Figure 12-15: The Static Analyzer warns about a memory leak.

If you click the little blue icon for the warning (refer to Figure 12-15), you get a “trace” of what happened, as I show in Figure 12-16.



DeepThoughtsAppDelegate.m – DeepThoughts
Simulator – 4.2 | Debug | DeepThou...
Overview
Action
Breakpoints
Build and Run
Tasks
Info
String Matching
Search
Groups & Files
DeepThoughts
Constants.h
Classes
Other Sources
Resources
Frameworks
Products
Targets
Executables
Find Results
Bookmarks
SCM
Project Symbols
Implementation Files
Interface Builder Files
File Name
Code
DeepThoughtsAppDelegate.h
DeepThoughtsAppDelegate.m 56K
DeepThoughtsViewController.h
DeepThoughtsViewController.m 69K
SettingsViewController.h
SettingsViewController.m 49K
DeepThoughtsAppDelegate.m:26
-application:didFinishLaunchingWithOptions:
Done
Build succeeded (4 analyzer results)
Succeeded 4

1. Method returns an Objective-C object with a +1 retain count (owning reference)

```
 9  #import "DeepThoughtsAppDelegate.h"
10  #import "DeepThoughtsViewController.h"
11
12  @implementation DeepThoughtsAppDelegate
13
14  @synthesize window;
15  @synthesize viewController;
16
17
18  #pragma mark -
19  #pragma mark Application lifecycle
20
21  - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
22
23      // Override point for customization after app launch.
24      [window addSubview:viewController.view];
25      [window makeKeyAndVisible];
26      DeepThoughtsViewController *viewController2 = [DeepThoughtsViewController alloc];
27                                  Method returns an Objective-C object with a +1 retain count (owning reference)
28      return YES;
29  }         Object allocated on line 26 and stored into 'viewController2' is Method returns an Objective-C object with a ount of +1 (object leaked)
30                                                                          +1 retain count (owning reference)
31
32  - (void)applicationWillResignActive:(UIApplication *)application {
33      /*
34
```

Figure 12-16: The expanded Static Analyzer warning showing a trace of what happened.

First you get the following warning, which you can see by moving your pointer over the blue arrow icon in the trace (as shown in Figure 12-16):

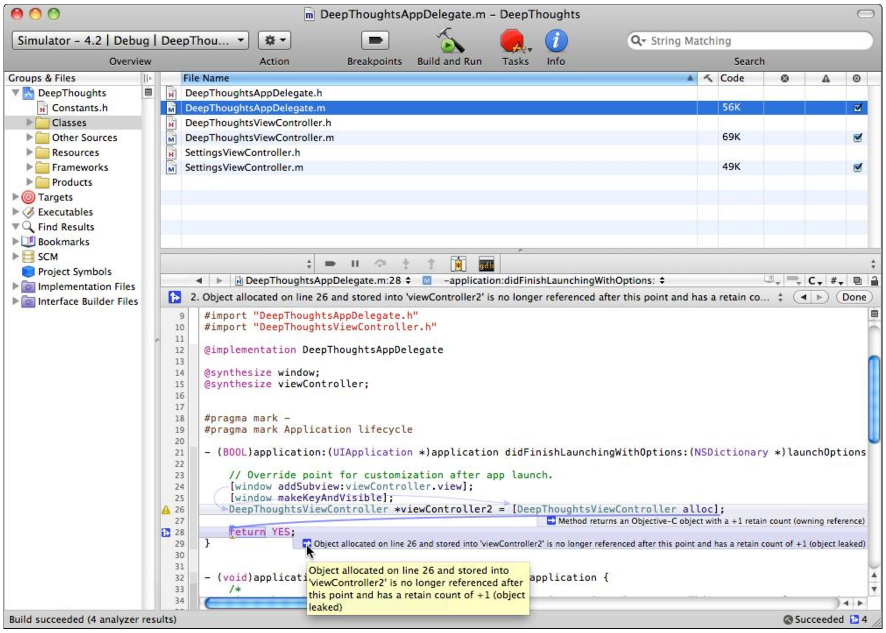
```
Method returns an Objective-C object with a +1 retain
count (owning reference)
```

Then, in the next line, if you move your pointer over the blue arrow icon as shown in Figure 12-17, you can see this:

```
Object allocated on line 26 and stored into
'viewController2' is no longer referenced after
this point and has a retain count of +1 (object leaked)
```

TIP

Notice that the results refer to line numbers. That’s why I made a point of explaining how to turn on line numbers in Xcode back in Chapter 5.



DeepThoughtsAppDelegate.m – DeepThoughts

Simulator – 4.2 | Debug | DeepThou...

File Name	Code	
DeepThoughtsAppDelegate.h		
DeepThoughtsAppDelegate.m	56K	✓
DeepThoughtsViewController.h		
DeepThoughtsViewController.m	69K	✓
SettingsViewController.h		
SettingsViewController.m	49K	✓

DeepThoughtsAppDelegate.m:28 -application:didFinishLaunchingWithOptions:

2. Object allocated on line 26 and stored into 'viewController2' is no longer referenced after this point and has a retain co...

```
 9  #import "DeepThoughtsAppDelegate.h"
10  #import "DeepThoughtsViewController.h"
11
12  @implementation DeepThoughtsAppDelegate
13
14  @synthesize window;
15  @synthesize viewController;
16
17
18  #pragma mark -
19  #pragma mark Application lifecycle
20
21  - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
22
23      // Override point for customization after app launch.
24      [window addSubview:viewController.view];
25      [window makeKeyAndVisible];
26      DeepThoughtsViewController *viewController2 = [DeepThoughtsViewController alloc];
27                                          Method returns an Objective-C object with a +1 retain count (owning reference)
28      return YES;
29  }       Object allocated on line 26 and stored into 'viewController2' is no longer referenced after this point and has a retain count of +1 (object leaked)
30
31
32  - (void)applicati                       application {
33      /*
34
```

Object allocated on line 26 and stored into 'viewController2' is no longer referenced after this point and has a retain count of +1 (object leaked)

Build succeeded (4 analyzer results)

Succeeded 4

Figure 12-17: The Static Analyzer uses line numbers in its warnings.

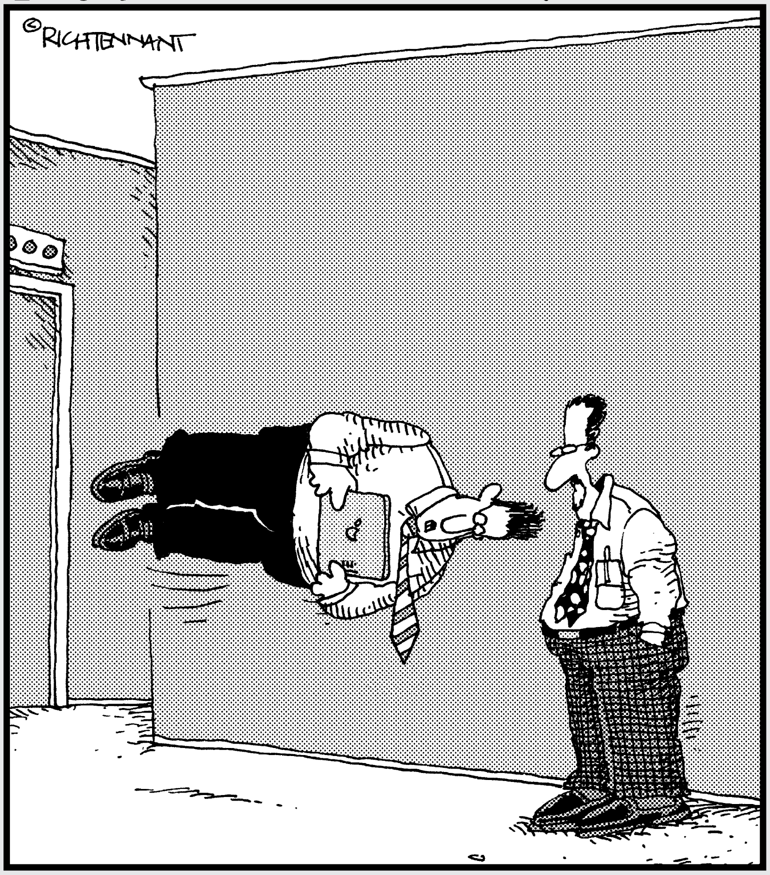
As you know by now, memory management is a big deal on the iPad.

Before you attempt to get your app into the App Store or even run it on anyone's iPad, you need to make sure it's behaving properly. By that I mean not only delivering the promised functionality, but also avoiding the unintentional misuse of iPad resources. Keep in mind that the iPad, as cool as it may very well be, is nevertheless somewhat resource-constrained when it comes to memory usage and battery life. Such restraints can have a direct effect on what you can (and can't) do in your application.

Now that you've meditated on DeepThoughts long enough to know the secrets of iPad app development, you're ready to tackle a truly industrial-strength application, which is displayed in all its glittering detail in Part V.

Part V
Building an Industrial-Strength Application

The 5th Wave By Rich Tennant



"Other than this little glitch with the landscape view, I really love my iPad."

In this part . . .

In this part, I explain the design of an application that has big muscles: a context-driven user interface, lots of functionality, Web access, an annotated custom map, and an application architecture that you can use to build your own version of The Next Great Thing.

- Chapter 13 takes you on a tour of how to start the whole process of designing and then building your app. You put yourself in the user's shoes and then take that understanding and transform it into a program architecture — one you can actually implement.
- Chapter 14 introduces you to Split view controllers, popovers, and table views — the primary ways that the user will discover all those neat things your app can do for them.
- Chapter 15 helps you find your way with maps. You find out about creating maps with MKMapView, centering them, displaying a region, and even pinpointing where you are.
- Chapter 16 shows you how to get to all that content that makes the iPad a superb user experience. You use data you've stored locally in your application bundle as well as data you have on a server someplace in the cloud (and want to save for later use when the user is no longer online), and you even display a Web page and allow a user to navigate out into the Internet and then back without ever leaving your app.
- Chapter 17 delves in to the Brave New world of iPad app printing — courtesy of the new iOS 4.2.
- Chapter 18 takes a further look at split views, showing how you can organize your content to display in Master and Detail views.
- Chapter 19 pays more attention to the user experience. You find out how to save the state of the application when the user quits and then restore it when he (or she) returns.

Chapter 13

Designing Your Application

In This Chapter

- Deciding the real value your application delivers
- Making functionality usable
- Creating the program architecture
- Identifying the technology you will use

Although the iPad can do almost anything that the iPhone can do (except the making calls stuff, and yeah, some models can use only Wi-Fi), you'll want to do certain things only on the iPad. (There are also some things you'll really prefer to do on the iPhone, but I'll leave that for you to explore on your own.)

In this chapter, I take you through an overview of the design cycle of a more complex application (iPadTravel411), and I show you how to take an idea that was developed for the iPhone and expand it to take advantage of the iPad's capabilities. Although I can't develop the entire application within the confines of this book, I show you how to take a subset of it and how to use the iPad's capabilities to implement it.

TIP

I start this chapter by explaining the app itself. As I start presenting the details, you may find things easier to follow if you first download the complete app from my Web site at `www.nealgoldstein.com`, compile it, and then play along with it during the discussion.

Defining the Problems

Innovation is usually born of frustration, and the iPadTravel411 project was no exception. It just turns out that my frustration was linked to a trip to beautiful Venice rather than, say, the vacuum cleaner doing a terrible job of picking up cat hair.

My wife and I were going to arrive late at night, and rather than trying to get into Venice at that hour, we decided we'd stay at a hotel near the airport and then go into Venice the next day. We were going to meet some friends who were leaving the day after that, and we wanted to get a relatively early start so we could spend the day with them.

I was a little concerned about the logistics. I thought we would have to go back to the airport terminal from the hotel and then get on a water bus or water taxi. Both the water taxi stand and the water bus stop are a distance from the terminal, and that meant more time and more trudging about. The water taxi was the fastest way, but very pricey (around $140 USD at the time). The water bus was much cheaper but more confusing — and only ran once an hour. It seemed like a major excursion.

My friends said, "Why not take a taxi or a bus?"

I said, "A bus to Venice — it's an island the last time I checked."

Okay, it *is* an island, but there's a causeway running from the mainland to Piazzale Roma, where you can then get a water bus or water taxi — or meet your friends.

Although it's more romantic to arrive by sea, it's a lot easier by land. Having been to Venice a couple times before, and considering our time constraints, we opted for the land route.

Now, I'm sure that information was in a guidebook someplace, but it would have taken a lot of work to dig it out; most guidebooks focus on attractions. Also, guidebooks go out of date quickly; the one I had for Venice was already two years old. Of course, I could have used the Internet before I left home to find the information, but that can also be a real chore. And, as I like to remember, "The great thing about the Internet is that you can find information about anything — and some of it is even true."

What I wanted was something that made it easier to travel by reducing all the hassles — getting to and from a strange airport, getting around the city, getting the best exchange rate, knowing how much I should tip in a restaurant — that sort of thing. (Not too much to ask, right?)

Don't get me wrong — I actually do a lot of research before I go someplace, and often I have that information handy already. But I end up with lots of paper because I usually don't take a laptop with me on vacation; even when I do, it's terribly inconvenient to have to take it out on a bus or in an airline terminal to find some information. And then there's the challenge of finding a Wi-Fi connection when you really need it.

I kept that idea in the back of my mind because at the time there was no real solution.

But then . . . enter the iPhone. After taking a look at the SDK, I realized I could write an app (without too much real difficulty) to do everything I thought would make traveling no more painful that a root canal. (Hey, my dentist does wonderful things with Novocain and nitrous oxide these days.)

Although I initially developed this application for the iPhone, when the iPad made its debut I realized that — for at least some parts of the application — the iPad was an even better solution. So I started by simply trying to port the application to the iPad. What I learned is valuable for anyone from the I-never-developed-anything-before developer to someone who already has an iPhone application in the App Store — so valuable, in fact, that I'm going to highlight it here:

WARNING!

The iPad is *not* simply a bigger iPhone, which means a simple Port may end up being an unmitigated disaster.

So, even though the goal of the iPadTravel411 app will remain the same, I take you through the process of designing the same solution to a user's problem for the iPad. (If you're curious about how I set up the original iPhone app, get yourself a copy of *iPhone Application Development For Dummies* and take a look at the MobileTravel411 and iPhoneTravel411 apps.) Much of the stuff is similar, but the user experience is far different, given both the size of the display as well as the available functionality in the SDK.

Categorizing the problems and defining the solutions

On desktop or laptop machines, features are often categorized by function, but given the way the iPad is designed to be used (as I describe in Chapter 1 and explain further in this chapter), categorizing by *context* makes more sense. So after I settled on the information and functionality I needed when I was traveling, I grouped things into the following contexts:

- **Getting and using money:** What is the country's currency (including denominations and coins), and what's the best way to exchange my currency for it? I want to understand the costs of using credit cards versus an ATM card, or exchanging at a *bureau de change.* I also want to be able to understand how the dreaded VAT (value-added tax) really works.
- **Getting to and from the airport:** What choices do I really have when it comes to things terminal? What are the costs, advantages, and disadvantages — and logistics — of each? Do I have to buy a ticket in advance? How do I find said ticket? What's the schedule?
- **Getting around the city:** Same kind of pickle as getting to and from the airport — what's available *and* best for a traveler's purposes? I once spent several days in Barcelona before I realized there was a subway system.

- **Seeing what's happening right now in the city:** Guidebooks are fine for visiting the sights, and I might want to (some day) re-create one on the iPad. But what I *would* like to know now is whether there's anything special happening when I'm in some particular place at some particular time. Bastille Day in Paris can be fun if you know about the Bastille Day parade, and less of a hassle if you know you can't cross the Champs-Élysées for a few hours.
- **Knowing the practical day-to-day stuff:** How do you make calls into, out of, and within a given city? How much and when should I tip? What is acceptable and unacceptable behavior? For example, that it's considered impolite to eat or drink something while walking down the street in Japan might not occur to someone from New York City.
- **Staying safe:** Being immediately informed of breaking news that could make things unsafe — large demonstrations or terrorist attacks, for example — would be high on my wish list. But even the more mundane things like the "dangerous" neighborhoods are important. What should you do in an emergency? A friend of mine had her passport stolen in Prague — at times like that, it would be nice to have the locations and phone numbers of embassies or consulates. This is stuff you hardly ever need, but when you need it, you need it right away.
- **What to do before I go:** In the past, I've forgotten to call my cell phone company before I leave home to get a roaming package and to notify my credit card company that I'll be out of the country or far from home, so please, *please* don't decline my hotel charge in Vladivostok. I also want to be able to download all the information before I leave so I can look at it on the plane, or as part of my strategy for avoiding roaming charges or handling an unexpected lack of connections.
- **Knowing where I am:** In all of these situations, I also want to be able to get my bearings by seeing where I am — and where I might need to get to — on a map.

I also wanted to make the app easy to use for someone who isn't intimately involved with the design — and perhaps doesn't immediately share my take on the best way to organize the information. So, for each choice in the main window, I wanted to be able to add a few words of explanation about what each category contained.

All great ideas, but as I said, the important thing is to know how to make an app actually fulfill the promise of all these great ideas. For that, you need someone to walk you through the design cycle of the application — and I'm nominating myself. Although you could use at least half a dozen models for the process (I'm a recovering software development methodologist myself), the one I go through here is pretty simple and is well suited for the iPad to boot.

The Great Application Cycle of Life

Here goes:

1. Defining the problems
2. Categorizing the problems and defining the solutions
3. Designing the user experience
    a. Leveraging the iPad's strengths
    b. Seeing what you have to work with when it comes to the device
    c. Recognizing the constraints of the device
4. Creating the program architecture
    a. Content views
    b. View controllers
    c. Models
5. Writing the code (and testing it along the way)
6. Doing it until you get it right

Of course, the actual analysis, design, and programming (not to mention testing) process has a bit more to it than this — and coming up with the specifications and design definitely involve more than what you see in these few pages. But from a process perspective, it's pretty close to the real thing. It does give you an idea of the questions you need to ask — and have answered — in order to develop an iPad application.

A word of caution, though. Even though iPad apps are smaller and much easier to get your head around than, say, a full-blown enterprise service-oriented architecture, they come equipped with a unique set of challenges. Between the iPad capabilities (which ironically become a kind of requirement for creating a good app) and the high expectation of iPad users, you have your hands full.

Designing the User Experience

Because you've already been through Steps 1 and 2 of my handy-dandy iPad application development design cycle — see the previous section — what I do next is talk about the user experience. After I've gone through all of that, I show you how to develop a subset of the application (I call the resulting app the iPadTravel411) in Chapters 14 through 17.

To be honest, I actually started the process of defining the user experience process earlier in the chapter when I defined the kinds of contexts I was interested in and the information and capabilities I wanted in each of them. (That's yet more proof that it's hard to compartmentalize experience into discrete steps.) But to get the actual *design* ball of my application rolling, I started out by thinking a bit more about what else (besides the features, of course) I wanted from the application — in other words, I started thinking about what the *experience* of using the application should be like.

TIP

To further the process, you would actually want to model the user *work-flow,* so to speak — how the user would want to use the information and capabilities you could provide in each of those contexts I identified in the "Categorizing the problems and defining the solutions" section, earlier in this chapter.

Although sketching out such a workflow completely is beyond the scope of this book, I do want to explain how knowing what you have to work with (the iPad's strengths and features) as well knowing what the device won't let you do (or, at least, not do without a fight) can help you define what your (and the user's) options are. To start that process of knowledge acquisition, I want to review some of the things that the iPad is really good at.

Leveraging the iPad's strengths

Although I've had a lot of experience designing, developing, and writing about iPhone applications, soon after the iPad was announced I began to realize that — contrary to what many of the pundits have said — the device isn't simply a larger iPhone or iPod touch. In fact, even though iPhone/iPod touch applications could be ported to the iPad without too much trouble, in order to offer real value on that device, many would have to be redesigned. (In fact, some iPhone/iPod touch applications weren't even relevant for the iPad.)

Although the iPad shares some features with the iPhone — portability, ease of use and convenience, its awareness of your location, and its ability to connect seamlessly to the Internet from most places — it is also significantly different. Two differences jumped out at me:

- The iPad is not as compact as an iPhone, for example, so the user is less likely to have the device with them all of the time, as they would with a cellphone or smartphone.
- The screen size is great for displaying lots and lots of content — not one of the iPhone's strengths, if I'm being totally honest.

What I realized is that these two differences, when put together, provide for the ability to allow the user to explore whatever interests them in a more intimate or personal way. So rather than a device that was ideal for short, ad hoc tasks, what you have is a device well-suited for longer-term, more intensive exploration of a subject.

Not that you can't do that on an iPhone; it's just that the iPhone is better for really short-term tasks that require no more than a limited amount of specific information that can be delivered on the small screen. In that sense, iPhones and other small-screen devices are all about execution as opposed to planning and/or immersion in a subject. Which is, on the other hand, the strong suit for the large-screen iPad.

So what do you get when you combine the iPad's form factor, touch interface, ease of use, and portability with its ability to beautifully display rich content, its awareness of your location, and its ability to connect seamlessly to the Internet from most places? You get a more intimate device with a better way to access information — one that allows the user to explore topics in the way they want to and one on which you can create a more natural interface that's consistent with the way the user wants to work.

Just think about it: The iPad is meant to be able to be used in any orientation, and the ability to flip the device over to share it with someone is a natural extension of that.

It starts with the last thing I mentioned in a previous paragraph: creating a user experience that's based on the way people naturally want to work. Among other things, you become the champion of relevance, searching out and destroying anything that isn't relevant to what the user is doing while he or she is using a particular part of your application.

Knowing the location of the user enables you to further refine the context by including the actual physical location and adding that to the crucial "relevance" filter. If you're in London, the iPad is well aware of that fact, meaning your application can "ask" the user whether he or she wants to use London as a filter for relevant information.

The idea is to focus on delivering rich content, understanding that the quality of information has to be better than the alternative — what you get by using the application has to have more value than alternative ways of doing the same thing. I can find airport transportation in a guidebook, but it's not up to date. I can get foreign exchange information from a *bureau de change,* but unless I know the bank rate, I don't know whether I'm being ripped off. I can get restaurant information from a newspaper, but I don't know whether the restaurant has subsequently changed hours or is closed for vacation. If the application can consistently provide me with better, more up-to-date information, it's the kind of application that's tailor-made for a context-driven design. This sort of design is possible on a mobile device because the device can access the Internet, which allows you to provide real-time, up-to-date information. In addition, it enables you to transcend the CPU and memory limitations of the iPad by offloading processing and data storage out to a server in the cloud.

What you have to work with

Okay, it's time to check the windows situation. On the Mac (or any other PC) you have lots of windows, and lots of different kinds of windows. On the iPhone, on the other hand, you have a single window with an occasional action sheet or alert.

The iPad falls somewhere in between the superabundant and the almost non-existent. You have the following in your bag of tricks:

- Full-screen views
- Split views
- Popover views
- Controls, less than full-screen modal dialogs, action sheets, and alerts

Full-screen views

On the iPad, you have the luxury of a large 9.7-inch (diagonal), LED-backlit, glossy, widescreen, Multi-Touch display with 1,024 x 768-pixel resolution at 132 pixels per inch.

Figure 13-1 shows one of the things you could fill it with.

Although one of the constraints on your application design for the iPhone and other mobile devices is the small screen, ironically the large screen on the iPad can also be thought of as a constraint. If you don't fill it (correctly) it can look really bad.

Split views

The split view, as you can see in Figure 13-2, was introduced in version 3.2 of what is now called the iOS SDK and is an iPad-only feature. You'll definitely be taking advantage of split views as you build the subset of iPadTravel411.

The split view enables you to display two views side by side. In this example, the user has an opportunity to navigate to a particular part of the application while looking at a map of London. In Figure 13-2, the view on one side of the split is a Navigation view, whereas the view on the other side is a Content view (I get to that next), but you can display anything you'd like.

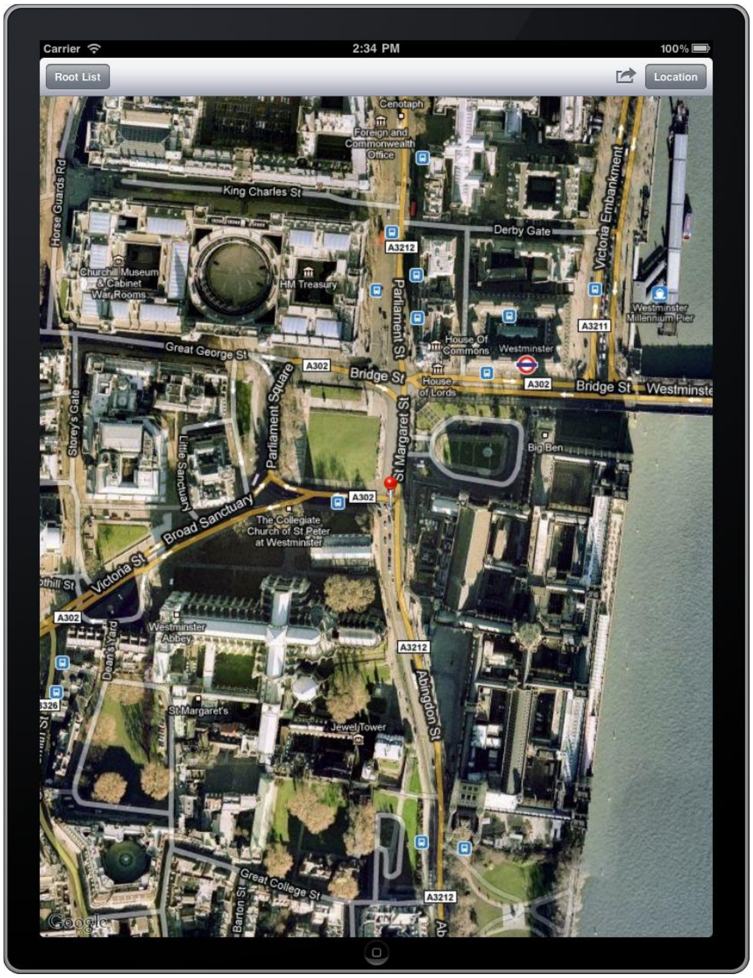


Figure 13-1: Nice view.

Popovers

Although split views work well for landscape mode, in portrait mode the left side disappears. Instead of a split view, you can create a Popover view that displays the same information that you had in the split view (in Figure 13-2 for example), or actually anything else you would like. You can see an example of that in Figure 13-3, where you have a nice little Map view of Paddington Station in London in the background and instructions on how to get there from Heathrow airport on the Heathrow Express making up the Popover view.

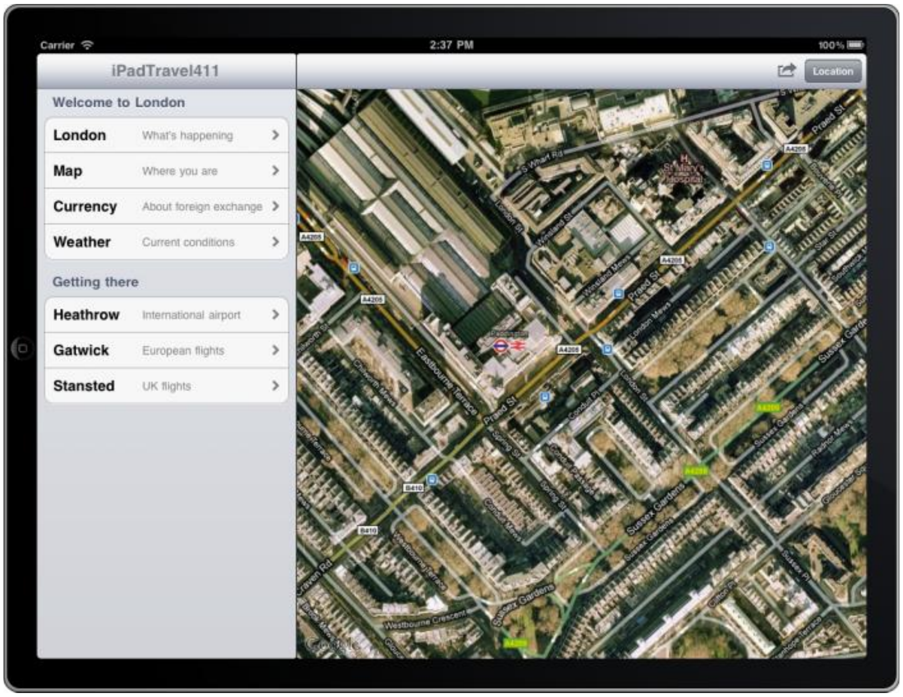


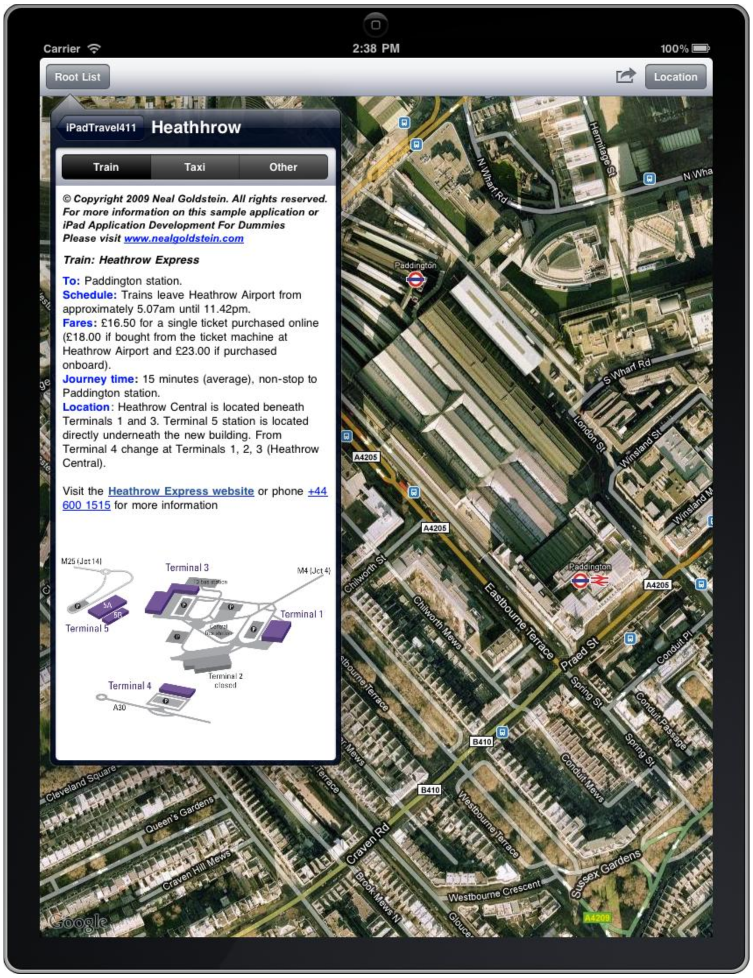
Figure 13-2: Split view.

Controls, less than full-screen modal dialogs, action sheets, and alerts

Controls, small modal dialogs, action sheets, alerts — all of these items let the user navigate the application. Controls allow the user to control the application — determining what they want to see, for example, or even letting them enter data. (You worked with controls when you developed the DeepThoughts app back in Part IV of this book.) In Chapter 19, I introduce you to the control you see in Figure 13-3. This fellow is known as a *segmented control,* and it enables the user to choose the transportation information he or she wants to see in the Popover view — Train, Taxi, or Other.

Device constraints

Although there are a host of possibilities on the iPad, you also need to live within some constraints. This means that you not only have to take into account the user context when designing an application, but you also need to take into account the device context.



© Copyright 2009 Neal Goldstein. All rights reserved. For more information on this sample application or iPad Application Development For Dummies Please visit www.nealgoldstein.com

Train: Heathrow Express

To: Paddington station.
Schedule: Trains leave Heathrow Airport from approximately 5.07am until 11.42pm.
Fares: £16.50 for a single ticket purchased online (£18.00 if bought from the ticket machine at Heathrow Airport and £23.00 if purchased onboard).
Journey time: 15 minutes (average), non-stop to Paddington station.
Location: Heathrow Central is located beneath Terminals 1 and 3. Terminal 5 station is located directly underneath the new building. From Terminal 4 change at Terminals 1, 2, 3 (Heathrow Central).

Visit the Heathrow Express website or phone +44 600 1515 for more information

Figure 13-3: A Popover view.

After all, the device is also a context for the user. He or she, based on individual experience, expects applications to behave in a certain way. As I explain in Chapter 1, this expectation provides another perspective on why staying consistent with the user interface guidelines is so important.

If you want to maximize the user experience, you have to take the following into account (I know I went through these in Chapter 1, but remembering them is critical):

- **Filling screen real estate:** On the iPhone, you may have had the problem of *too much* content. Although scrolling is built in to an iPhone and is relatively easy to do, folks don't particularly like to scroll on an iPhone, meaning you should require as little scrolling as possible, especially on navigation pages and on the main page. On the iPad, the opposite situation — *not enough* content — may end up being a real challenge. You need to fill that big, beautiful screen with rich, useful content. This means that many of the techniques you used on the iPhone to minimize screen displays work against you on the iPad. If you flip ahead to Figure 13-7 for example, you can see what happened when I tried to do a simple port of my app. There's no reason to devote all of the screen space to this view. I should fill it up with useful information for the user — other than what he or she can do next.
- **Limitations of a touch-based interface:** Although the Multi-Touch interface is an iPad feature, it brings with it limitations as well. Fingers aren't as precise as a mouse pointer, and user interface elements need to be large enough and spaced far enough apart so that the user's fingers can find their way around the interface comfortably. You also can do only so much with fingers. There are definitely fewer possibilities using fingers than when using the combination of multi-button mouse and keyboard. Even though the iPad does offer a "real" keyboard option, for most applications you'll want to take advantage of the touch-based interface.
- **Limited computer power, memory, and battery life:** As an application designer for the iPad, you have to keep practical issues like power and memory limitations in mind. Although the iPad definitely has more going for it in these matters than the iPhone, you still need to be realistic about what it has under the hood.
- **Connection limitations:** There's always a possibility that the user may be out of range, or on a plane, or has decided not to pay exorbitant roaming fees, or is using an iPad model that doesn't have Internet access except via Wi-Fi. You need to account for that possibility in your application and preserve as much functionality as possible. This usually means allowing the user to download and use the current real-time information, where applicable.

Coming up with a final design

After carefully thinking about all of the things I wanted the app to be able to do as well as meditating on the possibilities available to me on the iPad, and the limitations of the device (and a few glasses of wine with my partners in crime), the final user interface I came up with looks like Figure 13-4.

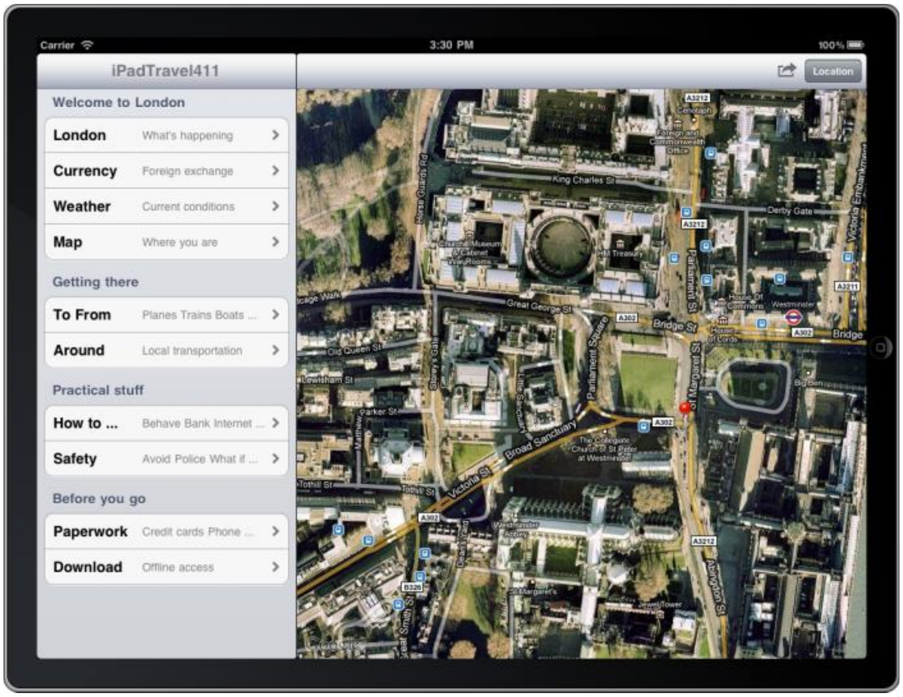

Carrier
3:30 PM
100%
iPadTravel411
Location
Welcome to London
London What's happening
Currency Foreign exchange
Weather Current conditions
Map Where you are
Getting there
To From Planes Trains Boats ...
Around Local transportation
Practical stuff
How to ... Behave Bank Internet ...
Safety Avoid Police What if ...
Before you go
Paperwork Credit cards Phone ...
Download Offline access

Figure 13-4: The full application.

In Figure 13-5, you can see the subset of features that I show you how to implement in Chapters 14 through 19. I chose this subset because it gives me the opportunity to show you the most how-to-do-that information without it being overwhelming.

REMEMBER

Part of making the app easy to use involves giving users a way to set their preferences for how the app should work. In Figure 13-6, you can see the iPadTravel411 Settings view. This setting allows the user to specify that he or she wants to work in a *stored data mode* — using previously stored data, rather than the current real-time version that would require Internet access. The idea is to download the information the user needs before he or she leaves the safe world of Wi-Fi. This is a requirement for iPad models without 3G, and a necessity for 3G models when the user is abroad and he or she wants to avoid data roaming charges and thus be able to afford food other than ramen noodles on the trip.

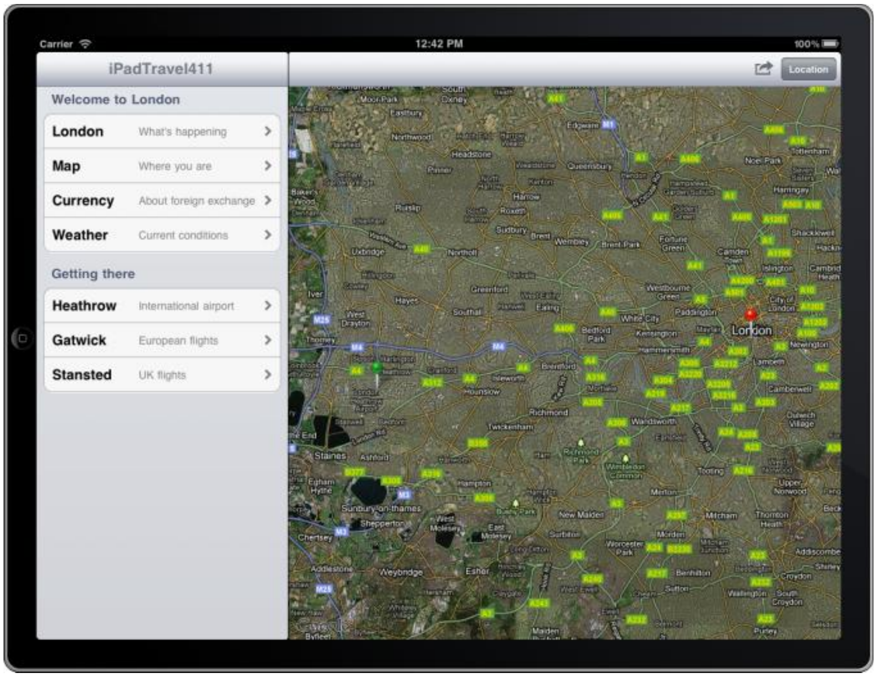
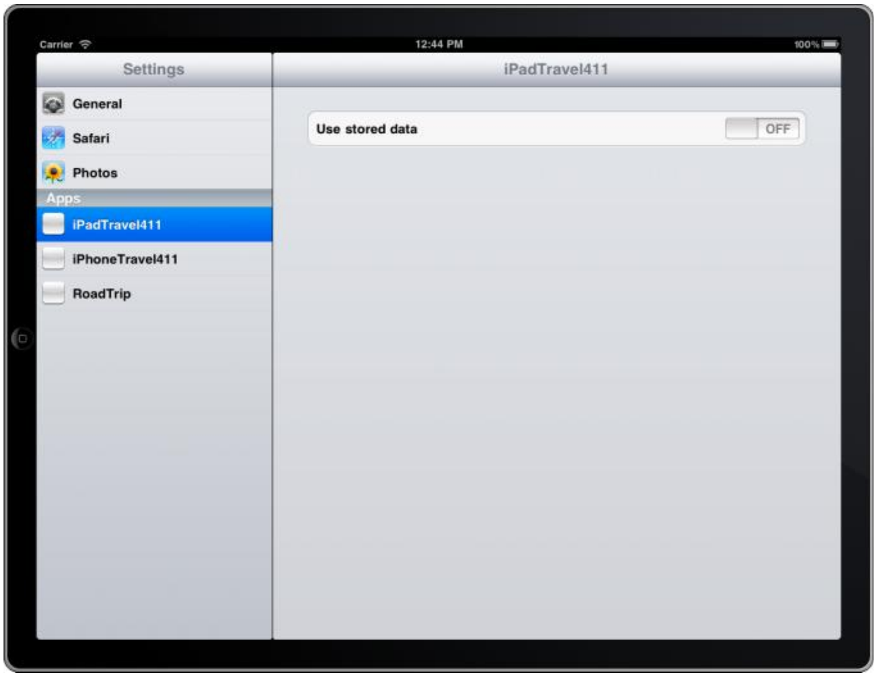


Figure 13-5: Your user-friendly subset.



Use stored data OFF

Figure 13-6: Use Stored Data preference.

Creating the Program Architecture

After you've come up with a user interface capable of delivering the kind of the user experience you've defined, you need to map that onto a program architecture.

Keeping things at a basic level, you can think of the iPadTravel411 application architecture as being made up of the following:

- **Models:** Model objects encapsulate the logic and (data) content of the application. (You may remember that there was no model object in the app DeepThoughts from Part IV, per se.) For iPadTravel411, I show you how to design, implement, and use model objects.
- **Views:** Views present the user experience; you have to decide what information to display and how to display it. In the DeepThoughts application, there was a Content view and a Modal view with controls as subviews. Now, with the iPadTravel411 app, you're going to be working with a Navigation view and several Content views. You'll also want to add a toolbar (control) in one set of views to allow the user to specify which type of airport transportation he or she would like information about: train, taxi, or other.
- **Controllers:** (They're known as *view controllers* in the iOS SDK.) Controllers manage the user experience. They connect the views that present the user experience with the models that provide the necessary content. In addition (as you'll see), controllers also manage the way the user navigates the application.

No big surprises here — especially because the MVC model (Model-View-Controller) is pretty much the basis for *all* iPad application development projects. The trick here is coming up with just the right views, controllers, and model objects to get your project off the ground. Within the requirements I spell out in the "Designing the User Experience" section earlier in the chapter, I came up with the elements highlighted in the next few sections.

I'm going to start with the views because they determine the functionality and information available in a given context — being at an airport and needing to get into the city, for example.

Views

Views offer the application's face to the world. They enable a user to

- See content.
- Navigate the application.
- Provide input into the application (both instructions and data) through controls.

Views can be categorized even further, as the next sections make clear.

Content views

Content views display the information the user wants — show me where I am on a map, or tell me the best way to get from Heathrow Airport to London.

As shown earlier in the chapter, the view in Figure 13-1, the view on the right side of Figure 13-2, and both views in Figure 13-3 are Content views, for example.

As I explain in Chapter 7, there are several kinds of views you can use on the iPad to display the various kinds of information you want to provide your users. In Chapters 14 through 19, I show you how to use the following view classes that come with the SDK:

- `UIWebView`
- `MKMapView`
- `UIImageView`

For now, I just want to highlight some of the main features of each view class.

UIWebView

`UIWebView` is especially good for displaying two specific kinds of content: text-based formatted data and Web content.

Highlighting text-based formatted data in this context may have come as a bit of a surprise, but Web views make it easy to access data from a central repository on the Internet. (Client-server is alive and well!) Because some of what I want to do needs to be updated regularly — if I want the current price and schedule of the Heathrow Express, for example, data from last year (or even last week) may not help me — being able to grab the most up-to-date information from the Web is a definite plus. I also want the most current information about what's happening in the city I plan to visit.

As for other benefits of Web views, keep in mind that real-time access isn't always necessary — sometimes it's perfectly fine to store some data on the iPad (and you will). It turns out that Web views can easily display formatted data that's locally stored — very handy. For example, the basics of foreign exchange are what they are — they're not going to change from today to tomorrow. That means the user doesn't need up-to-the-minute information on that particular topic.

The fact that `UIWebView` is great at displaying Web content should come as no surprise. If users want more information on the Heathrow Express, they can get to the Heathrow Express Web site by simply tapping a link. Web views allow you to manage the navigation out from your app, on to the Internet, back and forward in the history of Web pages, and then back to the originating view, without ever leaving the app.

MKMapView

`MKMapView` enables you to easily display maps similar to the one provided by the iPad's own Maps application. You use this class to display map information and to manipulate whatever gets displayed on the map from your application. You can center the map on a given coordinate, specify the size of the area you want to display, and annotate the map with custom information.

UIImageView

`UIImageView` allows you to display either a single image or animate a series of images. For animating the images, the `UIImageView` class provides controls to set the duration and frequency of the animation. You can also start and stop the animation freely.

In Chapter 9 you've already used `UIImageView` in the DeepThoughts application. In iPadTravel411, you can use Image views in a number of ways — displaying pictures of what you're talking about comes to mind. In the case of the subset of the application that you develop in Chapters 14 through 19, you'll display a graphic image that allows the user to navigate the maze known as Heathrow Airport, as you can see in Figure 13-3.

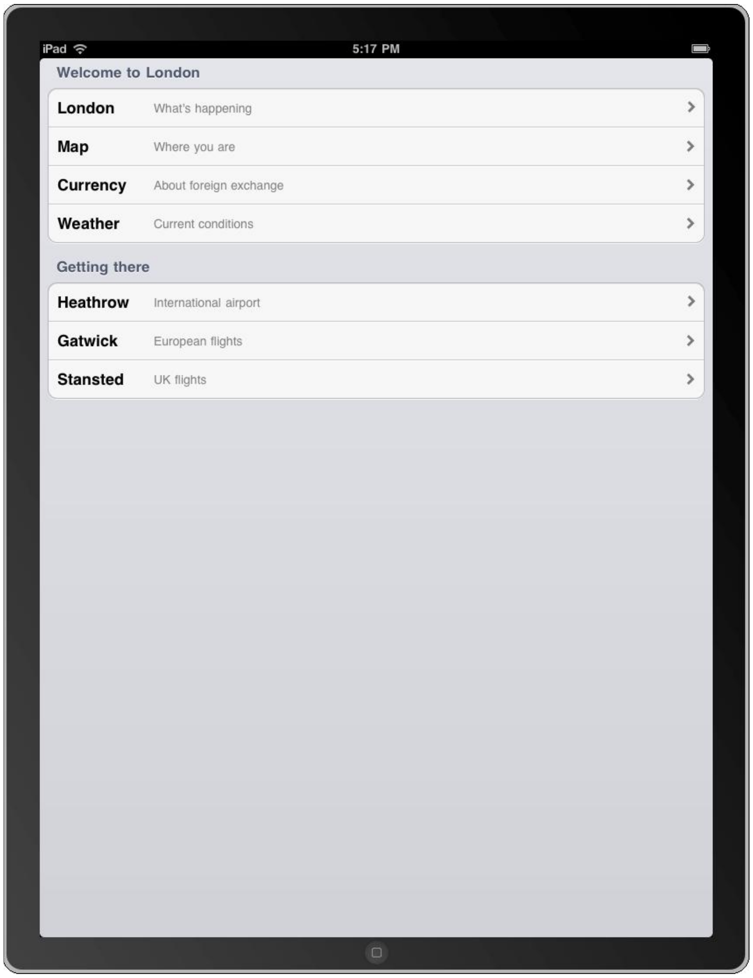
It turns out that there's one more view that you can use to display content: `UITableView`. But because you'll primarily be using it for navigation, I cover it in the following section.

Navigation views

Because of the screen real estate on the iPhone, most applications were designed around a Main view (which was primarily navigation based) and additional views (which could be content or navigational or even both) that were swapped in and out.

In the iPad, the need for a main Navigation view disappears; in fact, having one would probably result in a pretty bad application interface. After all, who wants to see something like Figure 13-7 with all that dull and empty space when it would be so much nicer to fill some of that valuable space with some actual content?

On the other hand, offering some kind of navigation help, as shown earlier in Figure 13-2, can be useful to a user trying to make her way through your app. The help you see in Figure 13-2 is provided by `UITableView`.



Welcome to London

London	What's happening
Map	Where you are
Currency	About foreign exchange
Weather	Current conditions

Getting there

Heathrow	International airport
Gatwick	European flights
Stansted	UK flights

Figure 13-7: Don't do this!

An instance of UITableView (or simply, a Table view) is a means for displaying and editing hierarchical lists of information. As such, they're used a lot in iPad applications to do two specific things:

- **Display hierarchal data:** Think of the iPod application, which gives you a list of albums, and if you select one, a list of songs.
- **Act as a table of contents (or for my purposes, contexts):** Now think of the Settings application, which gives you a list of applications that you can set preferences for. When you select one of those applications from the list, it takes you to a view that lists what preferences you're able to set as well as a way to set them.

You find out all about Table views — and using them as well — in Chapter 14.

Controls, less than full-screen modal dialogs, action sheets, and alerts

Controls are things the user can manipulate in order to "tell" the application what he or she wants it to do. Controls are derived from UIControl. They can operate on the data or even provide navigations. You're already quite familiar with controls from the DeepThoughts app. The controls you see on the iPad — and the ones you used in the DeepThoughts application — are views as well, and, as I explain in Chapter 7, are a subclass of UIControl.

In the design of the iPadTravel411 app, for example, you would use a control to allow a user to enter a dollar value and then display the amount in British pounds.

Although I won't be showing you that particular trick, I will be showing you how to use a UISegmentedControl to allow the user to choose the transportation information he or she wants to see in the Popover view — Train, Taxi, or Other. (You can see that segmented control in the Popover view back in Figure 13-3.) This particular control resides in a toolbar, which can also hold buttons, but the segmented control can actually be placed anywhere in a view.

Modal dialogs (those windows that display a dialog requiring the user to do something before he can get on with whatever he's doing in the app) are also views, but what makes them special are their view controllers. Action sheets and alerts are also views, and the same is true of them. I show you how to use an alert in Chapter 19, and action sheets are very similar.

View controllers

View controllers are responsible not only for providing the data for a view to display, but also for responding to user input and navigation requests.

They connect the model, which owns the data, with the view that displays the data and receives user input. The view controllers you'll use are all going to be derived from (a subclass of) `UIViewController`. You'll create custom view controllers to manage the data displayed in your Map view and Web views by subclassing `UIViewController`. These view controller subclasses will also implement delegate protocols as needed.

In addition, you'll be subclassing `UITableViewController` to manage the data displayed — take a look at the left side view in Figure 13-4 — as well as any user selection in a Table view, and create instances of `UISplitViewController` and `UIPopoverController` to manage the split view you see in Figure 13-2 and the Popover view in Figure 13-3, respectively.

Models

Although I could write a book on model design (in fact, I've written a couple, not to mention an Apple video — but that's another story), I want to concentrate on a couple things now to keep you focused.

The *models* own the data and the application logic. In the iPadTravel411 application, for example, a model object would convert U.S. dollars to pounds (or any other currency) and vice versa. This kind of model is closely tied to the functionality of the view it supports. The How Many Zimbabwean Dollars Can I Get For $2.75 (US) view requires a model that can compute exchange rates, and here's where the real-world objects associated with object-oriented programming come into play. In the full-blown iPadTravel411 app, I have a Currency (model) object that knows how to compute exchange rates, and I have a VAT (value-added tax) object that does something similar. So for each view like those two, I create a model object.

You have a couple of options when it comes to creating the model objects needed by the view controllers. One way is to have the view controllers themselves create the ones they'll use. For example, the `AirportController` would create the `Airport` object, and so on.

Although this does work, and I've actually done that in past versions, I'd like you to consider a different approach that results in a more extensible program. This approach is based upon creating a single model class that provides an interface to the view controllers, hiding from them any knowledge of how the model is constructed as well as which specific objects make up the model. (I explain this whole process in detail in *Objective-C For Dummies,* so if you're curious, you might want to pick up a copy of that book.)

One of the advantages of the MVC design pattern I explain in Chapter 2 is that it allows you to separate these three groups — the model, the view, the controller — in your application and work on them separately. If each group has a well-defined interface, it encapsulates many of the kinds of changes that are often made so that they don't affect the other groups. This is especially true of the model and view controller relationship.

If the view controllers have minimal knowledge about the model, you can change the model objects with minor impact on the view controllers.

As I said, what makes this possible is a well-defined interface. In Chapters 14 through 19, you create such a well-defined interface between the model and the controllers by using a technique called *composition,* which is a useful way to create interfaces.

Composition uses individual objects to carry out the roles and responsibilities declared in the model interface, so you don't need to have all the functionality in one bloated object. But it makes things easy to change by hiding those objects from the other objects that really end up using them. I'm a big fan of composition because it's another way to hide what's really going on behind the curtain. It keeps the objects that use the composite object ignorant of the objects the composite object uses and actually makes the components ignorant of each other, allowing you to switch components in and out at will.

The `Destination` class is going to be the basis for such an architecture, and even though I don't fully implement it here, Chapters 14 through 19 should give you enough background so you can understand the structure and have no trouble extending it on your own.

When it comes to the various Content views you need for the iPadTravel411, some of them clearly have more complex data requirements than others. For example, when transportation data is needed, if the user is in a real-time mode (that is, not using stored data), the data is downloaded from a server and then stored in file on the device. When the user has specified the stored data mode in his preferences, the data that was previously saved in the file is used. In a complex case like this, `Destination` creates and uses a model object (in this case, `Airport`), encapsulating the knowledge of what objects make up the model from the object that uses it. In less complex situations, the `Destination` object manages the data itself.

All the model objects are of a subclass `NSObject`, because `NSObject` provides the basic interface to the runtime system. It already has methods for allocation, initialization, memory management, introspection (what class am I?), encoding and decoding (which makes it quite easy to save objects as "objects"), message dispatch, and a host of other equally obscure methods that I don't get into but that are required for objects to be able to behave like they're expected to behave in an iOS/Objective-C world.

What I'm going to do in the iPadTravel411 application is actually have you create a model interface object and several model objects and view controllers to illustrate what you need to know about the model, view, and (view) controller relationship, how to access and display data stored locally or on a server, as well as how to simply display a Web site. That will be enough to keep you busy for a while.

Stored data mode, saving state, and localization

By using the application design I've described, adding all the features I mention in this section's little heading is easy; I explain them as I work through the implementation in Chapters 14 and 19. Although I don't dig too deeply into localization in this book, I show you how to build your application so that you can easily include that handy feature in your app.

Writing the Code

For me, writing the code is the fun part. I've been known to start working at 5 a.m. and quit at 2 a.m. the next morning because I was having so much fun. I help you have that kind of fun in Chapters 14 through19.

The iterative nature of the process

If there's one thing I can guarantee about development, it's that *nobody gets it right the first time.* Although object-oriented design and development are in themselves fun intellectual exercises (at least for some folks), they're also very valuable. An object-oriented program is relatively easier to modify and extend, not just during initial development, but also over time from version to version. (Actually, "initial development" and "version updating" are the same thing; they differ only by a period of rest and vacation between them.)

The design of my iPadTravel411 application evolved over time, as I learned the capabilities and intricacies of the platform and the impact of my design decisions. What I've tried to do in this chapter, and the ones following, is to help you avoid (at least most of) the blind alleys I stumbled down while developing my first application. So get ready for a stumble-free experience. On to Chapters 14 through 19.

Chapter 14

Working with Split View Controllers and the Master View

In This Chapter

- Working with Split view controllers
- Checking out the versatile Table view
- Making the table look good
- Ensuring that your application is usable worldwide
- Peeking behind the Table view screen
- Using model objects

Views are the user's window into your application; they are the gateway to the user's experience of your app. Their associated view controllers manage the user experience by providing the data displayed in the view, as well as by enabling user interaction.

My running example here is the iPadTravel411 application described in Chapter 13. Space prohibits dotting every *i* and crossing every *t* in implementing the application, but I can show you how to use the technology you need so you can do the detailed work on your own. (In order to examine memory management for example, you'll have to look in the code listings themselves.) You'll come across places where I'll suggest better (or alternative) ways to do things or ways to extend the application that space does not permit me to explain — things I'll leave up to you to implement. And even though I don't have the complete listings in this book, copies are available on my Web site at `www.nealgoldstein.com`.

Now, back in Chapter 13 I show you a screen shot of the iPadTravel411 application with a Table view — acting as a Navigation view on the left side (called the *Master view*) — and a Map view on the right side (called the *Detail view*). These two views make up the views in a Split view controller.

In this chapter, you get a closer look at both Split view controllers and the iPadTravel411 Master view — the view that allows the user to navigate the application — as well as the view controller that enables it. I show you how use a Table view as the Master view in a Split view controller.

In the chapters that follow, I show you how to implement the Detail views that you set up to deliver the content of your application — stuff like maps, what you need to know about currency, or views that let you check on the weather in London, Heathrow, or Greenwich — a few other things as well.

My advice to you: Pay careful attention to the Split view controller and the Table view. They are both very powerful classes that you'll use extensively in your apps. The Split view controller allows you to first display the Table view in landscape mode and then, when the user has turned the iPad to portrait mode, take the same Table view and display it in a popover window. (Neat.) The Table view is one of the basic views used in iPad and iPhone applications — the Mail and iPod applications come quickly to mind.

The Split View Controller

The `UISplitViewController` class is a view controller that simply manages the presentation of two side-by-side view controllers — it is, in this respect, a container controller. Using this class, you create a view controller on the left (the *Master view,* as I call it), which presents a list of items, and another view controller on the right, which presents the details, or content, of the selected item (the *Detail view,* as I call it).

The `UISplitViewController` class is what makes it possible for the iPadTravel411 app to look the way it looks in the upcoming Figure 14-7.

After you create and initialize a `UISplitViewController` object, you assign two view controllers to it by using the `viewController` property. The Split view controller has no interface — its job is to coordinate the presentation of its two view controllers and to manage the transitions between different orientations.

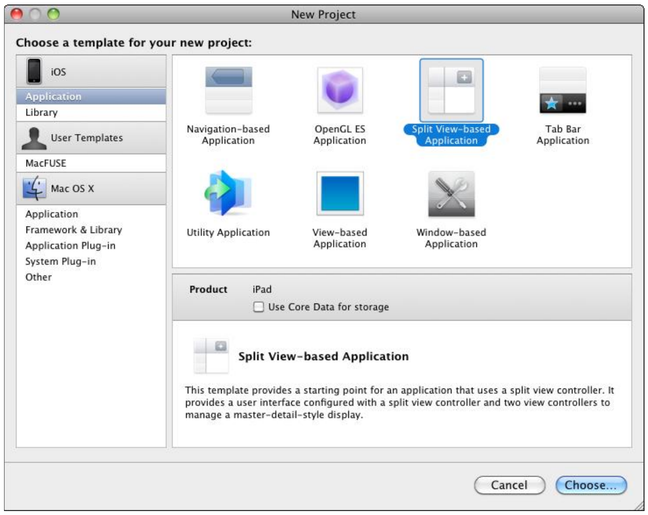
What's more, the Split view controller doesn't manage the communication between the two view controllers you assign to it. It's your responsibility to determine the best way to do that. I show you one way in this chapter.

To manage transitions between orientations, the Split view controller not only takes care of the mechanics of the transition, but also changes the display to effectively accommodate the orientation. In a landscape orientation, you can see both view controllers' views side by side. When the Split view controller rotates between portrait and landscape orientations, however, it can either hide or show the first view controller (the Master view controller by definition) in its array of view controllers. When the view controller is hidden, you can add a button to the toolbar of the remaining view controller that will display the hidden view controller in a popover.

The adding of a button is implemented in a delegate protocol. The popover sends its delegates messages to coordinate the display of a popover with the hidden view controller — the methods of this protocol are designed to be invoked at the right time so that you can add and remove the button.

To make all of this happen, fire up Xcode and officially launch the iPadTravel411 project. (If you need a refresher on how to set up a project in Xcode, take another look at Chapter 4.) As you can see in Figure 14-1, you need to go with a Split View-based Application template. Also be sure that the Use Core Data for Storage check box is not selected.

When you select Choose, you see a standard Save sheet. I saved the project as iPadTravel411 in a folder on my desktop.



Product iPad

☐ Use Core Data for storage

Split View-based Application

This template provides a starting point for an application that uses a split view controller. It provides a user interface configured with a split view controller and two view controllers to manage a master-detail-style display.

Figure 14-1: The Split View-based Application template for an Xcode project.

You see your new project's Groups & Files list on the left side of the Xcode Project window.

If you were to compile and run this project as-is, what you would see is in Figure 14-2.

TIP

Your new project will compile and then run on the device by default. You may want to change that default to Simulator in the Overview menu. If you do want to run the app on your iPad at this point, you might want to review Chapter 12 to refresh your memory about iPad provisioning.

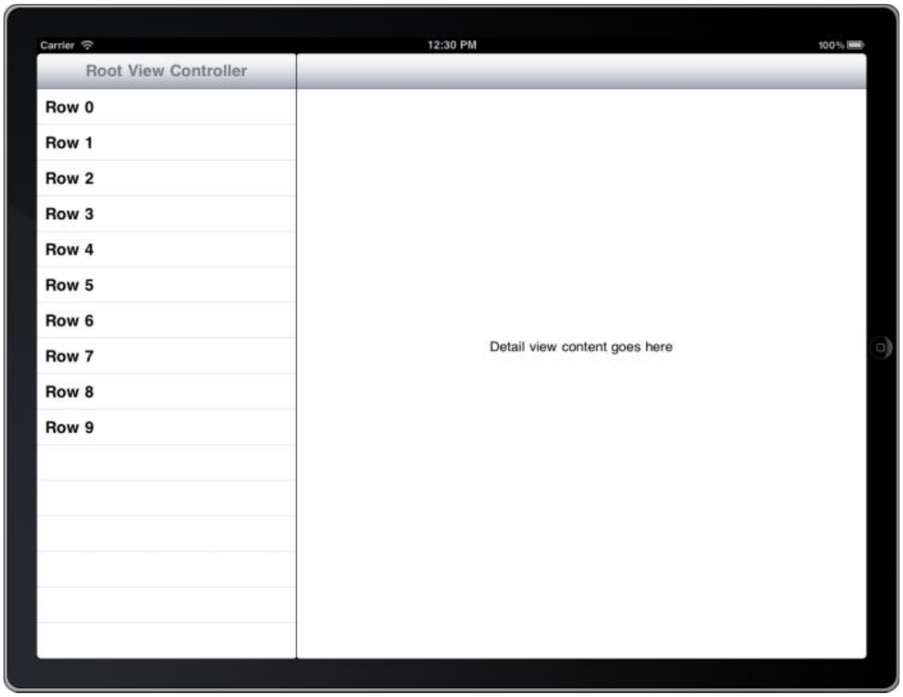


Figure 14-2: The app in landscape mode.

While still having a ways to go, this template does provide exactly what we're looking for in our application. On one side is a Navigation view (the Master view), and on the other side a Content view (the Detail view). If you were to rotate the iPad, as you can see in Figure 14-3, the Navigation view goes away, but there's a nice button there to get it back. (See Figure 14-4.)

As you can see, this is exactly the "infrastructure" needed for this app — all that with the press of a button.

Although the template provides almost everything you need, in this version of the SDK, for some reason, it does not include a stub for `application DidEnterBackground`. I explain in Chapter 19 that you're going to need to use the `applicationDidEnterBackground` method to put things on hold when your application goes into the background and I'll have you add that method there.

Click here to see the Navigation view.

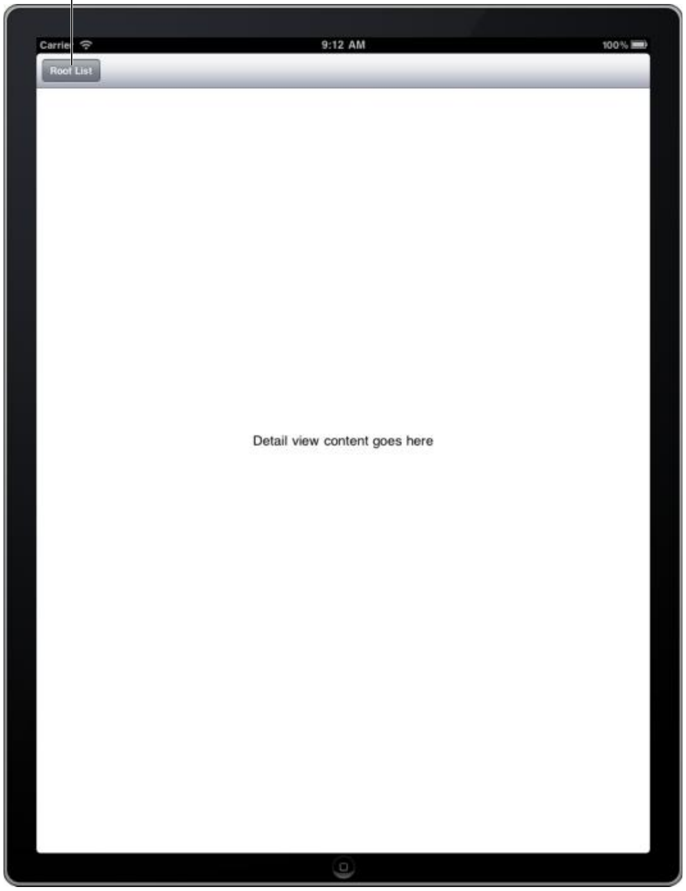


Figure 14-3: In portrait mode with a button to display the hidden Navigation view.

Now, of course, comes the hard part: actually adding all the content. Go ahead and start out with the Navigation view — a *Table view,* in this case — on the left side of the Split view controller, but before you do, I want to explain how all this works.

As you can see, this is exactly the "infrastructure" needed for this app.

As I mention earlier, after you create and initialize a `UISplitViewController` object, you assign two view controllers to it. Fortunately, that's done for you in the `MainWindow.xib` file created by the template. To see what is in the file, open the disclosure triangle next to Resources in the Groups & Files. ***Note:*** For your Interface Builder to look like what I'm showing you in Figure 14-5, you'll have to choose List view (the horizontal parallel lines) in the View Mode control in the `MainWindow.xib` window and open the disclosure triangles.

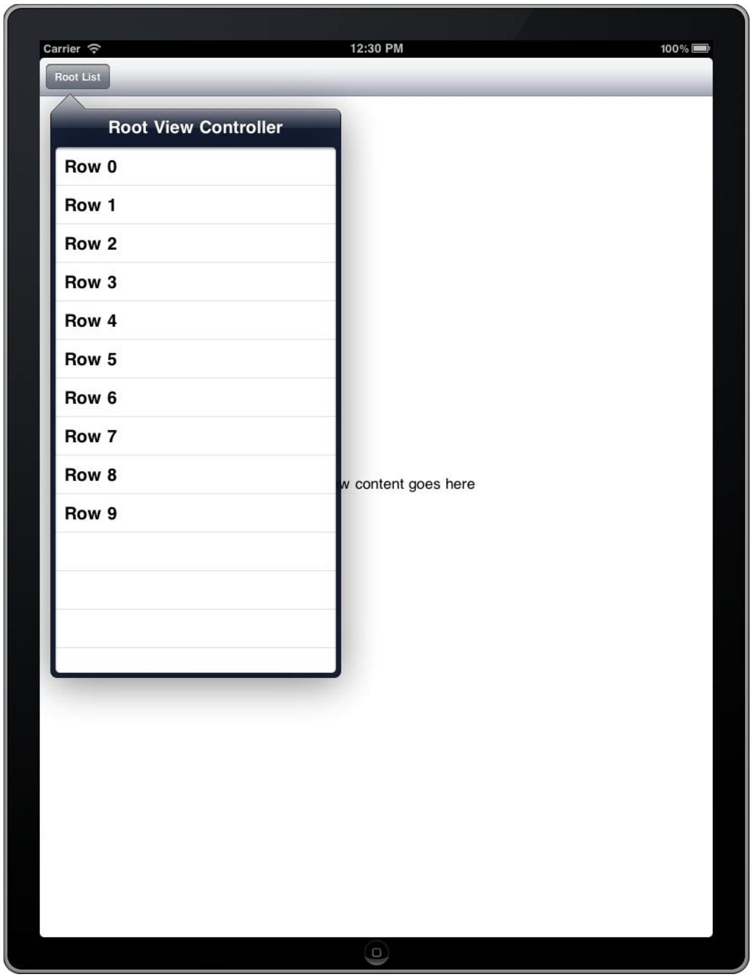


Figure 14-4: Tap the button and *voilà* — the popover appears.

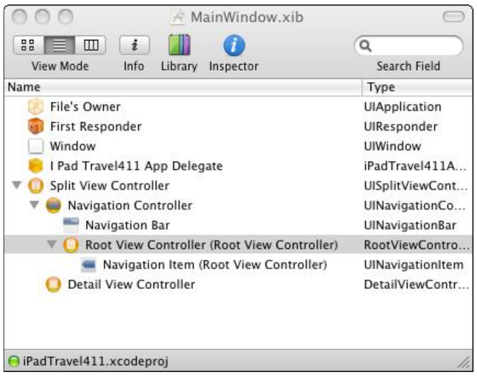
As you can see in Figure 14-5, there are *two* view controllers. The `RootViewController` (which is the Master view and the view the user will use to navigate the application) and the `DetailViewController` that will be responsible for displaying much (although not all) of the content. (You'll also see a Navigation Controller and a Navigation Bar; I explain both of those in Chapter 18.)

In the `MainWindow.xib` (see Figure 14-5), double-click the Split View Controller icon.

The window you see in Figure 14-6 makes an appearance.

Notice how `RootViewController` has found a new home in the Master view and that the view in the Detail view is `"Loaded From DetailView"`. I'd like you to put the Detail view away for now, but I promise you'll get back to it soon.

Before you get into Table views — all in good time, my friend — I want to finish this discussion by explaining how the Split view controller implements that nice popover you see earlier in Figure 14-4.



Name	Type
File's Owner	UIApplication
First Responder	UIResponder
Window	UIWindow
I Pad Travel411 App Delegate	iPadTravel411A…
▼ Split View Controller	UISplitViewCont…
▼ Navigation Controller	UINavigationCo…
Navigation Bar	UINavigationBar
▼ Root View Controller (Root View Controller)	RootViewContro…
Navigation Item (Root View Controller)	UINavigationItem
Detail View Controller	DetailViewContr…

iPadTravel411.xcodeproj

Figure 14-5: The MainWindow.xib for the iPadTravel411 app.

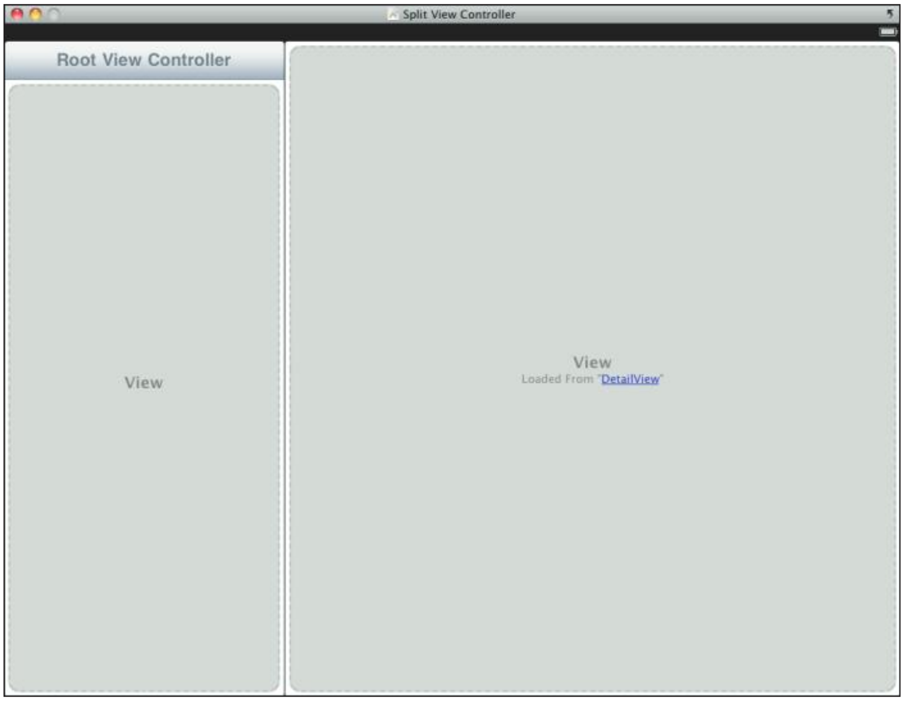


Figure 14-6: The Split view controller in all its glory.

Popovers

One of the user interface elements you'll want to add to your application is the popover. Although popovers have a number of uses, in this chapter I show you how to display that Master view you see in landscape mode in a Popover view when you switch to portrait mode.

It turns out that the Split view controller you just implemented displays both view controllers — Master and Detail — in landscape orientations, but only the Detail view controller gets displayed in portrait orientations. When the Master view controller is hidden, it is standard practice to add a button to the toolbar of the Detail view controller that the user can then use to display the Master view controller in a popover. All this is accomplished by making the Detail view controller a delegate of the Split view controller, which sends its delegate messages at the appropriate times to add and remove the button (and a few other times as well, which I'm not going to get into). If you look at the `DetailViewController` interface (`DetailViewController.h`) in your project, you can see that this whole delegation business has already been done for you:

```
@interface DetailViewController : UIViewController
       <UIPopoverControllerDelegate,
                         UISplitViewControllerDelegate> {
```

You can also see that the `DetailViewController` is a `UIPopoverControllerDelegate` — something you won't be using here, but, hey, it's still nice to know.

The `UIPopoverController` class manages the presentation of content in a popover. The content is provided in the same way you provide content in any other view — using a view controller that you provide to the popover. Popovers present information temporarily, but they don't take over the entire screen like a modal view does. This information is layered on top of your existing content in a special type of window. It remains visible until the user taps outside the popover window or you explicitly dismiss it.

REMEMBER

You also can specify the size of the popover window by assigning a value to the `contentSizeForViewInPopover` property. You should be aware that the size you specify is just the preferred size for your view. The actual size may be reduced to make the popover fit on the screen and not collide with the keyboard.

When displayed, taps outside of the popover window cause the popover to be dismissed automatically. You can, however, allow the user to interact with the specified views and not dismiss the popover, using the `passthroughViews` property (although you won't be doing that here). Taps inside the popover window do not dismiss the popover, and, as you'll see, your Master view controller will include code to dismiss the popover explicitly.

As you can see, in `DetailViewController.m`, the code in this template already does what you need to do to display the kind of popover I just explained. It does it by implementing two `UISplitViewController` delegate methods:

```
splitViewController:willHideViewController:
                  withBarButtonItem:forPopoverController:
```

and

```
splitViewController:willShowViewController:
                               invalidatingBarButtonItem:
```

The first of these methods is invoked when the Split view controller rotates from a landscape to portrait orientation and hides the Master view controller. When that happens, the Split view controller sends a message to add a button to the toolbar (or navigation bar) of the detail controller. If you look at the implementation in `DetailViewController.m`, you can find this:

```
- (void)splitViewController:(UISplitViewController *)svc
   willHideViewController:(UIViewController *)
                                    aViewController
   withBarButtonItem:(UIBarButtonItem *)barButtonItem
          forPopoverController:(UIPopoverController *)pc {

    barButtonItem.title = @"Root List";
    NSMutableArray *items = [[toolbar items] mutableCopy];
    [items insertObject:barButtonItem atIndex:0];
    [toolbar setItems:items animated:YES];
    [items release];
    self.popoverController = pc;
}
```

When this message is sent, the button that should be placed in the toolbar has been created for you and passed in as an argument. All you have to do is set the title

```
barButtonItem.title = @"Root List";
```

and then add it to the toolbar thusly

```
NSMutableArray *items = [[toolbar items] mutableCopy];
    [items insertObject:barButtonItem atIndex:0];
    [toolbar setItems:items animated:YES];
    [items release];
```

You'll notice that you first make a copy of the toolbar `items`, insert the button, and then assign the `items` property. You do it this way because (as you'll soon see) there may be other buttons on the toolbar that you'll want to maintain.

REMEMBER

Note that in your last line of code you save a reference to the Popover controller:

```
self.popoverController = pc;
```

You end up using this reference in Chapter 16 in the section on responding to a selection.

You may have noticed the catchy title, `@"Root List"`, for the button label. Although this is probably not the most user-friendly title, I'll stick with it — you should feel free to change it to whatever you'd like.

WARNING!

A word to the wise: In Chapter 16 I use `@"Root List"` for the button title. If you are going to change it here, be sure to change it in the code I outline in Chapter 16.

The second of the delegate methods is invoked when the view controller rotates from portrait to landscape orientation; it shows its hidden view controller once more. If you added the specified button to your toolbar to facilitate the display of the hidden view controller in a popover, you must implement this method and use it to remove that button. This is how it is implemented in `DetailViewController.m`:

```
- (void)splitViewController:(UISplitViewController *)svc
      willShowViewController:(UIViewController *)
           aViewController invalidatingBarButtonItem:
                         (UIBarButtonItem *)barButtonItem {

    NSMutableArray *items = [[toolbar items] mutableCopy];
    [items removeObjectAtIndex:0];
    [toolbar setItems:items animated:YES];
    [items release];
    self.popoverController = nil;
}
```

As you can see, you have simply reversed what you did earlier — you've removed the button from the toolbar and set the `self.popoverController` property to `nil`.

So there's where you'll start with your popovers — now I need to take some time to talk about the Table view, also known as the *Master* view.

Working with Table Views

Table views are front and center in several applications that come with the iPad (and iPhone) out of the box; they play a major role in many of the more complex applications you can download from the App Store. (Obvious examples: Almost all the views in the Mail, iPod, and Settings applications are Table views.) Table views not only display data, but also serve as a way to navigate a hierarchy.

If you take a look at an application such as Mail or iPod, you'll find that Table views present a scrollable list of *items* (or *rows* or *entries* — I use all three terms interchangeably) that may be divided into *sections.* A row can display text or images. So, when you select a row, you may be presented with another Table view or with some other view that may display a Web page or even some controls such as buttons and text fields. You can see an illustration of one of the things you can do in Figure 14-7. Selecting Map on the left leads to a Content view displaying a map of London and its environs — very handy after a long flight.

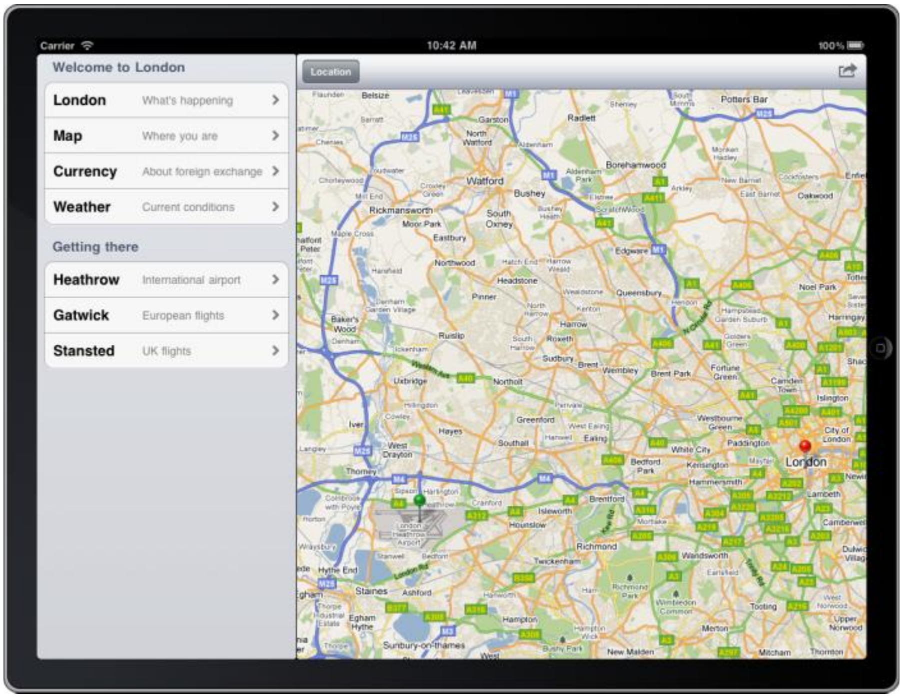


Figure 14-7: A Table view and a Map view.

But while a Table view is an instance of the class `UITableView`, each visible row of the table uses an `UITableViewCell` to draw its contents. Think of a *Table view* as the object that creates and manages the table structure, and think of the *Table-view cell* as being responsible for displaying the content of a single row of the table.

Creating the Table view

Although powerful, Table views are surprisingly easy to work with. To create a table, you need only do four — count 'em, four — things, in the following order:

1. **Create and format the view itself.**

   This includes specifying the table style and a few other parameters — most of which is done in Interface Builder.

2. **Specify the table-view configuration.**

   Not too complicated, actually. You let `UITableView` know how many sections you want, how many rows you want in each section, and what you want to call your section headers. You do that with the help of the `numberOfSectionsInTableView:` method, the `tableView:numberOfRowsInSection:` method, and the `tableView:titleForHeaderInSection:` method, respectively.

3. **Supply the text (or graphic) for each row.**

   You return that from the implementation of the `tableView:cellForRowAtIndexPath:` method. This message is sent for each visible row in the Table view, and you return a Table-view cell to display the text or graphic.

4. **Respond to a user selection of the row.**

   You use the `tableView:didSelectRowAtIndexPath:` method to take care of this task. In this method, you create a view controller and a new view. For example, when the user selects Map in Figure 14-7, this method is called, and then a Map controller and a Map view are created and displayed.

REMEMBER

A `UITableView` object must have a *data source* and a *delegate.* The data source supplies the content for the Table view, and the delegate manages the appearance and behavior of the Table view. The data source adopts the `UITableViewDataSource` protocol, and the delegate adopts the `UITableViewDelegate` protocol — no surprises there. Of the preceding methods, only the `tableView:didSelectRowAtIndexPath:` is included in the `UITableViewDelegate` protocol. All the others I list earlier are included in the `UITableViewDataSource` protocol.

The data source and the delegate are often (but not necessarily) implemented in the same object — which is often a subclass of `UITableViewController`. I plan to use the `RootViewController` for my iPhone-411Travel app.

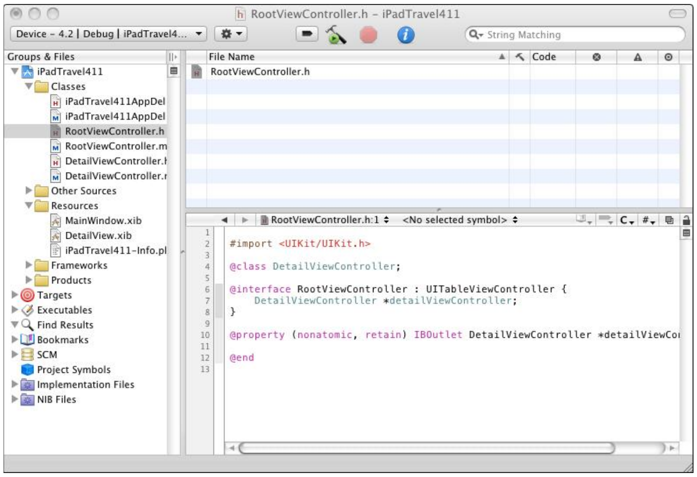
Implementing these five (count 'em, five) methods — and taking Interface Builder for a spin or two, along with the same kind of initialization methods and the standard memory-management methods you used in the DeepThoughts application — creates a Table view that can respond to a selection made in the table.

Not bad.

Creating and formatting a grouped Table view

Now let's start drilling down in your project in the Groups & Files list until you end up selecting `RootViewController.h` (as shown in Figure 14-8). The main pane of the Xcode Project window reveals the fact that `RootViewController` is derived from a `UITableViewController`.

Inquisitive type that you are, you look up `UITableViewController` in the Documentation reference by right-clicking its entry and then choosing Find Selected Text in Documentation from the pop-up menu that appears. The Class reference tells you that `UITableViewController` conforms to the `UITableViewDelegate` and `UITableViewDataSource` protocols (and a few others) — the two protocols I said were necessary to implement Table views. What luck. (Kidding. It's all intentional.)



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

@class DetailViewController;

@interface RootViewController : UITableViewController {
    DetailViewController *detailViewController;
}

@property (nonatomic, retain) IBOutlet DetailViewController *detailViewCo

@end
```

Figure 14-8: RootViewController is derived from UITableViewController.

Table views come in two basic styles. The default style is called *plain* and looks really unadorned — plain vanilla. It's a list: just one darn thing after another. You can index it, though, just as the Table view in the Contacts application is indexed, so it can be a pretty powerful tool.

The other style is the *grouped* Table view; unsurprisingly, it allows you to clump entries into various categories. In Figure 14-2, you can see the plain Table view in the Master view; Figure 14-7 shows the grouped Table view you'll be using.

TIP

Grouped tables cannot have an index.

When you configure a grouped Table view, you can also have header, footer, and section titles. (A plain view can also have section headers and footers.) I show you how to do section titles shortly.

Now go back into the nib file you were examining in the section "The Split View Controller." If you put the file away, open your project's Resources group, then click the `MainWindow.xib` file to launch Interface Builder, and in the `MainWindow.xib` (refer to Figure 14-5), double-click the Split View Controller icon.

Notice that you don't see the actual Table view yet. (For reasons beyond the scope of this book, it isn't always necessary to have it in the nib file.) If you want to change from a plain to a grouped Table view, you need to drag a Table view in from the library. After you have done that you should see what you see in Figure 14-9.

To get the final duck in a row, choose Grouped from the Style drop-down menu in the Attributes Inspector, shown in Figure 14-9, to make the switch from plain to grouped. Be sure to save the file after you do this, either by choosing File➪Save from the main menu or by using the handy ⌘+S keyboard shortcut.

At this point, you can build and run this project; go for it. What you see in the Simulator is a Table view — and if you try to scroll it, you get a "bounce scroll," where the view just bounces back up when you scroll it, but not much else. In fact, you won't even see it as a grouped view. What you do have is the basic framework, however, and now you can format it the way you like.

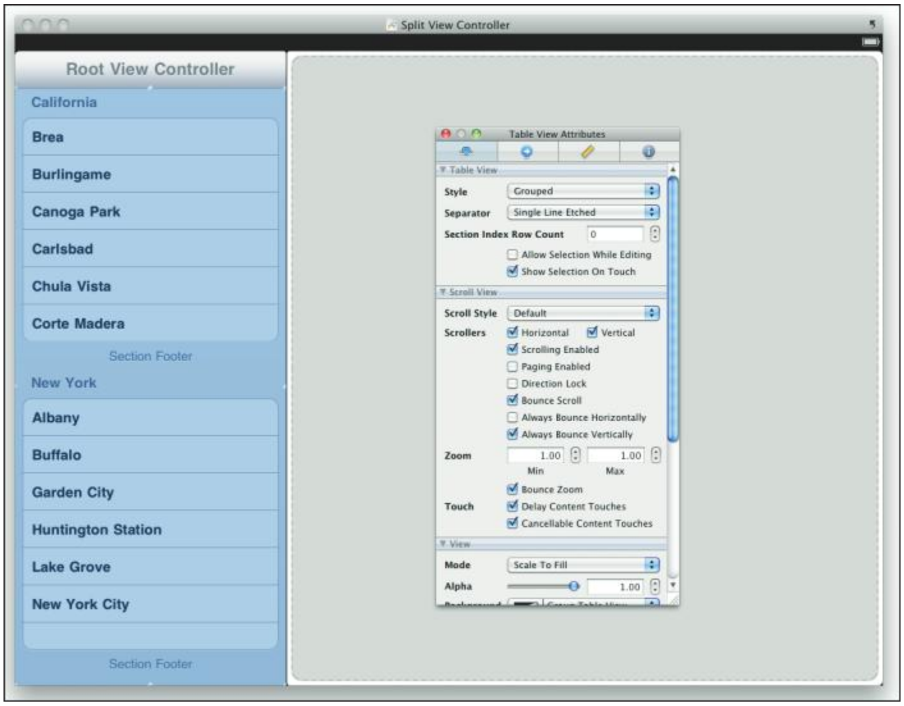


Table View
Style: Grouped
Separator: Single Line Etched
Section Index Row Count: 0
☐ Allow Selection While Editing
☑ Show Selection On Touch

Scroll View
Scroll Style: Default
Scrollers: ☑ Horizontal ☑ Vertical
☑ Scrolling Enabled
☐ Paging Enabled
☐ Direction Lock
☑ Bounce Scroll
☐ Always Bounce Horizontally
☑ Always Bounce Vertically
Zoom: 1.00 Min 1.00 Max
☑ Bounce Zoom
Touch: ☑ Delay Content Touches
☑ Cancellable Content Touches

View
Mode: Scale To Fill
Alpha: 1.00

Figure 14-9: Choosing Grouped in the Attributes Inspector.

Making UITableViewController work for you

The data source and the delegate for Table views are often (but not necessarily) the same object — and that object is frequently a custom subclass of `UITableViewController`. For the iPadTravel411 project, the `RootViewController` created by the Split View-based Application template is a subclass of `UITableViewController` — and the `UITableViewController` has adopted the `UITableViewDelegate` and `UITableViewDataSource` protocols. So you're free to implement those methods I mention in the "Creating the Table view" section, earlier in the chapter. (Just remember that you need to implement them in `RootViewController` to make your table usable.) Start with the methods that format the table the way you like.

Adding sections

In a grouped Table view, each group is referred to as a *section.*

In an indexed table, each indexed grouping of data is also called a *section*. For example, in the iPod application on the iPhone, all the albums beginning with *A* would be one section, those beginning with *B* another section, and so on. While having the same name, this is not the same thing as sections in a grouped table (which doesn't have an index).

The two methods you need to start things off are as follows:

```
numberOfSectionsInTableView:(UITableView *)tableView
```

and

```
tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
```

Each of these methods returns an integer that tells the Table view something — the number of sections and the number of rows in a given section, respectively.

In Listing 14-1, you can see the code that results in two sections with four rows in the first section and three rows in the second. These methods are already implemented for you by the Split View-based Application template in the `RootViewController.m` file. You just need to remove the existing code and replace it with what you see in Listing 14-1.

Listing 14-1: Modifying numberOfSectionsInTableView: and tableView: numberOfRowsInSection:

```
- (NSInteger)numberOfSectionsInTableView:
                                (UITableView *)tableView {

  return 2;
}

- (NSInteger)tableView:(UITableView *)tableView
              numberOfRowsInSection:(NSInteger)section {

  NSInteger rows;
  switch (section) {
    case 0:
      rows = 4;
      break;
    case 1:
      rows = 3;
      break;
    default:
      break;
  }
  return rows;
}
```

You implement `tableView:numberOfRowsInSection:` by using a simple `switch` statement:

```
switch (section) {
```

Keep in mind that the first section is zero, as is the first row.

As I mention earlier, the Table view will send the messages to its delegate. That delegate relationship was already set for you in the nib file by the template.

Although that's as easy as it gets, it's not really the best way to do it. Read on.

In the interest of showing you how to implement a robust application, I'm going to use constants to represent the number of sections *and* the number of rows in each section. I put those constants in a file, `Constants.h`, which will eventually contain other constants. I do this for purely defensive reasons: Both of these values will be used often in this application (I know that because hindsight is 20-20), and declaring them as constants makes changing the number of rows and sections easy, and it also helps avoid hard-to-detect typing mistakes.

I show you some techniques here that make life much, much easier later. It means paying attention to some of the less-glamorous application nuts-and-bolts functionalities that may be annoying to implement along the way but are *really* difficult to retrofit later. (Can you say, "memory management"?) I want to head you away from the boulder-strewn paths that so many developers have gone down (me included), much to their later sorrow.

To implement the `Constants.h` file, do the following:

1. **Choose File➪New File from the Xcode main menu.**

   I recommend having the Classes group selected in the Groups & Files list.

2. **In the New File dialog that appears, choose Other from the listing on the left (under the Mac OS X heading) and then choose Empty File in the main pane, as shown in Figure 14-10.**

3. **In the new dialog that appears, name the file `Constants.h` and then click Finish.**

   The new empty file is saved in the Classes group, as shown in Figure 14-11.

With a new home for your constants all set up and waiting, all you have to do is add the constants you need so far. (Listing 14-2 shows you the constants you need to add to the `Constants.h` file.)

Listing 14-2: Adding to the Constants.h File

```
#define kSections       2
#define kSection1Rows   4
#define kSection2Rows   3
```

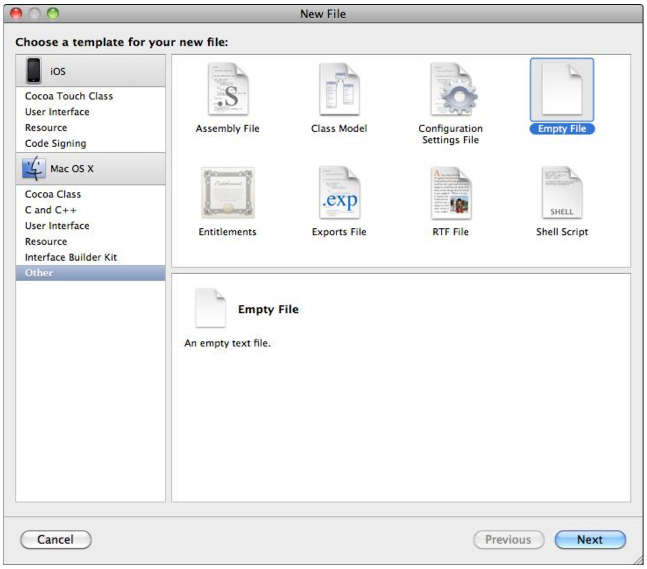


Figure 14-10: Creating an empty file.

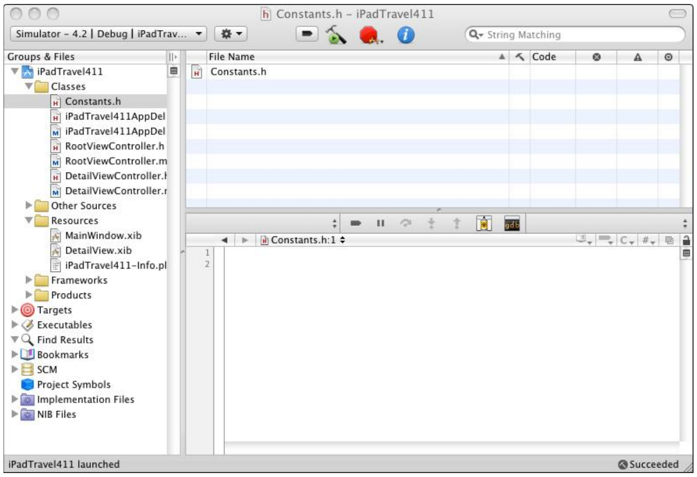


Figure 14-11: The Constants.h file.

Having a Constants.h file in hand is great, but you have to let RootViewController.m know that you plan to use it. To include Constants.h in RootViewController.m, open RootViewController.m in Xcode and add the following statement at the top of the file with the other import statements:

```
#import "Constants.h"
```

You can then use these constants in all the various methods used to create your Table view, as shown in Listing 14-3.

Listing 14-3: Sections and Rows Done Better

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)
                                               tableView {

  return kSections;
}

- (NSInteger)tableView:(UITableView *)tableView
               numberOfRowsInSection:(NSInteger)section {

  NSInteger rows;
  switch (section) {
    case 0:
      rows = kSection1Rows;
      break;
    case 1:
      rows = kSection2Rows;
      break;
    default:
      break;
  }
  return rows;
}
```

When you build and run this (provisional) app, you get what you see in Figure 14-12: two sections, the first with four rows and the second with three.

Although using constants and a switch statement does make your program more extensible, it does require you to change the switch statement if you want to add or change the layout. An even better solution is to create the array in viewDidLoad that you see in Listing 14-4. Add the code in bold to the viewDidLoad method in the RootViewController.m file and delete the bold, underlined, and italic code.

REMEMBER

I'll be asking you to do this delete business several times, so I'll be referring to code I want you delete as BUI (*b*old, *u*nderlined, *i*talic).

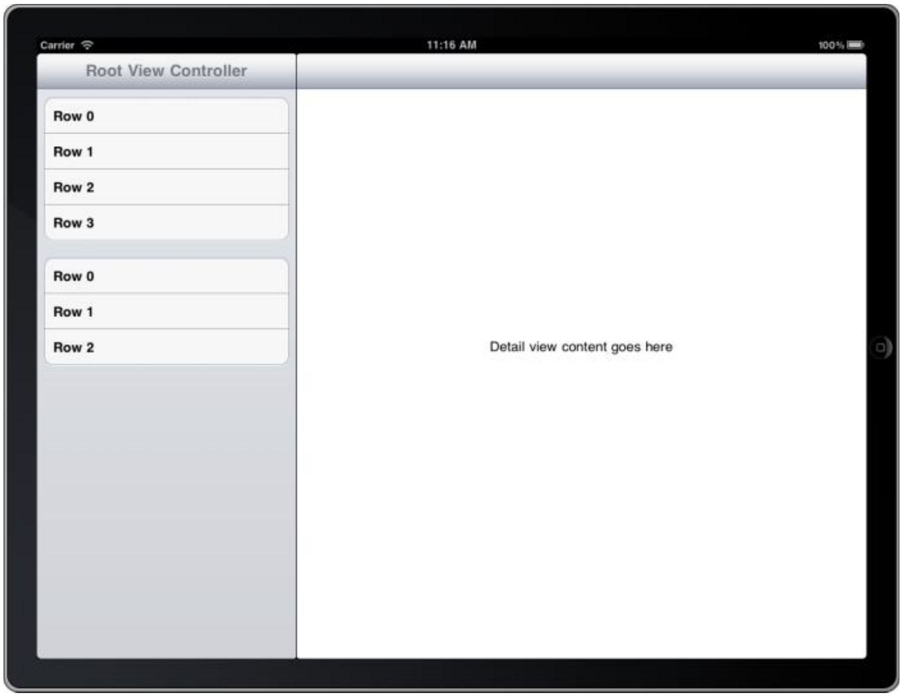


Figure 14-12: Now you have the right number of sections and rows.

Listing 14-4: viewDidLoad

```
- (void)viewDidLoad {

  [super viewDidLoad];
 self.clearsSelectionOnViewWillAppear = NO;
 self.contentSizeForViewInPopover =
       CGSizeMake(320.0 kPopoverWidth,
                               600.0 kPopoverHeight);
 sectionsArray = [[NSArray alloc] initWithObjects:
    [[NSNumber alloc]initWithInt:kSection1Rows],
    [[NSNumber alloc]initWithInt:kSection2Rows], nil];
}
```

`clearsSelectionOnViewWillAppear` indicates whether the controller should clear the selection when the table appears. (It receives a `viewWillAppear:` message.) The default value of this property is `YES`. (If you were to set this property to `NO`, it would preserve the selection.) Because a selection in our app will always result in the presentation of a new controller, it isn't necessary to bother with this line of code, so you can delete this.

You'll also notice that, with Listing 4-4, you have changed the `contentSizeForViewInPopover` in two ways. First, you deleted the hard coded values of `320` and `600` and replaced them with constants. You now have to add the constants to the `Constants.h` file.

```
#define kPopoverWidth  320
#define kPopoverHeight 700
```

As I explain in the "Popovers" section earlier in this chapter, you also can specify the size of the popover window by assigning a value to the `contentSizeForViewInPopover` property. You should be aware that the size you specify is just the preferred size for your view. The actual size may be reduced to make the popover fit on the screen and not collide with the keyboard.

REMEMBER

The size you're setting your popover to is about the size of the view in landscape mode. While you *could* make it smaller — since the view is not going to be filled all the way — I'm going to have you make it this size because, as you'll see in Chapter 18, you're going to need the extra room for the Airport view.

Of course, you could have just left things as they were, with the 320 and 700 pixels all nice and hard-coded. But instead, in the interest of showing you how to implement a robust application, I'm going to use constants for the popover width and height, just like I used constants to represent the number of sections and the number of rows in each section for the Table view. While the width is fixed, as you develop your application you'll find yourself revisiting what you'll want in the popover, and subsequently the height you'll need, and it's easier to keep that in a single place.

With all the popover sizing out of the way, go ahead and take a look at the rest of what Listing 14-4 has done for you.

Going down the line, you see that the code creates an array with two entries — one for each section. That means you can determine the number of sections by getting the number of items in the array. Then, for each array item/section, you make the value of the `NSNumber` equal to the number of rows. (You have to use the `NSNumber` object here instead of a plain `int` because the array entry must be an object.)

From there, you can use the array count `[sectionsArray count]` to return the number of sections and then use the index path section as an index into the array for the number of rows in a section `[sectionsArray objectAtIndex:section]`.

And in fact, that's what I'm going to do. Replace the `numberOfSectionsInTableView:` and `tableView:numberOfRowsInSection:` methods in `RootViewController.m` with the code in Listing 14-5 yet again (and for the last time).

To keep things on the up and up, you also need to add a new instance variable to `RootViewController.h`:

```
NSArray               *sectionsArray;
```

Listing 14-5: Final Versions of numberOfSectionsInTableView: and tableView:numberOfRowsInSection:

```
- (NSInteger)numberOfSectionsInTableView:
                               (UITableView *) tableView {

  return [sectionsArray count];
}

- (NSInteger)tableView:(UITableView *)tableView
              numberOfRowsInSection:(NSInteger)section {

  return [[sectionsArray objectAtIndex:section] intValue];
}
```

Adding titles for the sections

With sections in place, you now need to title them so users know what the sections are for. Luckily for you, the `UITableViewdataSource` protocol has a handy method — titled, appropriately enough, the `tableView:titleForHeaderInSection:` method — that enables you to add a title for each section. Listing 14-6 shows how to implement the method.

Listing 14-6: Adding Section Titles

```
- (NSString *)tableView:(UITableView *)tableView
             titleForHeaderInSection:(NSInteger)section {

  NSString *title = nil;
  switch (section) {
    case 0:
      title = @"Welcome to London";
      break;
    case 1:
      title =  @"Getting there";
      break;
    default:
      break;
  }
  return title;
}
```

This (again) is a simple `switch` statement. For `case 0`, or the first section, you want the title to be `"Welcome to London"`, and for `case 1`, or the second section, you want the title to be `"Getting there"`.

WARNING!

Okay, this, too, was really easy, so you probably won't be surprised to find that it's *not* the best way to tackle the whole titling business. It's another path not to take — in fact, a really *important* one not to take. Really Serious Application Developers insist on catering to the needs of an increasingly global audience, which means — paradoxically — that they have to *localize* their applications. In other words, an app must be created in such a way that it presents a different view to different, local audiences. The next section explains how you do that.

Localization

Localizing an application isn't difficult, just tedious. To localize your application, you create a folder in your application bundle (I'll get to that) for each language you want to support. Each folder has the application's translated resources.

The way it works is that the user will have set the language — Spanish or Italian, for example — and the region format in the Settings application.

For example, if the user's language is Spanish, available regions range from Spain to Argentina to the United States and lots of places in between. When a localized application needs to load a resource (such as an image, property list, or nib), the application checks the user's language and region and looks for a localization folder that corresponds to the selected language and region. If it finds one, it loads the localized version of the resource rather than the *base* version — the one you're working in.

Showing you all the ins and outs of localizing your application is a bit too Byzantine for this book. But I *do* show you what you must do to make your app localizable when you're ready to tackle the chore on your own.

REMEMBER

What you have to get right — right from the start — are the strings you use in your application that get presented to the user. (If the user has chosen Spanish as his or her language of choice, what's expected in the main view is now *Moneda,* not *Currency.*) You ensure that the users see what they're expecting by storing the strings you use in your application in a `strings` text file; this file contains a list of string pairs, each identified by a comment. You would create one of these files for each language you support.

Here's an example of what an entry in a `strings` file might look like for this application:

```
/*Airport choices */
"Getting there"  = "Getting there";
```

The values between the `/*` and the `*/` characters are just comments for the (human) translator you task with creating the right translation for the phrase — assuming, of course, that you're not fluent in the ten-or-so languages you'll probably want to include in your app, and therefore will need some translating help. You write such comments to provide some context — how that string is being used in the application.

Okay, this example has two strings — the one to the left of the equal sign is used as a key; the one to the right of the equal sign is the one displayed. In the example, both strings are the same — but in the `strings` file used for a Spanish speaker, here's what you'd see:

```
/*Airport choices */
"Getting there"  = "Cómo llegar";
```

Looking up such values in the table is handled by the `NSLocalizedString` macro in your code.

To show you how to use the macro, I take one of the section headings as an example. Rather than

```
title = @"Getting there";
```

I code it as follows:

```
title = NSLocalizedString(@"Getting there",
                                      @"Airport choices");
```

As you can see, the macro has two inputs. The first is the string in your language, and the second is the general comment for the translator. At runtime, `NSLocalizedString` looks for a `strings` file named `localizable.strings` in the language that has been set: Spanish, for example. (A user would have done that by going to Settings and choosing General➪International➪Language➪Español.) If `NSLocalizedString` finds the `strings` file, it searches the file for a line that matches the first parameter. In this case, it would return "Cómo llegar," and that is what would be displayed as the section header. If the macro doesn't find the file or a specified string, it returns its first parameter, and the string will appear in the base language.

To create the `localizable.strings` file, you run a command-line program named `genstrings`, which searches your code files for the macro and places them all in a `localizable.strings` file (which it creates), ready for the (human) translator. `genstrings` is beyond the scope of this book, but it's well documented. When you're ready, I leave you to explore it on your own.

Okay, sure, it's really annoying to have to do this sort of thing as you write your code (yes, I know, *really, really* annoying). But that's not nearly as annoying as having to go back and *find and replace all the strings you want to localize* after the application is almost done. Take my word for it!

Listing 14-7 shows how to use the `NSLocalizedString` macros to create localizable section titles. Add the code in Listing 14-7 to `RootViewController.h`.

Listing 14-7: Adding Localizable Section Titles

```
- (NSString *)tableView:(UITableView *)tableView
              titleForHeaderInSection:(NSInteger)section {

  NSString *title = nil;
  switch (section) {
    case 0:
      title = NSLocalizedString(@"Welcome to London",
                                               @"City name");
      break;
    case 1:
      title = NSLocalizedString(@"Getting there",
                                         @"Airport choices");
      break;
    default:
      break;
  }
  return title;
}
```

Creating the Row Model

As all good iPhone and iPad app developers know, the Model-View-Controller (MVC) design pattern is the basis for the design of the framework you use to develop your applications. In this design pattern, each element (model, view, or controller) concentrates on the task at hand; it doesn't much care what the other elements are doing. For Table views, that means the method that draws the content doesn't know what the content is, and the method that decides what to do when a selection is made in a particular row is equally ignorant of what the selection is. The important thing is to have a model object — one for each row — to hold and provide that information.

In this kind of situation, you usually want to deal with the model-object business by creating an array of models, one for each row. In this case, the model object will be a dictionary that holds the following three items:

- **The selection text:** Map, for example
- **The description text:** Where you are, for example
- **The view controller to be created when the user selects that row:** `MapController`, for example

You can see all three items illustrated in Figure 14-13.

REMEMBER

In more complex applications, you could provide a dictionary *within* the dictionary and use it to provide the same kind of information for the next level in the hierarchy. The iPod application is an example: It presents you with a list of albums, and then when you select an album, it shows you a list of songs on that album.

The following code shows you how to create a single dictionary for a row. Later on, I show you how to create all the dictionaries and tell you where all this code needs to go.

```
menuList = [[NSMutableArray alloc] init];

[menuList addObject:[NSMutableDictionary
     dictionaryWithObjectsAndKeys:
     NSLocalizedString(@"Map", @"Map Section"),kSelectKey,
     NSLocalizedString(@"Where you are", @"Map Explain"),
                                             kDescriptKey,
     [NSNull null], kControllerKey, nil]];
```

Dictionary

Key	Value
kSelectKey	Map
kDescriptKey	Where are you
kControllerKey	"MapContoller"

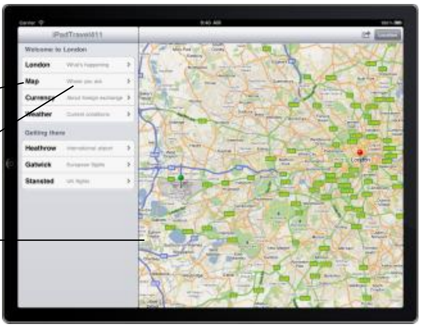


Figure 14-13: The model for a row.

Here's the blow-by-blow account:

1. **Create an array to hold the model for each row.**

   An `NSMutableArray` is a good choice here because it allows you to easily insert and delete objects.

In such an array, the position of the dictionary corresponds to the row it implements, that is, relative to row zero in the table and not taking into account the section.

2. **Create an `NSMutableDictionary` with three entries and the following keys:**
   - `kSelectKey`: The entry that corresponds to the main entry in the Table view (`"Map"`, for example).
   - `kDescriptKey`: The entry that corresponds to the description in the Table view (`"Where you are"`, for example).
   - `kControllerKey`: This entry contains a pointer to a view controller that will display the map. You're going to create an entry for the controller, but not just yet; you just use an `NSNull` object as a placeholder for now. ("Objects" in an array have to be objects.) The first time the user selects a row, you create the view controller and save that value in here. That way, if the user selects that row again, the controller will simply be reused.
3. **Add the keys to the `Constants.h` file.**

```
#define kSelectKey       @"selection"
#define kDescriptKey     @"description"
#define kControllerKey   @"viewController"
```

The `@` before each of the preceding strings tells the compiler that this is an `NSString`.

With all this information, you're now in a position to get rid of these controllers if you were to ever get a low-memory warning. You'd simply go through each dictionary in the array and `release` every controller except the one that's currently active.

You'll want to create this array and all the dictionaries in an initialization method `viewDidLoad`, which you added to the `RootViewController.m` file in the "Making UITableViewController work for you" section. (Refer to Listing 14-4.) Add the code in bold in Listing 14-8 to `viewDidLoad`. The `viewDidLoad` message is sent to the `RootViewController` after all the objects in the nib file have been loaded and the `RootViewController`'s outlet instance variables have been set.

You could argue that you really should create a model class that creates this data-model array and get its data from a file or property list. For simplicity's sake, you add it in the `viewDidLoad` method for the iPadTravel411 app.

Listing 14-8: viewDidLoad

```
- (void)viewDidLoad {

  [super viewDidLoad];
  self.contentSizeForViewInPopover =
       CGSizeMake(kPopoverWidth, kPopoverHeight);
  sectionsArray = [[NSArray alloc] initWithObjects:
       [[NSNumber alloc]initWithInt: kSection1Rows],
       [[NSNumber alloc]initWithInt: kSection2Rows], nil];

  self.title = [[[NSBundle mainBundle] infoDictionary]
                         objectForKey:@"CFBundleName"];
  menuList = [[NSMutableArray alloc] init];
  [menuList addObject:[NSMutableDictionary
                            dictionaryWithObjectsAndKeys:
     NSLocalizedString(@"London", @"City Section"),
                                              kSelectKey,
     NSLocalizedString(@"What's happening",
                         @"City Explain"), kDescriptKey,
     [NSNull null], kControllerKey, nil]];
  [menuList addObject:[NSMutableDictionary
                            dictionaryWithObjectsAndKeys:
     NSLocalizedString(@"Map", @"Map Section"),
                                              kSelectKey,
     NSLocalizedString(@"Where you are",
                          @"Map Explain"), kDescriptKey,
     [NSNull null], kControllerKey, nil]];
  [menuList addObject:[NSMutableDictionary
                            dictionaryWithObjectsAndKeys:
     NSLocalizedString(@"Currency", @"Currency Section"),
                                              kSelectKey,
     NSLocalizedString(@"About foreign exchange",
                      @"Currency Explain"), kDescriptKey,
     [NSNull null], kControllerKey, nil]];
  [menuList addObject:[NSMutableDictionary
                            dictionaryWithObjectsAndKeys:
     NSLocalizedString(@"Weather", @"Weather Section"),
                                              kSelectKey,
     NSLocalizedString(@"Current conditions",
                      @"Weather  Explain"), kDescriptKey,
     [NSNull null], kControllerKey, nil]];
  [menuList addObject:[NSMutableDictionary
                            dictionaryWithObjectsAndKeys:
     NSLocalizedString(@"Heathrow", @"Heathrow Section"),
                                              kSelectKey,
     NSLocalizedString(@"International airport",
                      @"Heathrow Explain"), kDescriptKey,
     [NSNull null], kControllerKey, nil]];
  [menuList addObject:[NSMutableDictionary
                            dictionaryWithObjectsAndKeys:
```

```
      NSLocalizedString(@"Gatwick", @"Gatwick Section"),
                                                kSelectKey,
      NSLocalizedString(@"European flights",
                            @"Gatwick Explain"), kDescriptKey,
      [NSNull null], kControllerKey, nil]];
   [menuList addObject:[NSMutableDictionary
                                 dictionaryWithObjectsAndKeys:
      NSLocalizedString(@"Stansted",
                             @"Stansted Section"), kSelectKey,
      NSLocalizedString(@"UK flights",
                            @"Stansted Explain"), kDescriptKey,
      [NSNull null], kControllerKey, nil]];
}
```

You also have to add the following to `RootViewController.h`:

```
NSMutableArray *menuList;
```

Going through the code in Listing 14-8, you can see that the first thing you do is get the application name from the bundle so you can use it as the Main view title.

```
self.title = [[[NSBundle mainBundle] infoDictionary]
                            objectForKey:@"CFBundleName"];
```

"What bundle?" you ask. Well, when you build your iPad application, Xcode packages it as a bundle containing the following:

- The application's executable code
- Any resources that the app has to use (for instance, the application icon, other images, and localized content)
- The `info.plist`, also known as the information property list, which defines key values for the application, such as bundle ID, version number, and display name

`infoDictionary` returns a dictionary that's constructed from the bundle's `info.plist`. `CFBundleName` is the key to the entry that contains the (localizable) application name on the home page. The title is what will be displayed in the Navigation bar at the top of the screen.

As I mention earlier, I also create `sectionsArray`, which I can use to compute the offset in the menu. I save a reference to that array in the instance variable.

Going through the rest of the code, you can see that for each entry in the Main view, you have to create a dictionary and put it in the `menuList` array. You put the dictionary in the `menuList` array so you can use it later when you need to provide the row's content or create a view controller when the user selects the row. Because there's no controller yet, you create an `NSNull` object that simply acts as a placeholder. The `NSNull` class defines a singleton object, with a single class method `null`, that you can use to represent null values in collection objects.

Seeing How Table-View Cells Work

I've been going steadily from macro to micro, so it makes sense that after setting up a model for each row, I get to talk about cells, the individual constituents of each row.

Cell objects are what draw the contents of a row in a Table view. The method `tableView:cellForRowAtIndexPath:` is called for each visible row in the Table view. It's expected that the method will configure and return a `UITableViewCell` object for each row. The `UITableView` object uses this cell to draw the row.

When providing cells for the Table view, you have three general approaches you can take:

- Use vanilla (not subclassed) `UITableViewCell` cell objects.
- Add subviews to a `UITableViewCell` cell object's Content view.
- Use cell objects created from a custom subclass of `UITableViewCell`.

The next few sections take a look at these options, one by one.

Using vanilla cell objects

Using the `UITableViewCell` class directly, you can create cell objects with text and an optional image. (If a cell has no image, the text starts near the left edge of the cell.) You also have an area on the right of the cell for accessory views, such as disclosure indicators (the one shaped like a regular chevron), detail disclosure controls (the one that looks like a white chevron in a blue button), and even control objects such as sliders, switches, or custom views. (The layout of a cell is shown in Figure 14-14.) If you like, you can format the font, alignment, and color of the text, as well as have a different format when the row is selected.

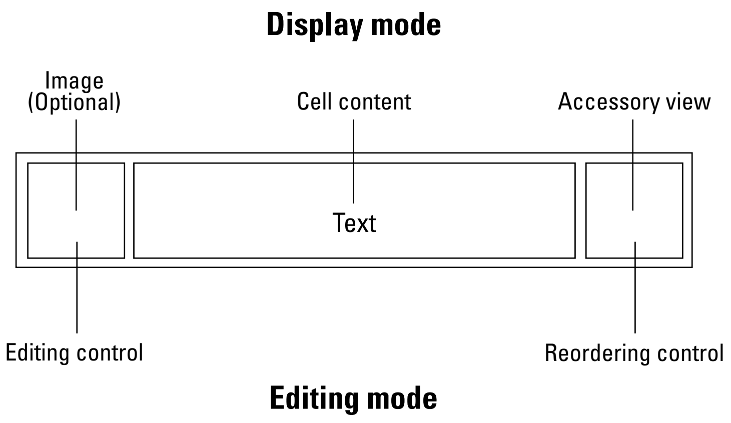


Figure 14-14: The cell architecture.

Adding subviews to a cell's Content view

Although you can specify the font, color, size, alignment, and other characteristics of the text in a cell by using the `UITableViewCell` class directly, the formatting is applied to all the text in the cell. To get the variation that I suspect you want between the selection and description text (and, it turns out, the alignment as well), you have to create subviews within the cell.

A cell that a Table view uses for displaying a row is, in reality, a view in its own right. `UITableViewCell` inherits from `UIView`, and it has a Content view. With Content views, you can add one subview (containing, say, the selection text "Weather") formatted the way you want, and you can add a second subview (holding, say, the description text, "Current conditions") formatted an entirely different way. You may remember that you already experienced adding subviews (the slider, text field, and labels) in creating a "preferences" view in the DeepThoughts application, although you may not have known you were doing that at the time. Well, now it can be told.

Creating a custom subclass UITableViewCell

Finally, you can create your very own custom cell subclass when your content requires it — usually when you need to change the default behavior of the cell or if you want to use a nib file to lay out a more complex view.

Creating the Cell

As I mention in the previous section, you're going to use the `UITableViewCell` class to create the cells for your Table views and then add the subviews you need in order to get the formatting you want.

The place to create the cell is `tableView:cellForRowAtIndexPath:`. This method is called for each visible row in the Table view, as shown in Listing 14-9. (Replace the method the template supplies with Listing 14-9 in `RootViewController.m`.)

Listing 14-9: Drawing the Text

```
- (UITableViewCell *)tableView:(UITableView * )tableView
          cellForRowAtIndexPath:(NSIndexPath *)indexPath {

  UILabel *selectLabel;
  UILabel *descriptLabel;

  UITableViewCell *cell = [tableView
       dequeueReusableCellWithIdentifier:kCellIdentifier];
  if (cell == nil) {
    cell = [[[UITableViewCell alloc]
      initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:kCellIdentifier] autorelease];

    cell.accessoryType =
              UITableViewCellAccessoryDisclosureIndicator;

    CGRect subViewFrame = cell.contentView.frame;
    subViewFrame.origin.x += kInset;
    subViewFrame.size.width = kInset+kSelectLabelWidth;

    selectLabel = [[UILabel alloc]
                          initWithFrame:subViewFrame];
    selectLabel.tag = kSelectLabelTag;
    selectLabel.textColor = [UIColor blackColor];
    selectLabel.highlightedTextColor =
                                 [UIColor whiteColor];
    selectLabel.font = [UIFont boldSystemFontOfSize:18];
    selectLabel.backgroundColor = [UIColor clearColor];
    [cell.contentView addSubview:selectLabel];

    subViewFrame.origin.x += kInset+kSelectLabelWidth;
    subViewFrame.size.width = kDescriptLabelWidth;

    descriptLabel = [[UILabel alloc]
                          initWithFrame:subViewFrame];
    descriptLabel.tag = kDescriptLabelTag;
    descriptLabel.textColor = [UIColor grayColor];
    descriptLabel.highlightedTextColor =
                                 [UIColor whiteColor];
    descriptLabel.font = [UIFont systemFontOfSize:14];
    descriptLabel.backgroundColor = [UIColor clearColor];
    [cell.contentView addSubview:descriptLabel];
}
  else {
    selectLabel = (UILabel *)[cell.contentView
                                 viewWithTag:kSelectLabelTag];
    descriptLabel = (UILabel *)[cell.contentView
                               viewWithTag:kDescriptLabelTag];
  }
  int menuOffset =
             [self menuOffsetForRowAtIndexPath:indexPath];
  NSDictionary *cellText =
                    [menuList objectAtIndex:menuOffset];
  selectLabel.text = [cellText objectForKey:kSelectKey];
  descriptLabel.text =
                   [cellText objectForKey:kDescriptKey];
  [selectLabel release];
  [descriptLabel release];
  return cell;
}
```

Here's the logic behind all that code:

1. **Determine whether there are any cells lying around that you can use.**

   Although a Table view can display only a limited number of rows at a time on the iPad's screen, the table itself can conceivably hold a lot more. A large table would chew up a lot of memory if you were to create cells for every row. Fortunately, Table views are designed to *reuse* cells. As a Table view's cells scroll off the screen, they're placed in a queue of cells available to be reused.

   You can ask the Table view for a specific reusable cell object by sending it a `dequeueReusableCellWithIdentifier:` message:

   ```
   UITableViewCell *cell = [tableView
      dequeueReusableCellWithIdentifier:kCellIdentifier];
   ```

   This asks whether any cells of the type you want are available.

2. **Create a *cell identifier* that indicates what cell type you're using. Add this to the `Constants.h` file:**

   ```
   #define kCellIdentifier  @"MasterViewCell"
   ```

   Table views support multiple cell types, which makes the identifier necessary. In this case, you need only one cell type, but sometimes you may want more than one.

   If the system runs low on memory, the Table view gets rid of the cells in the queue, but as long as it has some available memory for them, it will hold on to them in case you want to use them again.

3. **If there aren't any cells lying around, you have to create a cell by using the cell identifier you just created.**

```
   if (cell == nil) {
     cell = [[[UITableViewCell alloc]
         initWithStyle:UITableViewCellStyleDefault
         reuseIdentifier:kCellIdentifier] autorelease];
```

   You now have a Table view cell that you can return to the Table view.

   `UITableViewCellStyleDefault` gives you a simple cell with a text label (black and left-aligned) and an optional image view. There are also several other styles:

   • `UITableViewCellStyleValue1` gives you a cell with a left-aligned black text label on the left side of the cell and a smaller blue text and right-aligned label on the right side. (The Settings application uses this style of cell.)

   • `UITableViewCellStyleValue2` gives you a cell with a right-aligned blue text label on the left side of the cell and a left-aligned black label on the right side of the cell.

   • `UITableViewCellStyleSubtitle` gives you a cell with a left-aligned label across the top and a left-aligned label below it in smaller gray text. (The iPod application uses cells in this style.)

4. **Define the accessory type for the cell.**

```
   cell.accessoryType =
          UITableViewCellAccessoryDisclosureIndicator;
```

   As I mention earlier in the brief tour of a cell, its layout includes a place for an accessory — usually something like a disclosure indicator.

   TIP

   In this case, use `UITableViewCellAccessoryDisclosureIndicator` (the one shaped like a regular chevron). It lets the user know that tapping this entry will result in something (hopefully wonderful) happening, such as the display of the current weather conditions.

   If you're using a Table view, and you want to display more detailed information about the entry itself, you might use a Detail Disclosure button. This allows you to then use a tap on the row for something else. In the Favorites view in the iPhone application, for example, selecting the Detail Disclosure button gives you a view of the contact information; if you just tap the row, it places the call for you.

   You're not limited to these kinds of indicators; you also have the option of creating your own view — you can put in any kind of control. (That's what you see in the Settings application, for example.)

5. **Create the subviews.**

   Here I show you just one example. (The other is the same except for the font size and text color.) You get the `contentView` frame and base the subview on it. The inset from the left (`kInset`) and the width of the subview (`kLabelWidth`) are defined in the `Constants.h` file — you'll need to add them. They look like this:

```
#define kInset               10
#define kSelectLabelWidth    100
#define kDescriptLabelWidth  160
```

   To hold the text, the subview you're creating is a `UILabelView`, which meets your needs exactly:

```
CGRect subViewFrame = cell.contentView.frame;
subViewFrame.origin.x += kInset;
subViewFrame.size.width = kInset+kSelectLabelWidth;
selectLabel =
         [[UILabel alloc] initWithFrame:subViewFrame];
```

   Next you assign a tag to the `selectLabel`.

```
selectLabel.tag = kSelectLabelTag;
```

   A tag is an identifier of a view; it allows you to locate a view in its view hierarchy by calling the `viewWithTag:` method. You need to do that because if you get the designated cell from the Table view's queue, as you'll soon see, you'll use the tags to obtain references to the two subviews so that you can assign them the right text. You also need to add the two tags you'll need to `Constants.h`:

```
#define kSelectLabelTag   1
#define kDescriptLabelTag 2
```

   You then set the label properties that you're interested in; you do it by manually writing code rather than using Interface Builder. Just set the font color and size, the highlighted font color when an item is selected, and the background color of the label (as indicated in the code that follows). Setting the background color to transparent allows me to see the bottom line of the last cell in the group.

```
selectLabel.textColor = [UIColor blackColor];
selectLabel.highlightedTextColor = [UIColor
                                         whiteColor];
selectLabel.font = [UIFont boldSystemFontOfSize:18];
selectLabel.backgroundColor = [UIColor clearColor];
[cell.contentView addSubview:selectLabel];
```

   TECHNICAL STUFF

   I could have inset the view one pixel up from the bottom, made the label opaque, and given it a white (not clear) background, which would be more efficient to draw. But with such a small number of rows, making that effort really has no appreciable performance impact, and the way

I've set it up here requires less code for you to go through. Feel free to do it the "right way" on your own.

6. **If you do have a cell you can reuse, get the `selectLabel` and `descriptLabel` views using the tags you used when you created them.**

```
selectLabel = (UILabel *)
       [cell.contentView viewWithTag:kSelectLabelTag];
descriptLabel = (UILabel *)
     [cell.contentView viewWithTag:kDescriptLabelTag];
```

7. **Based on the row and section, you offset into the array of dictionaries you created earlier to find the right dictionary and then use the dictionary to assign the text to the labels.**

```
int menuOffset =
         [self menuOffsetForRowAtIndexPath:indexPath];
NSDictionary *cellText =
                  [menuList objectAtIndex:menuOffset];
selectLabel.text = [cellText objectForKey:kSelectKey];
descriptLabel.text =
                 [cellText objectForKey:kDescriptKey];
```

   The trouble is that you won't get the absolute row passed to you. You get only the row within a particular section — and you need the absolute row to get the right dictionary from the array. Fortunately, one of the arguments used when this method is called is the `indexPath`, which contains the section and row information in a single object. To get the row or the section out of an `NSIndexPath`, you just have to invoke its section method (`indexPath.section`) or its row method (`indexPath.row`), both of which return an `int`. This neat trick enables you to compute the offset for the row in the array you created in `viewDidLoad`.

   You added the method — `menuOffsetForRowAtIndexPath:` — so that you can use it again later to get the menu offset, as you'll see.

The method and its algorithm are shown in Listing 14-10. Add the method to `RootViewController.m` and its declaration to `RootViewController.h`.

Listing 14-10: Computing the Menu Offset

```
- (int) menuOffsetForRowAtIndexPath:
                          (NSIndexPath *)indexPath {

  int menuOffset = indexPath.row;
  for (int sectionRow=0; sectionRow <
                 indexPath.section; ++sectionRow) {
    menuOffset += [[sectionsArray
                objectAtIndex:sectionRow] intValue];
  }
  return menuOffset;
}
```

You start by passing the row to `menuOffset`. You then increment the offset by the number of rows in each of the previous sections, which is stored in the array as an `NSNumber`. (Because `NSNumber` is an object, you have to use the `intValue` method to get the number as an integer.)

You also have to add the declaration of `menuOffsetForRowAtIndexPath:` to `RootViewController.h`.

Finally, because you no longer need the labels you created, you release them

```
[selectLabel release];
[descriptLabel release];
```

and return the formatted cell with the text it needs to display in that row.

```
return cell;
```

It's time for some final housekeeping. At this point you've implemented the table of contents, but of course nothing else. If this were an iPhone app, that would be good enough, because this view would entirely fill the screen. But on the iPad you have all that space.

So one of the things you'll want to decide is what the user should see at launch. Now, under multitasking, when the user leaves your app and then returns, most likely the user will see where she left off. But in two instances you'll need to make a decision of what the user will see. You'll need to decide what the user will see when the application is first launched and you'll also need to decide what the user will see if the application is purged from memory.

While the choice is up to you in your apps, personally I like maps, so that's what you'll explore next. But because you're now getting into content, it's time to explain how you'll implement the Model piece of the Model-View-Controller design pattern.

The Destination Model

As I explain in Chapters 7 and 13, the basic architecture for iPadTravel411 (and any other iPad program) is the Model-View-Controller. In this chapter, you have created a View-Controller structure within a Split view controller using a Table view as the Master view.

As you recall from Chapters 7 and 13, the model "owns" the data — and you'll start by creating a main model class called `Destination`. I refer to it as the main model class because, as you'll see, `Destination` will have plenty of help from other objects, but for the time being I want you to focus on it alone.

1. **Choose File➪New File from the main menu (or press ⌘+N) to recall the New File dialog.**

   It's a good idea to add a new group in the Groups & Files list to hold all your new model classes. To do so, select the iPadTravel411 Project icon and then choose Project➪New Group. You'll get a brand-spanking-new group, named New Group, already selected and waiting for you to type in the name you want. (I named mine Model classes.) To change what group a file is in, select and then drag the file to the group you want it to occupy. The same goes for groups as well. (After all, they can go into other groups.)

2. **In the leftmost column of the dialog, select Cocoa Touch Classes under the iPhone OS heading just like you did before, but this time select the Objective-C class template in the topmost pane, making sure that the Subclass drop-down menu has NSObject selected. Then click Next.**

   You see a new dialog asking for some more information.

3. **Enter** Destination **in the File Name field and then click Finish.**

You need to add some functionality to the `Destination` class to get you started. I'll start with an initialization method that simply saves a Destination name in an instance variable. Add the code in Listing 14-11 to `Destination.m`.

Listing 14-11: Initializing Destination

```
- (id) initWithName:(NSString *) theDestination {

  if ((self = [super init])) {
    destinationName = theDestination;
  }
  return self;
}
```

The first thing this method does is send the `init` message to its superclass — the message to `super` precedes the initialization code added in the method. This sequencing ensures that initialization proceeds in the order of inheritance. Calling the superclass's `init` method initializes the controller, loads and initializes the objects in the nib file (views and controls, for example), and then sets all its outlet instance variables and Target-Action connections for good measure.

TECHNICAL STUFF

The `init…:` methods all return a pointer to the object created. Although not the case here, the reason you assign whatever comes back from an `init…:` method to `self` is that some classes actually return a different class than what you created. The assignment to `self` becomes important if your class is derived from one of those kinds of classes. Keep in mind as well that an `init...:` method can also return `nil` if there's a problem initializing an object. If you're creating an object where that is a possibility, you have to take that into account. (Both of those situations are beyond the scope of this book.)

After the superclass initialization is completed, the `Destination` is ready to do its own initialization, including saving the `theDestination` argument to the `destinationName` instance variable.

Add the code in bold in Listing 14-12 to the `Destination.h` interface file to support your additions.

Listing 14-12: The Destination interface

```
@interface Destination : NSObject {

  NSString  *destinationName;
}
- (id) initWithName:(NSString *) theDestination;

@end
```

Of course, the next thing you'll need to do is create the `Destination` object and make it accessible to the view controllers. I'm going to have you do that in a method you'll add to `iPadTravel411AppDelegate.m` called `awakeFromNib`.

The `awakeFromNib` message is sent by the nib-loading infrastructure to each object created from the nib file. It is sent after all the objects have been loaded and initialized — which means that all outlet and action connections for an object have been set. By the way, you do have to be sure to send the `awakeFromNib` message to your superclass to give it an opportunity to do its initialization.

This is a handy method to know about because this is the place you may need to do some initialization type tasks before the action starts. I'm having you do it here because, as you'll soon find out, the `RootViewController` is going to need to know about the `Destination` object to do its work.

Add the code in Listing 14-13 to `iPadTravel411AppDelegate.m`.

Listing 14-13: awakeFromNib

```
- (void)awakeFromNib {

  [super awakeFromNib];
  destination =
             [[Destination alloc] initWithName: @"London"];
}
```

Of course, you'll have to do some housekeeping here as well.

Add the code in bold to `iPadTravel411AppDelegate.h` to declare the instance variable and make it a property.

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

@class RootViewController;
@class DetailViewController;
@class Destination;
@interface iPadTravel411AppDelegate : NSObject
           <UIApplicationDelegate> {

  UIWindow *window;

  UISplitViewController *splitViewController;

  RootViewController    *rootViewController;
  DetailViewController  *detailViewController;
  Destination           *destination;
}

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

@property (nonatomic, retain) IBOutlet
           UISplitViewController *splitViewController;
@property (nonatomic, retain)
     IBOutlet RootViewController *rootViewController;
@property (nonatomic, retain) IBOutlet
           DetailViewController *detailViewController;
@property (nonatomic, retain) Destination *destination;

@end
```

Add the import and synthesize statements to `iPadTravel411AppDelegate.m`.

```
#import "iPadTravel411AppDelegate.h"
#import "RootViewController.h"
#import "DetailViewController.h"
```

```
#import "Destination.h"

@implementation iPadTravel411AppDelegate

@synthesize window, splitViewController,
           rootViewController, detailViewController;
@synthesize destination;
```

Now is also a good time to make it accessible to `RootViewController`. How should you do that? Because it's a property, all the `RootViewController` has to do is access it from `iPadTravel411AppDelegate`. But how does it find the `iPadTravel411AppDelegate`?

It turns out that this is done so often that there's a really easy way to do it. All you do is send a message to the `UIApplication` and ask for the delegate:

```
[[UIApplication sharedApplication] delegate]
```

`UIApplication` is a singleton object (there is only one) and the class method `delegate` returns the `delegate` object. But frankly, I find myself doing this so often in my program that, in order to save typing, I simply add a constant to `Constants.h` (and you should to):

```
#define kAppDelegate ((iPadTravel411AppDelegate *)
             [[UIApplication sharedApplication] delegate])
```

I also cast the `delegate` object that's returned to an `iPadTravel411AppDelegate`. You have to do that if you want to access instance variables and methods you have added to your app delegate.

Now is also a good time to add the `import` statements that you'll need to access the `iPadTravel411AppDelegate` and the `Destination` class to the top of `RootViewController.m`. Check out the following bolded lines:

```
#import "RootViewController.h"
#import "DetailViewController.h"
#import "Constants.h"
#import "iPadTravel411AppDelegate.h"
#import "Destination.h"
```

So while you haven't used these statements yet, you're all set for the next chapter.

Expanding the Architecture to a "Real" App

The way I've had you construct the model, the user gets to go to only one place. No matter how interesting and exciting London is, there are probably some other places in the world you'd also like to travel to (Katmandu comes to mind).

What I could have done was add a `Trip` class as well. That way, the user would see a list of destinations in some sort of Table view first (London, Paris, Katmandu) and then be able to choose one — which would then display the Table view you created at the beginning of this chapter.

This does, however, make this app much more complicated, so I haven't done that — simplicity of presentation being my main goal here. But it's definitely an exercise for you to pursue on your own.

Chapter 15
Finding Your Way

In This Chapter

- Using the Map framework
- Specifying the location and zoom level of a map
- Identifying the iPad's current location
- Annotating significant locations on the map

One of the things that makes iPad applications compelling is the ability you have as a developer to incorporate the user's location into the application functionality. And one of the more compelling ways to do that is through the use of maps.

Being able to build maps into your application is an important new feature in the iOS 3.0 SDK and beyond, and it doesn't hurt that working with maps is one of the funnest things you can do on the iPad because Apple makes it so easy.

You've heard me say it before, and I'll say it again: The iPad is about content. For an app bearing the name iPadTravel411, there's no better place to start dealing with content than with a map. In this chapter, I introduce you to `MKMapView` and have you create a `UIViewController` subclass (`MapController`) to manage the map display.

Putting Content First

The iPad is about delivering content, and that's what you'll want your user to experience when he launches your app — *real* content, rather than some sort of screen that enables you to navigate the app that you'd normally get in a mobile device with a smaller screen.

Although this book doesn't go on and on about what *kind* of content you should be providing in your app, you should know that the SDK does make it really easy to deliver certain types of content — in this case, maps. In this chapter, I take you on a tour of `MKMapKit` and show you how to create really useful annotated maps that contain the information the user needs. (*Annotated* in this context means those cute pins in the map that display a callout to describe that location when you touch them.)

Because the goal of iPadTravel411 is to reduce the hassles involved with traveling, it should come as no surprise that one thing you can do right off the bat is present the users with a handy map that should be able to help them in whatever situation they're in. In Figure 15-1, you can see the iPad displaying a map that includes the airport (in this case, Heathrow) and the user's destination (London).

Including the ability to display a map in iPadTravel411 became important as people begin to realize the kinds of solutions that can be delivered on the iPad. To many travelers, nothing brands you more as a tourist than unfolding a large map (except of course looking through a thick guidebook). In this chapter, I show you how to take advantage of the iPad's built-in capability to display a map of virtually anywhere in the world, as well as determine your location and then indicate it in the map. As I mention earlier, the iPad's awareness of your location is one of the things that enables you to develop a totally new kind of application that really sets an iPad application apart from a desktop one.

Oh, and by the way, it turns out that working with maps is one of the most fun things you can do on the iPad because Apple makes it so easy. In fact, you can display a map that supports those great panning and zooming gestures you find on all the Big Boy apps by simply creating a view controller and a nib file. (You do that soon.)

In this chapter, I show you how to center your map on an area you want to display (London and Heathrow airport), add annotations that display a callout to describe that location when you touch them, drag an annotation to someplace else on the map, and even show the user's current location. And because your users (and Apple) will insist that they be able to use the iPad in any orientation, I show you how to make sure Heathrow Airport and London are both visible on the map no matter what orientation the user chooses.

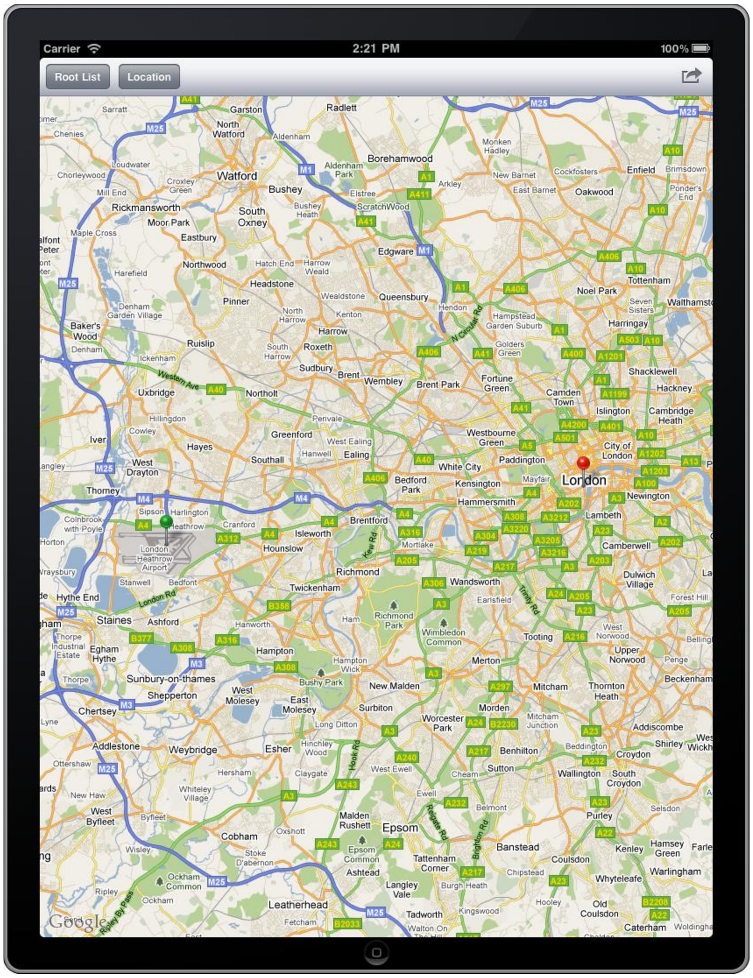


Figure 15-1: Where's Waldo?

Adding the Map Controller

To use maps, you have to add a few more files to your project: a `MapController.h` and `.m` and a `MapController` nib file.

1. **Add a new group in the Groups & Files list to hold all your new View controllers classes.** To do so, select the iPadTravel411 Project icon and then choose Project➪New Group. You'll get a brand-spanking-new group, named New Group, already selected and waiting for you to type in the name you want; type **View controllers**. (To change what group a file is in, select and then drag the file to the group you want it to occupy. The same goes for groups as well.)
2. **In the IPadTravel411Project window, select the View controllers group and then choose File➪New File from the main menu (or press ⌘+N) to call up the New File dialog.**
3. **In the leftmost pane of the dialog, first select Cocoa Touch Classes under the iOS heading, then select the UIViewController Subclass template in the topmost pane, and then make sure that the following are selected:**
    - With XIB for User Interface
    - Targeted for iPad

    You see a new dialog asking for some more information.
4. **Enter** MapController.m **in the File Name field and then click Finish.**

Implementing the MapController

To enable the `MapController` to do what you need it to do, there's no time like the present to add some functionality to it.

To start with, add the code in bold in Listing 15-1 to `MapController.h`, then delete the code in bold, underline, and italic (BUI).

Listing 15-1: Starting the MapController Interface

```
#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
#import "DetailViewController.h"
@class Destination;

@interface MapController : UIViewController
               DetailViewController <MKMapViewDelegate> {

  Destination          *destination;
  IBOutlet MKMapView   *mapView;

}
- (id) initWithDestination:(Destination *) theDestination;

@end
```

Here's what you're doing here.

1. **Make the `MapController` a subclass of the `DetailViewController` and a `MKMapViewDelegate` by deleting the code with a strikethrough (`UIViewController`, in other words) and adding the code in bold shown in this step to the `MapView.h` file.**

   ```
   @interface MapController : ~~UIViewController~~
                 DetailViewController <MKMapViewDelegate>
            {
   ```

   The reason you're making the `MapController` a subclass of the `DetailViewController` is because the `DetailViewController` already has all the code you'll need to manage the toolbar when you move from landscape to portrait (add the button to display the popover controller) and from portrait to landscape (remove the button).

   You make the `MapController` a `MKMapViewDelegate` because there are some methods in the delegate that you will need to implement later. (I explain that when you get there.)

2. **Add the `import` statements to enable the compiler to know who's who.**

   The one I want to draw your attention to is `#import "DetailViewController.h"`. For the compiler to be able to derive the `MapController` from the `DetailViewController`, it has to know the details of the `DetailViewController`.

   `#import <MapKit/MapKit.h>` gives you access to all the lovely functionality that came your way when you added the `MapKit.framework` in the preceding section.

3. **Add the instance variables `destination` and `mapView` because the `MapController` will need to use them — `mapView` to set some map properties and `destination` to access the annotations. (I explain that in the "Adding Annotations" section, later in this chapter.)**

   Although you get a default map for free (look ahead to Figure 15-6), which is all well and good, there's a lot more that you can do with it. For that to happen, though, you're going to need to be able to access the Map view. To do that, make `mapView` an outlet by using the keyword `IBOutlet` in the `mapView` declaration.

4. **Add an initialization method `initWithDestination:` to assign the `destination` argument to the instance variable.**

   You could have had the `MapController` access the app delegate's instance variable, but you do it this way to allow for more flexibility later. (I might want to have maps showing not only the current `destination` in the `iPadAppDelegate` instance variable, but others as well.)

5. **Save the file by choosing File➪Save.**

   Only after it's saved can Interface Builder find the new outlet.

Now you need to add the code in bold in Listing 15-2 to `mapController.m`.

Listing 15-2: Initializing the MapController

```
#import "MapController.h"
#import "Destination.h"

@implementation MapController

- (id) initWithDestination:
                          (Destination *) theDestination {

  if (self =
     [super initWithNibName:@"MapController"bundle:nil]) {
    destination = theDestination;
  }
  return self;
}

@end
```

The first thing this method does is invoke its superclass's initialization method. You pass it the nib filename (the one you just created in the preceding section) and `nil` as the bundle, telling it to look in the main bundle.

```
[super initWithNibName:@"MapController" bundle:nil]) {
```

Note that the message to `super` precedes the initialization code added in the method. This sequencing ensures that initialization proceeds in the order of inheritance. Calling the superclass's `initWithNibName:bundle:` method initializes the controller, loads and initializes the objects in the nib file (views and controls, for example), and then sets all its outlet instance variables and Target-Action connections for good measure.

The `init…:` methods all return a pointer to the object created. While not the case here, the reason you assign whatever comes back from an `init…:` method to `self` is that some classes actually return a different class than what you created. The assignment to `self` becomes important if your class is derived from one of those kinds of classes. Keep in mind as well that an `init…:` method can also return `nil` if there's a problem initializing an object. If you're creating an object where that's a possibility, you have to take that into account. (Both of those situations are beyond the scope of this book.)

After the superclass initialization is completed, the `MapController` is ready to do its own initialization, including saving the `theDestination` argument to the `destination` instance variable.

Be sure to save the file to be able to access the new outlet in Interface Builder.

Cleaning up the DetailViewController

You can delete a number of methods from `DetailViewController` because you won't need them.

Make the modifications you see in Listing 15-3 to `DetailViewController.h`, and make the modifications in Listing 15-4 to `DetailViewController.m`. The additions are in bold, and deletions are in BUI.

Listing 15-3: Cleaning Up the DetailViewController Interface

```
@interface DetailViewController : UIViewController
          <UIPopoverControllerDelegate,
                           UISplitViewControllerDelegate>
          {

  UIPopoverController *popoverController;
  UIToolbar *toolbar;

id detailItem;
UILabel *detailDescriptionLabel;
}

@property (nonatomic, retain) IBOutlet UIToolbar *toolbar;
@property (nonatomic, retain) UIPopoverController
                                      *popoverController;
@property (nonatomic, retain) id detailItem;
@property (nonatomic, retain) IBOutlet UILabel
                                 *detailDescriptionLabel;
@end
```

Listing 15-4: Cleaning Up the DetailViewController Implementation

```
#import "DetailViewController.h"
#import "RootViewController.h"

@interface DetailViewController ()
@property (nonatomic, retain) 
                 IPopoverController *popoverController;
- (void)configureView;
@end

@synthesize toolbar, popoverController; , detailItem, 
                                 detailDescriptionLabel;


#pragma mark -
#pragma mark Managing the detail item

/*
 When setting the detail item, update the view and dismiss
           the popover controller if it's showing.
*/
- (void)setDetailItem:(id)newDetailItem {

  if (detailItem != newDetailItem) {
    [detailItem release];
    detailItem = [newDetailItem retain];

      // Update the view.
    [self configureView];
  }

  if (popoverController != nil) {
      [popoverController dismissPopoverAnimated:YES];
  }
}


- (void)configureView {
// Update the user interface for the detail item.
  detailDescriptionLabel.text = [detailItem description];
}

...



(void)dealloc {
  [popoverController release];
  [toolbar release];
  [detailItem release];
  [detailDescriptionLabel release];
  [super dealloc];
}
```

One thing I want to call your attention to is the following deletion:

```
@interface DetailViewController ()
@property (nonatomic, retain)
                UIPopoverController *popoverController;
- (void)configureView;
@end
```

Because it's currently implemented as a class extension, `popover Controller` is not visible to subclasses. So you had to move the `popover Controller` property from where it is in the `DetailViewController` implementation into the interface to be able to access it in your derived classes later. (I explain class extensions in Chapter 18.) `configureView` is deleted because you deleted the method.

Finally, you also have to delete the reference in the `RootViewController` to the `detailItem`. In `RootviewController.m`, delete the line in `table View:didSelectRowAtIndexPath:`, as I have in Listing 15-5.

Listing 15-5: Doing the Last of the Cleanup

```
- (void)tableView:(UITableView *)aTableView didSelectRowAt
          IndexPath:(NSIndexPath *)indexPath {

    /*
     When a row is selected, set the detail view
          controller's detail item to the item associated
          with the selected row.
     */
    detailViewController.detailItem =
      [NSString stringWithFormat:@"Row %d", indexPath.row];
}
```

You'll implement this method for your app shortly.

Finally, finally — and I mean it this time — you should delete `DetailView. xib` from your project — you won't need it any longer because you no longer instantiate the `DetailViewController` (and its nib). Instead you'll instantiate View controllers and their nibs whose classes are *derived* from the `DetailViewController`. (You'll do that in the "Creating the MapController" section, later in this chapter.)

To delete DetailView.xib, simply select the file in your project's Groups & Files list and press Delete. When the dialog appears, Select Also Move to Trash.

There is no undo for this Delete action in Xcode, so be careful.

Adding the framework

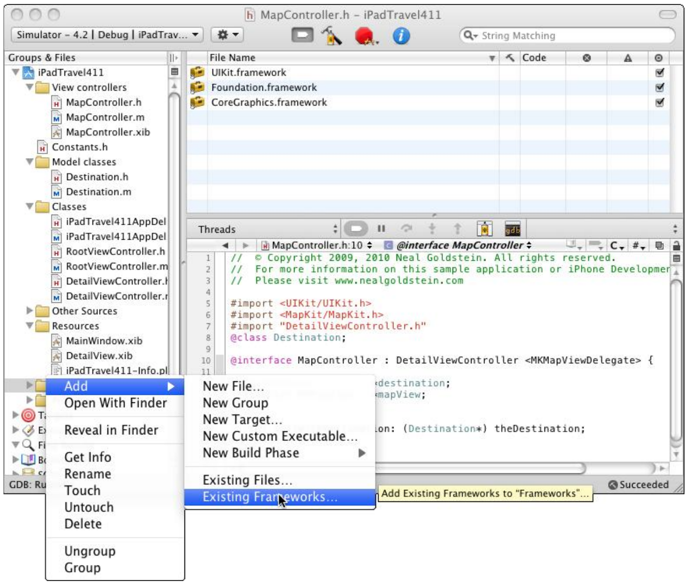
Okay, that takes care of most of what you will have to do in your MapController.m and .h files. The next thing to do: You have to add a new framework.

Up until now, all you've needed is the framework that more or less came supplied when you created a project. But now, you need to add a new framework to enable the Map view. (Officially it is a MKMapView, but I refer to it as simply a Map view.)

1. **Make sure the disclosure triangle next to Frameworks in the Groups & Files list is pointing *down* (opened) and then right-click Frameworks.**

   Be sure to do this using the Frameworks group.

2. **From the submenu that appears, select Add and then select Existing Frameworks, as I've done in Figure 15-2.**



```
//  © Copyright 2009, 2010 Neal Goldstein. All rights reserved.
//  For more information on this sample application or iPhone Developmer
//  Please visit www.nealgoldstein.com

#import <UIKit/UIKit.h>
#import <MapKit/MapKit.h>
#import "DetailViewController.h"
@class Destination;

@interface MapController : DetailViewController <MKMapViewDelegate> {
```

Figure 15-2: Adding a new framework.

3. **In the new window that appears (see Figure 15-3), select MapKit.framework and then click Add.**

   Your new framework is placed into the Frameworks group.

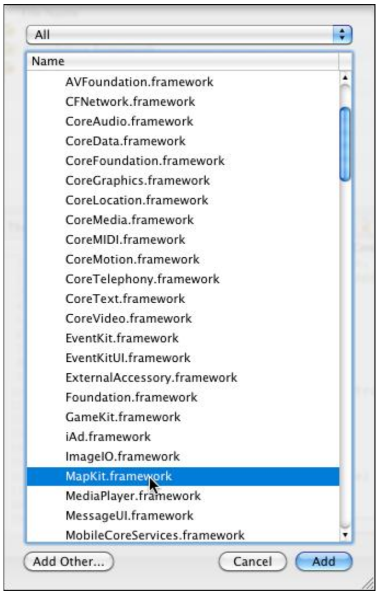


Figure 15-3: Adding the MapKit. framework.

Setting up the nib file

To display a map, I show you how to use an `MKMapView` to display the map information. To set up the `MKMapView`, you use Interface Builder.

1. **Use the Groups & Files list on the left in the Project window to drill down to the `MapController.xib` file; then double-click the file to launch it in Interface Builder.**

   TIP

   If the Attributes Inspector window is not open, choose Tools➪Inspector or press ⌘+Shift+1. If the View window isn't visible, double-click the View icon in the `MapController.xib` window.

   If for some reason you can't find the `MapController.xib` window (you may have minimized it, whether by accident, on purpose, or whatever), you can get it back by choosing Window➪MapController.xib or whichever nib file you're working on.

2. **Select File's Owner in the `MapController.xib` window.**

   Its type should already be set to `MapController`. If not, retrace your steps to see where you may have made a mistake.

   You need to be sure that the File's Owner is `MapController`. You can set File's Owner from the Class drop-down menu in the Identity Inspector.

3. **Drag in a toolbar from the Library to the top of the View window. Make sure it is a toolbar and not a navigation bar. (It looks like it belongs on the bottom of the view, but on the iPad it can be at the top.) Select and delete the button that says "item" in the toolbar — you won't need it.**

4. **In the View Attributes inspector window for the view, be sure to deselect Autoresize Subviews, as shown in Figure 15-4.**

5. **Drag in a Map View from the Library below the toolbar and resize it to the size of the remaining view — the View window minus the toolbar. When you're done, it should look like Figure 15-4.**

   Note that in Figure 15-4. I have the view selected and the View Attributes Inspector window is showing.

6. **Back in the `MapController.xib` window, right-click File's Owner to call up a connections panel with a list of connections.**

   You can get the same list using the Connections tab in the Attributes Inspector.

7. **Drag from the little circle next to the `mapView` outlet in the list onto the Map view.**

   Doing so connects the `mapView` outlet for the `MapController` to the Map view.

8. **Drag from the little circle next to the `toolbar` outlet in the list onto the toolbar.**

   Doing so connects the `toolbar` outlet for the `MapController` to the toolbar.

9. **Go back to that list of connections in the File's Owner connections panel menu and click the disclosure triangle (if necessary) next to Referencing Outlets. This time drag from the little circle next to the New Referencing Outlet list onto the Map view.**

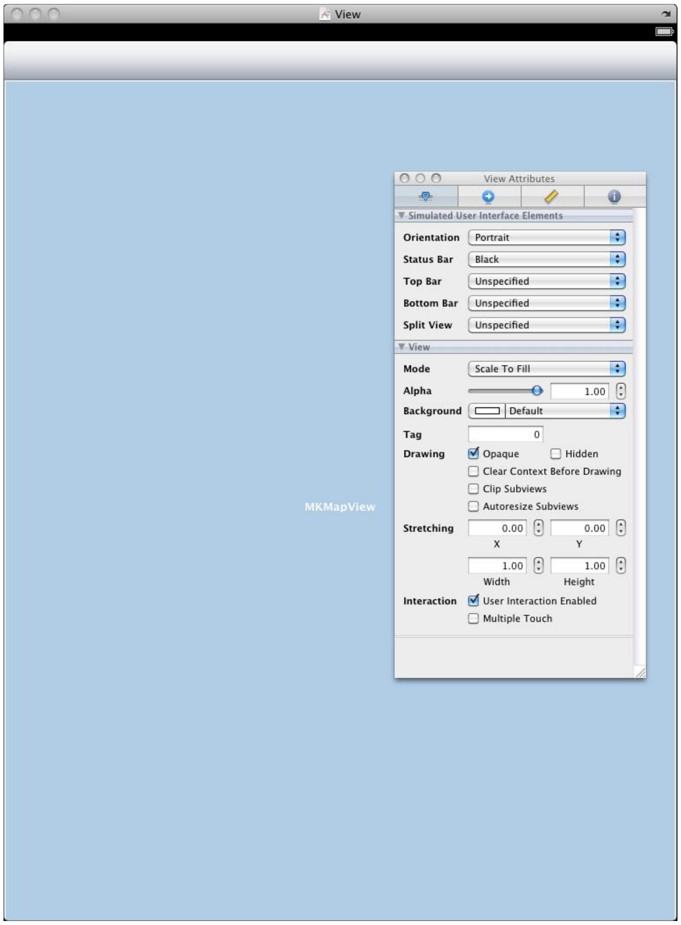
10. **With the cursor still in the Map view, let go of the mouse button.**

    A pop-up menu appears.

11. **Choose Delegate from the pop-up menu.**

    When you're done, the contextual menu should look like Figure 15-5.

12. **Save the file.**



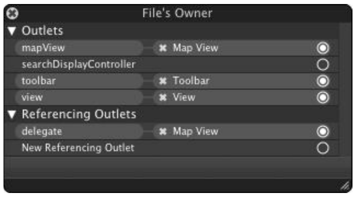
▼ Simulated User Interface Elements

Orientation	Portrait
Status Bar	Black
Top Bar	Unspecified
Bottom Bar	Unspecified
Split View	Unspecified

▼ View

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

Figure 15-4: The Interface Builder windows.


File's Owner

▼ Outlets

mapView	Map View
searchDisplayController	
toolbar	Toolbar
view	View

▼ Referencing Outlets

delegate	Map View
New Referencing Outlet	

Figure 15-5: The connections are all made.

The `MapController` now has its outlet to the Map view and toolbar connected, and it also will receive the delegate messages as a `MKMapViewDelegate`. I show you what methods you need to implement next.

Only after the file's saved will the changes you made be reflected in your application.

Creating the MapController

As it stands now, if you were to compile and run the app, you'd get the same old thing as you did in the last chapter — except for the label text you just removed. To see a Map view, you're going to have to replace the old Detail view controller with your brand-new Map view controller.

It all starts in `RootViewController.m`. At the end of the `RootViewController.m` file's `viewDidLoad` method, add the code in bold in Listing 15-6. You're going to replace the `DetailViewController` with the `MapController`.

Listing 15-6: Adding to viewDidLoad

```
- (void)viewDidLoad {
  [super viewDidLoad];
…

  DetailViewController *mapController =
      [[MapController alloc]
            initWithDestination:kAppDelegate.destination];
  [[kAppDelegate splitViewController] setViewControllers:
    [NSArray arrayWithObjects:self.navigationController,
                                      mapController, nil]];
  kAppDelegate.splitViewController.delegate =
          (<UISplitViewControllerDelegate>)mapController;
}
```

As I explain in Chapter 14, the `UISplitViewController` class is a view controller that simply manages the presentation of two side-by-side view controllers — it is, in this respect, a container controller. Using this class, you create a view controller on the left (the Master view), which presents a list of items, and one on the right, which presents the details, or content, of the selected item (the Detail view).

After you create and initialize a `UISplitViewController` object, you assign two view controllers to it by using the `viewController` property (an array of two controllers). The Split view controller, as I explain in Chapter 14, has no interface — its job is to coordinate the presentation of its two view controllers and to manage the transitions between different orientations.

And this is exactly what you do here. You create a new view controller — `mapController` — and then create the array consisting of the existing master controller — `(self.navigationController)` and the newly created Detail (view) controller — `mapController`. You assign your new array to the controllers property using the setViewControllers: access method.

```
 [[kAppDelegate splitViewController] setViewControllers:
   [NSArray arrayWithObjects:self.navigationController,
                                       mapController, nil]];
```

You may be curious why you're using self.navigationController and not something that resembles the RootViewController here (like self). Answering that question involves understanding navigation using view controllers, and this is not the right place to do that. So be patient for now; you learn all about navigation controllers in Chapter 18.

You also assign the mapController as the delegate to the Split view controller — that will ensure it gets those handy messages when the view rotates (a process I explain in Chapter 14).

```
kAppDelegate.splitViewController.delegate = mapController;
```

In case you didn't notice, you declared the mapController as a DetailViewController. Doing it this way allows you to access all the functionality out there supporting popovers that is already part of the DetailViewController. This isn't necessary here, but as you'll see, you're going to need that functionality later when you have a whole slew of these controllers, all derived from DetailViewController.

Finally, as you probably guessed already, you're also going to have to add the following import statement to RootViewController.m:

```
#import "MapController.h"
```

One further note: shouldAutorotateToInterfaceOrientation: is a method that returns a YES or NO depending on whether your app supports a device orientation; it's implemented in the DetailViewController. Because one of the requirements for an iPad app is that it (usually) should support any orientation, this method by default returns YES. It's also implemented in all the UIViewController files you'll create, meaning you need to delete it because you're going to be adding some behavior to the method in DetailViewController in the next section.

So, do the dirty deed and delete the shouldAutorotateToInterface Orientation: method in MapController.m:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
            (UIInterfaceOrientation)interfaceOrientation {
    // Overriden to allow any orientation.
    return YES;
}
```

If you were to build and run your program at this point, you'd get the default Map view you see in Figure 15-6.

You'll have all the right toolbars and a nice little button that says Root List that displays the Table view you created in Chapter 14. You're able to rotate the device just as you could when using the base Detail view controller and to have the button disappear in landscape mode.

But you — and your users — want and deserve more than that. Figure 15-7 shows what you might like to see rather than the standard Map view you get right out of the box. But before I get to that, we need to have a heart-to-heart discussion about managing the view sizes in this environment.

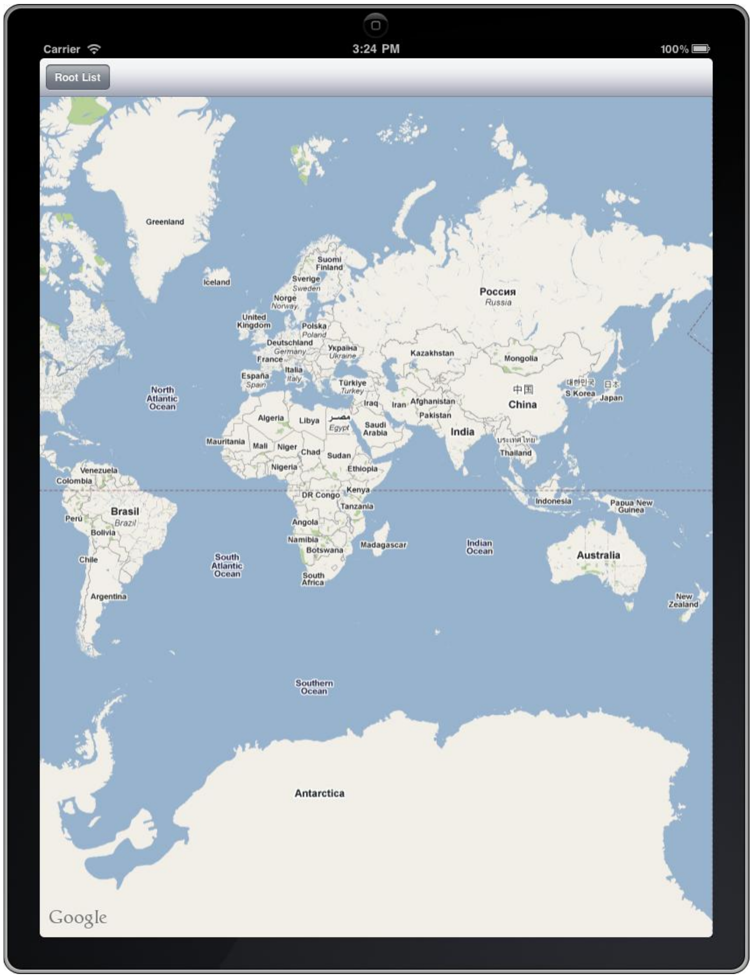


Figure 15-6: The default Map view.

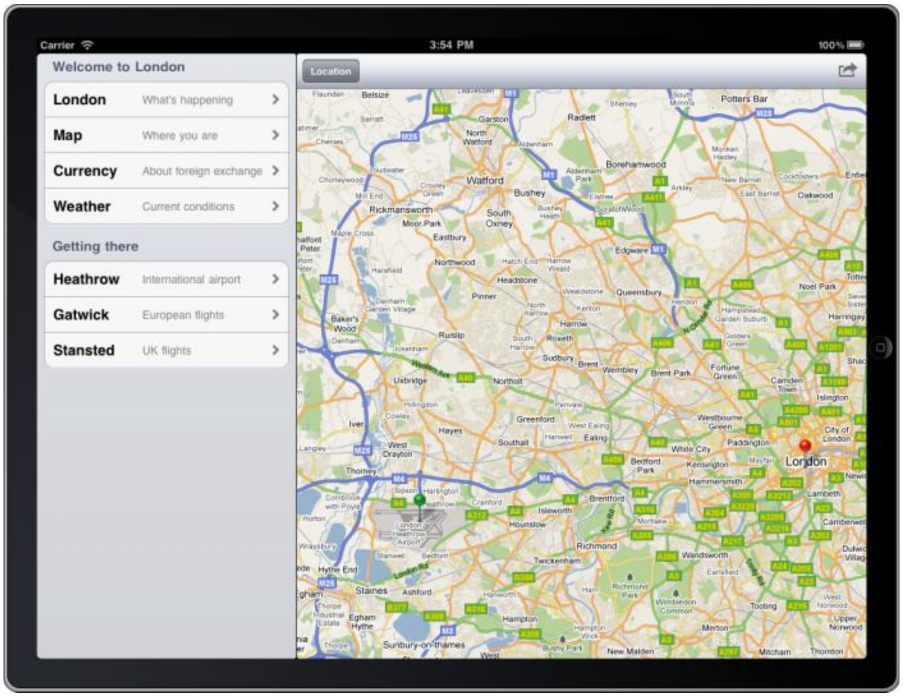


Figure 15-7: Heathrow and London on the map.

Managing the views

Although it's perfectly all right to replace view controllers in a Split controller view, it does come at a price. For reasons beyond the scope of this book, you're going to have to resize your views depending on the device orientation. Now don't get your knickers in a twist; it's not hard, and by the time I'm done explaining it to you, you'll have extended your understanding of view geometry considerably — all in all, a good thing. You'll have an appreciation of how views and nibs work, and you'll be able to modify your code to meet your needs in any application that isn't a boring-off-the-shelf-created-from-a-template one.

The problem arises from the fact that in your nib file you (implicitly) specify a size for the view and toolbar when you drag in the views and toolbar. You created your view in portrait orientation (as opposed to landscape) and the default here is to have the view controller resize the views when you move into landscape orientation.

Although this works fine in the template, when you start substituting your own view controllers, the view controller's default behavior for resizing and placing views when you move the device from portrait to landscape orientation doesn't work the way you need it to. This is especially true with the Map view and (generally speaking) whenever you add content to the Master view. (Chapter 18 looks at this issue in a bit more detail.)

Now it can be revealed: This is why I had you deselect the Autoresize Subviews option when you were working with the `MapController.xib` file in the “Setting up the nib file” section, earlier in this chapter.

The thing is that if you’re not going to have the view controller resize the views for you, someone has to — and that someone is you.

Listing 15-7 shows you how to compute the right size for the view and toolbars. You should add the code in Listing 15-7 to the `DetailViewController.m` file. That way the method will be inherited by all the `DetailViewController` subclasses you’ll be creating.

Listing 15-7: Computing the View and Toolbar Sizes

```
- (void) computeFrames:(UIView *) aView forOrientation:
          (UIInterfaceOrientation)interfaceOrientation {

   CGRect screenBounds = [[UIScreen mainScreen] bounds];
   CGRect viewFrame = aView.frame;
   CGRect toolbarFrame = toolbar.frame;
   if
    (UIDeviceOrientationIsPortrait(interfaceOrientation)) {
     viewFrame.size.height = screenBounds.size.height -
               toolbarFrame.size.height - kStatusBarHeight;
     viewFrame.size.width = screenBounds.size.width;
     toolbarFrame.size.width =  screenBounds.size.width;
   }
   else {
    viewFrame.size.height = screenBounds.size.width -
               toolbarFrame.size.height - kStatusBarHeight;
    viewFrame.size.width =
               screenBounds.size.height - kMasterViewWidth;
    toolbarFrame.size.width =
               screenBounds.size.height - kMasterViewWidth;
   }
   toolbar.frame = toolbarFrame;
   aView.frame = viewFrame;
}
```

To start with, you get the bounds of the screen. From this you can get the height and width, which currently on the iPad is 1,024 x 768 — but it’s better not to count on it. Then you get the frames for the view (in this case, the Map view) and the toolbar, respectively.

```
CGRect screenBounds = [[UIScreen mainScreen] bounds];
CGRect viewFrame = aView.frame;
CGRect toolbarFrame = toolbar.frame;
```

To refresh your memory, the screenBounds, viewFrame, and toolbar Frame are all CGRects which describe the view's location and size in its superview's coordinate system. It is made up of two data structures: origin and size.

```
struct CGRect {
   CGPoint origin;
   CGSize size;
};
```

Which are defined as

```
struct CGSize {
   CGFloat width;
   CGFloat height;
};
struct CGPoint {
   CGFloat x;
   CGFloat y;
};
```

Next in line in Listing 15-7, you check to see whether the device is in portrait mode.

```
if (UIDeviceOrientationIsPortrait(interfaceOrientation) {
```

UIDeviceOrientationIsPortrait is a handy UIKit function that returns a Boolean value indicating whether the device is in a portrait orientation. You're using the interfaceOrientation argument that was passed in with the method — you'll see where that comes from next.

If the app is in portrait mode, you compute the view height to be the screen height minus the status bar height minus the toolbar height. You assign the view and toolbar width as the screen width.

```
viewFrame.size.height =  screenBounds.size.height -
              toolbarFrame.size.height - kStatusBarHeight;
viewFrame.size.width = screenBounds.size.width;
toolbarFrame.size.width =  screenBounds.size.width;
```

You'll also have to add the following constants to Constants.h — both of these sizes are specified by the SDK:

```
#define kMasterViewWidth 320
#define kStatusBarHeight 20
```

You also have to add #import "Constants.h" to DetailViewController.m.

If the app is in landscape mode, you do something similar, but because the screen bounds are based on portrait orientation no matter what the actual device orientation, you're going to base the height computation, the screen width, and the width computation on the screen height.

```
viewFrame.size.height =  screenBounds.size.width-
              toolbarFrame.size.height - kStatusBarHeight;
viewFrame.size.width =
              screenBounds.size.height - kMasterViewWidth;
toolbarFrame.size.width =
              screenBounds.size.height - kMasterViewWidth;
```

See? That wasn't so hard, and now you know something a few developers with apps in the store don't even know.

Now that you have implemented the `computeFrames:` method, it's time to look at how and when the `computeFrames:` message gets sent.

REMEMBER

You'll want to send this message whenever the device orientation changes. One place you definitely want to do that is in `shouldAutorotateToInterfaceOrientation:`, the message that is sent to the view controller whenever the user moves the device from one orientation to another (portrait to landscape for example).

Add the code in bold in Listing 15-8 to `shouldAutorotateToInterfaceOrientation:` in `DetailViewController.m` and add its declaration to `DetailViewController.h`.

Listing 15-8: Adding to shouldAutorotateToInterfaceOrientation:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
          (UIInterfaceOrientation)interfaceOrientation {

  [self computeFramesForOrientation:interfaceOrientation];
  return YES;
}
```

You probably see a bit of a disconnect here. The message you need to send to compute the right view and toolbar sizes is `computeFrames:forOrientation:`. The message you're sending is `computeFramesForOrientation:`. (The former takes two arguments, and the latter just one.) What gives?

The problem is that you need to provide `computeFrames:forOrientation:` with the view it needs to adjust, and that is only known in the subclasses.

To do that, you need to add a variation on Listing 15-9 to each of your subclasses, replacing `mapView` with the right view. (Don't worry; I explain exactly what to do in each instance.) Add the method in Listing 15-9 to `MapController.m`.

Listing 15-9: Passing in the Right View

```
- (void) computeFramesForOrientation:
            (UIInterfaceOrientation)interfaceOrientation {

   [self computeFrames:mapView
                    forOrientation:interfaceOrientation];
}
```

You also have to add the same method as an empty method in `DetailViewController.m` for it to be able to send that message to itself. One of the rules of subclassing is that the superclass has no knowledge of any methods of its subclass — makes sense. So you'll need to extend the `DetailViewController` class to include this message, which will then be inherited and overridden in each subclass. Add the code in Listing 15-10 to `DetailViewController.m`.

Listing 15-10: The Empty Method

```
- (void) computeFramesForOrientation:
           (UIInterfaceOrientation)interfaceOrientation {

   // Must be implemented in any sub class
}
```

Finally, you also have to add the new method declarations. Add the following to `DetailViewController.h`:

```
- (void) computeFrames:(UIView *) aView forOrientation:
             (UIInterfaceOrientation)interfaceOrientation;
- (void) computeFramesForOrientation:
             (UIInterfaceOrientation)interfaceOrientation;
```

Well, I don't know about you, but I'm glad to be done with the plumbing. Time to get on to something more interesting (to the user at least).

Putting MapKit through Its Paces

You've prepared the ground for some great map functionality, but now it's time to put the code in place so that you can get some real work done. Undergirding all this effort is the `MapKit.framework`. One of the great features of iOS 3.0 SDK and beyond is the `MapKit.framework`, which enables you to display a map and also do things with your map without having to do much work at all.

The map looks like the maps in the built-in applications and creates a seamless mapping experience across multiple applications.

MKMapView

The essence of mapping on the iOS devices is the `MKMapView`. It's a `UIView` subclass, and as you saw in the previous section, you can use it out of the box to create a world map. You use this class as-is to display map information and to manipulate the map contents from your application. It enables you to center the map on a given coordinate, specify the size of the area you want to display, and annotate the map with custom information.

REMEMBER

You added the `MapKit.framework` earlier in this chapter.

When you initialize a Map view, you can specify the initial region for that map to display. You do this by setting the *region* property of the map. A region is defined by a center point and horizontal and vertical distances, referred to as the *span.* The span defines how much of the map will be visible; it also determines the zoom level. The smaller the span, the greater the zoom.

The Map view supports the standard map gestures:

- Scroll
- Pinch zoom
- Double-tap zoom in
- Two-finger–tap zoom out (You may not even have known about that one.)

You can also specify the map type — regular, satellite, or hybrid — by changing a single property.

Because `MapKit.framework` was written from scratch, it was developed with the limitations of the iPhone (and subsequently iPad) in mind. As a result, it optimizes performance on the iPad by caching data as well as managing memory and seamlessly handling connectivity changes (like moving from 3G to Wi-Fi, for example).

The map data itself is Google-hosted map data, and network connectivity *is* required. And because `MapKit.framework` uses Google services to provide map data, using it binds you to the Google Maps/Google Earth API terms of service.

REMEMBER

Although you shouldn't subclass the `MKMapView` class itself, you can tailor a Map view's behavior by providing a delegate object. The delegate object can be any object in your application, as long as it conforms to the `MKMapViewDelegate` protocol. (You made the `MapController` the `MKMapView` delegate in the "Setting up the nib file" section, earlier in this chapter.)

Enhancing the map

What about showing your location on the map? That's just as easy!

In the `MapController.m` file, uncomment out `viewDidLoad` and add the code in bold:

```
- (void)viewDidLoad {

  [super viewDidLoad];
  mapView.showsUserLocation = YES;
}
```

`showsUserLocation` is a `MKMapView` property that tells the Map view whether to show the user location. If `YES`, you get that same blue pulsing dot you see displayed in the built-in Map application.

If you were to compile and run the application as it stands, you'd get what you see in Figure 15-8: a world map with a blue dot that represents the phone's current location. You'll probably have to pan the map over to see it. (There may be a lag until the iPad is able to determine that location, but you should see it eventually.)

TIP

If you don't see the current location, you might want to check and make sure you've connected the `mapView` outlet to the Map view in the nib file — see the "Setting up the nib file" section, earlier in the chapter.

REMEMBER

You get your current location *if you're running your app on the iPad.* If you're running it on the Simulator, that location is Apple — in beautiful, Cupertino, California, to be precise. Touching the blue dot also displays what's called an *annotation,* and I tell you how to customize the text to display whatever you cleverly come up with — including, as you discover in the upcoming "Adding Annotations" section, the address of the current location.

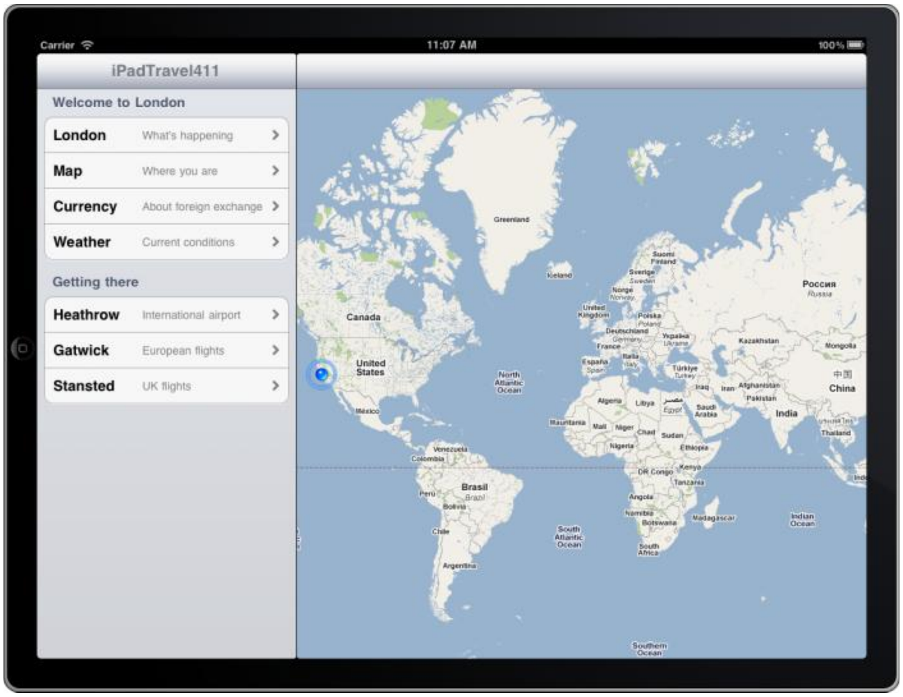


Figure 15-8: Displaying a map with a user location.

Although I won't have you do it here, you can also specify what kind of map type to use. The screen shots in this book (with the exception of Chapter 13) all use `MKMapTypeStandard` (the "standard" map view) — which displays a street map with roads and some road names. What I showed you in Chapter 13 was a map type of `MKMapTypeHybrid` — which displays a satellite image of the area with road and road name information layered on top (my favorite). Your third choice is `MKMapTypeSatellite`, which displays just the satellite imagery of the area.

If you wanted to change the map to `MKMapTypeHybrid`, all you would need to do is add the following statement to `viewDidLoad` in `MapController.m` (or anywhere you wanted to make a map type selection):

```
mapView.mapType = MKMapTypeHybrid;
```

It's about the region

Okay, now you have a blue dot on a map.

Having this global map is kind of interesting but not very useful if you're planning to go to London. The following sections show you what you would have to do to make the map more useful.

As I mention at the beginning of this chapter, ideally, when you get to Heathrow (or wherever), you should see a map that centers on Heathrow as opposed to the world map. To get there from here, however, is also pretty easy.

First you need to look at how you center the map.

Back in your Project window, add the following code to `MapController.m`:

```
- (void)updateRegionLatitude:(float) latitude
          longitude:(float) longitude
          latitudeDelta:(float) latitudeDelta
                    longitudeDelta:(float) longitudeDelta {

  MKCoordinateRegion region;
  region.center.latitude = latitude;
  region.center.longitude = longitude;
  region.span.latitudeDelta = latitudeDelta;
  region.span.longitudeDelta = longitudeDelta;
  [mapView setRegion:region animated:NO];
}
```

Also add the declaration to the `MapController.h` file.

Setting the *region* is how you center the map and set the zoom level. You accomplish all this with the following statement:

```
[mapView setRegion:region animated:NO];
```

A region is a Map view property that specifies four things (as illustrated in Figure 15-9):

1. `region.center.latitude` specifies the latitude of the center of the map.
2. `region.center.longitude` specifies the longitude of the center of the map.

   For example, if you were to set those values as

   ```
   region.center.latitude = 51.471184;
   region.center.longitude = -0.452542;
   ```

   the center of the map would be Heathrow airport.
3. `region.span.latitudeDelta` specifies the north-to-south distance (in latitudinal degrees) to display on the map. One degree of latitude is approximately 111 kilometers (69 miles). A `region.span.latitudeDelta` of 0.0036 would specify a north-to-south distance on the map of about a quarter of a mile. Latitudes north of the equator have positive values, whereas latitudes south of the equator have negative values.

4. `region.span.longitudeDelta` specifies the east-to-west distance (in longitudinal degrees) to display on the map. Unfortunately, the number of miles in one degree of longitude varies based on the latitude. For example, one degree of longitude is approximately 69 miles at the equator but shrinks to 0 miles at the poles. Longitudes east of the zero meridian (by international convention, the zero or Prime Meridian passes through the Royal Observatory, Greenwich, in east London) have positive values, and longitudes west of the zero meridian have negative values.

Although the span values provide an implicit zoom value for the map, the actual region you see displayed may not equal the span you specify because the map will go to the zoom level that best fits the region that is set. This also means that even if you just change the center coordinate in the map, the zoom level may change because distances represented by a particular span may change at different latitudes and longitudes. To account for that, those smart developers at Apple included a property you can set that will change the center coordinate without changing the zoom level.

```
@property (nonatomic) CLLocationCoordinate2D
          centerCoordinate
```

When you change the value of this property with a new `CLLocationCoordinate2D`, the map is centered on the new coordinate, and it updates span values to maintain the current zoom level.

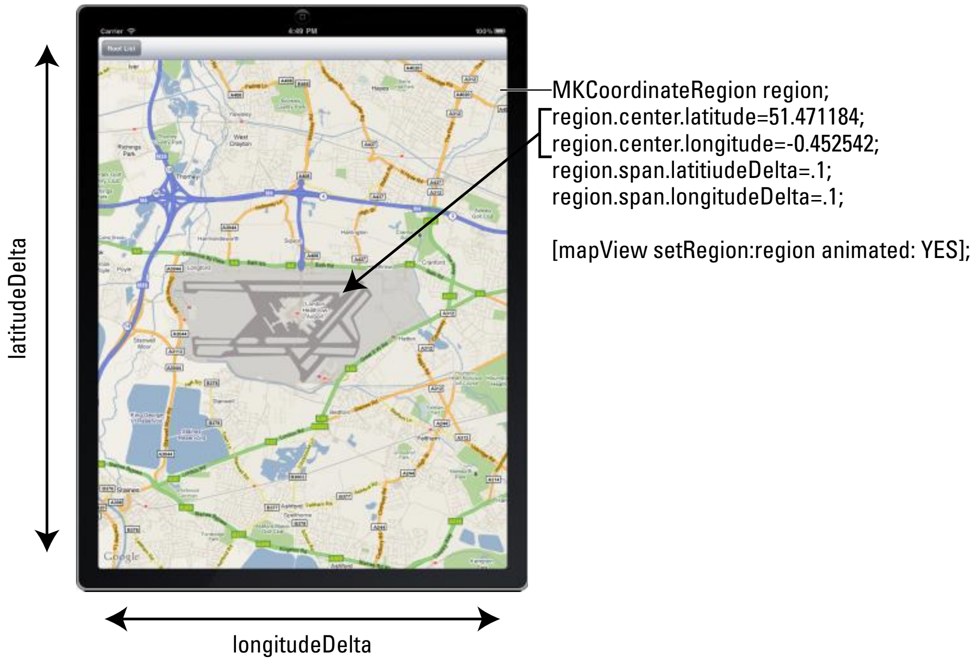
That `CLLocationCoordinate2D` type is something you'll be using a lot, so I'd like to explain that before I take you any further.

The `CLLocationCoordinate2D` type is a structure that contains a geographical coordinate using the *WGS 84 reference frame* — the reference coordinate system used by the Global Positioning System.

```
typedef struct {
CLLocationDegrees latitude;
CLLocationDegrees longitude;
} CLLocationCoordinate2D;
```

Here's a little explanation:

- `latitude` is the latitude in degrees. This is the value you set in the code you just entered (`region.center.latitude = latitude;`).
- `longitude` is the longitude in degrees. This is the value you set in the code you just entered (`region.center.longitude = longitude;`).



```
MKCoordinateRegion region;
region.center.latitude=51.471184;
region.center.longitude=-0.452542;
region.span.latitiudeDelta=.1;
region.span.longitudeDelta=.1;

[mapView setRegion:region animated: YES];
```

Figure 15-9: How regions work.

To center the map display on Heathrow, you send the `updateRegionLatitude:longitude: latitudeDelta:longitudeDelta` message (the code you just entered) when the view is loaded, that is, in the `viewDidLoad:` method in `MapController.m`. You already added some code there to display the current location, so add the following code in bold:

```
- (void)viewDidLoad {

   [super viewDidLoad];
  mapView.showsUserLocation = YES;
  CLLocationCoordinate2D initialCoordinate =
                              [destination initialCoordinate];
   [self updateRegionLatitude: initialCoordinate.latitude
        longitude: initialCoordinate.longitude
                      latitudeDelta:.1 longitudeDelta:.1];
  self.title = [destination mapTitle];
}
```

Take a look at what this code does:

1. The initialCoordinate message is sent to the Destination object to get the initial coordinates you want displayed. You're adding some additional functionality to the model, whose responsibility now includes specifying that location. The user may have requested that location when he or she set up the trip (I don't cover that topic in this book, leaving it as an exercise for you to do), or it may have been a default location that you decided on when you wrote the code (an airport specified in the destination, for example).
2. The Map title is set by sending the mapTitle message to the Destination object — adding another model responsibility.

For all of this to work, of course, you have to add some code to the Destination class.

Add the initialCoordinate and title methods to Destination.m.

```
- (CLLocationCoordinate2D)initialCoordinate {

  CLLocationCoordinate2D startCoordinate;
  startCoordinate.latitude = 51.471184;
  startCoordinate.longitude = -0.452542;
  return startCoordinate;
}
- (NSString *) mapTitle {

  return @" map";
}
```

You have to include the MapKit in Destination, so add the following to Destination.h:

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

And you'll also need to add the new method declarations. When you're done, the Destination.h file should look like Listing 15-11.

Listing 15-11: Destination.h

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>

@interface Destination : NSObject {

  NSString  *destinationName;
}
- (id) initWithName: (NSString *) theDestination;
- (CLLocationCoordinate2D) initialCoordinate;
- (NSString *) mapTitle;
@end
```

If you compile and build your project, you should see what's shown in Figure 15-10.

Although this is better, I'd rather be able to see both the airport (Heathrow) and my destination (London) at the same time. Although I could pinch to reduce the map, I show you in the "Displaying multiple annotations" section, later in this chapter, how to ensure that two (or more) locations are visible at the initial launch.

At this point, when the user touches Map in the Main view, iPadTravel411 displays a map centered on Heathrow, and if you pan over (a tedious task you'll fix soon) to Cupertino (or wherever you are), you can see the blue dot.

If you tap the blue dot (refer to Figure 15-8), you see a callout known as an *annotation* displaying the message `Current Location`. You can also add annotations on your own, which is what you do in the upcoming "Adding Annotations" section.

Dealing with failure

But what if the Internet isn't available? The Apple Human Interface Guidelines (and common sense) say that you should keep the user informed of what's going on. By virtue of the fact that you've made the `MapController` a `MKMapView` delegate, your app is in the position to send a message in the event of a load failure. Adding the following code to the `MapController.m` file makes it final:

```
- (void)mapViewDidFailLoadingMap:(MKMapView *)mapView
                              withError:(NSError *)error {

  NSLog(@"Unresolved error %@, %@", error,
                                       [error userInfo]);

  UIAlertView *alert = [[UIAlertView alloc]
      initWithTitle:@"Unable to load the map"
      message:@"Check to see if you have internet access"
      delegate:self cancelButtonTitle: @"Thanks"
      otherButtonTitles:nil];
  [alert show];
  [alert release];
}
```

Testing this alert business on the Simulator doesn't always work because it does some caching. You're better off testing it on the device by turning on Airplane mode.

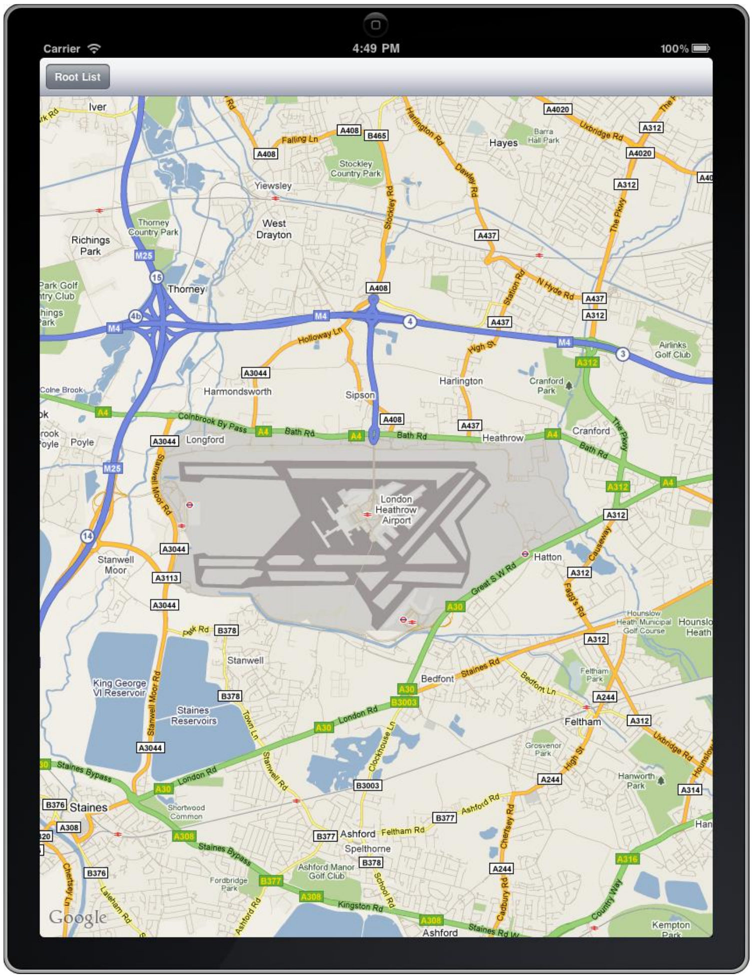


Figure 15-10: Regions determine what you see on the map.

Adding Annotations

The `MKMapView` class supports the ability to annotate the map with custom information. There are two parts to the annotation: the annotation itself, which contains the data for the annotation, and the Annotation view that displays the data.

REMEMBER

An annotation plays a similar role to the dictionary you created in Chapter 14, where the dictionary was meant to hold the text to be displayed in the cell of a Table view. Both dictionaries and annotations act as models for their corresponding view, with a view controller connecting the two.

Tracking location changes

You can also track changes in user location by using key-value observing, which enables you to move the map as the user changes location. I don't go into detail on key-value observing here, other than to show you the code.

If you want to track location changes by using key-value observing, add the code in bold to `viewDidLoad:` in `MapController.m` to add an observer that's to be called when a certain value is changed — in this case, `userLocation`.

```
- (void)viewDidLoad {
  [super viewDidLoad];
  mapView.showsUserLocation = YES;
  CLLocationCoordinate2D initialCoordinate =
                                             [map
   initialCoordinate];
  [self updateRegionLatitude:initialCoordinate.latitude
                      longitude:initialCoordinate.longitude
                           latitudeDelta:.06
   longitudeDelta:.06];
  self.title = [trip mapTitle];
  [mapView.userLocation addObserver:self
   forKeyPath:@"location"
                                             options:0
   context:NULL];
}
```

Adding that code causes the `observeValueForKeyPath::` message to be sent to the observer (self or the `Destination`). To implement the method in `Destination.m`, enter this method:

```
- (void)observeValueForKeyPath:(NSString *) keyPath
               ofObject:(id)object change:(NSDictionary *)
   change
                                            context:(void *)
   context {

  NSLog (@"Location changed");
}
```

In this method, the `keyPath` field returns `mapView.userLocation.location`, which you can use to get the current location. In this example, I'm simply displaying a message on the Debugger Console, but as I said, after the user moves a certain amount, you may want to re-center the map.

Note: This isn't exactly the same location you'd get from `CLLocationManager` — it's optimized for the map, whereas `CLLocationManager` provides the raw user location.

Of course, you have to run this on the iPad for the location to change.

An Annotation object is any object that conforms to the MKAnnotation protocol; typically, they're existing classes in your application's model. The job of an Annotation object is to know its location (coordinate) on the map along with the text to be displayed in the callout. The MKAnnotation protocol requires a class that adopts that protocol to implement the coordinate property. If you think about what user experience you're trying to achieve with this app, you'll see there are really two places you would want to display. The first is the airport, and the second is the city I'm going to — London.

Although there will be much more in the City and Airport classes you'll create, it makes sense that they also know the coordinate and callout data. I'll have you start by creating a base class for both, one that implements the MKAnnotation protocol, and then derive the City and Airport classes from Annotation.

1. **I'll put my new classes into the Model classes group, so select that group in the Groups & Files pane.**
2. **Choose File➪New File yet again from the main menu (or press ⌘+N) to open the New File dialog.**
3. **In the leftmost column of the dialog, select Cocoa Touch Classes under the iOS heading just like you did before, but this time select the Objective-C class template in the topmost pane, making sure that the Subclass drop-down menu has NSObject selected. Then click Next.**

   You see a new dialog asking for some more information.
4. **Enter** Annotation **in the File Name field and then click Finish.**

Do the same for City and Airport.

To code the Annotation class interface, add the code in bold in Listing 15-12.

Listing 15-12: The Annotation Class Interface

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

@interface Annotation :  NSObject  <MKAnnotation> {
  CLLocationCoordinate2D coordinate;
  NSString * title;
  NSString * subtitle;
}
@property (nonatomic, retain) NSString *title;
@property (nonatomic, retain) NSString *subtitle;
@property (nonatomic) CLLocationCoordinate2D coordinate;

@end
```

The first thing you do with this code is have `Annotation` adopt the `MKAnnotation` protocol — you also need to import `MapKit` here. The rest of the interface is simply what is required to adopt the MKAnnotation protocol.

REMEMBER

The `MKAnnotation` protocol requires a `coordinate` property — the `title` and `subtitle` methods are optional. It turns out of course that by making `title` and `subtitle` properties and synthesizing the accessors, you have those very methods.

Add the code in bold in Listing 15-13; that is all you need for the implementation.

Listing 15-13: The Annotation Class Implementation

```
#import "Annotation.h"

@implementation Annotation

@synthesize coordinate;
@synthesize title;
@synthesize subtitle;

@end
```

All you need to do, then, is synthesize the accessors.

You now have a base class that has everything a class needs to be an annotation, except for one important thing — the data.

You'll add that in the `City` and `Airport` objects.

Start by adding the code in bold to `City.h` and deleting the BUI in Listing 15-14.

Listing 15-14: The City Interface

```
#import "Annotation.h"

@interface City : NSObject Annotation {

}

@end
```

As you can see, you're deriving `City` not from the normal `NSObject`, but from the `Annotation` class that is already set up to do all the things an annotation must do. You have to add the `init` method in Listing 15-15 to `City.m`.

Listing 15-15: The City init

```
- (id) init {

  if ((self = [super init])) {
    coordinate.latitude = 51.500153;
    coordinate.longitude= -0.126236;
    self.title = @"London";
    self.subtitle = @"A great city";
  }
  return self;
}
```

Do the same thing for `Airport.h` and `.m` as shown in Listings 15-16 and 15-17.

Listing 15-16: The Airport Interface

```
#import "Annotation.h"

@interface Airport : NSObject Annotation {

}

@end
```

Listing 15-17: The Airport init

```
- (id) init {

  if ((self = [super init])) {
    coordinate.latitude = 51.471184;
    coordinate.longitude= -0.452542;
    self.title = @"Heathrow";
    self.subtitle = @"International airport";
  }
  return self;
}
```

You'll notice that even though the `RootViewController` shows three airports, I'm hard-coding Heathrow here. Normally, you would create an `Airport` object and have it initialize itself based on some argument (an ID, for example) you pass in when you create it.

Of course, you'll also need to create the objects; here's where `Destination` starts to show its value. You create `City` and `Airport` in `Destination`'s initialization method. Add the code in bold in Listing 15-18 to the `initWithName:` method in `Destination.m` and add the `import` statements.

Listing 15-18: Update initWithName:

```
#import "Destination.h"
#import <MapKit/MapKit.h>
#import "City.h"
#import "Airport.h"

@implementation Destination

- (id) initWithName:(NSString *) theDestination {

  if ((self = [super init])) {
    destinationName = theDestination;
    city = [[City alloc] init];
    airport = [[Airport alloc] init];
  }
  return self;
}
```

So, as soon as you create the `Destination` object in the `iPadTravel411 AppDelegate` method `awakeFromNib` (you did that in Chapter 14), it turns around and creates the objects it will need: `City` and `Airport`.

You also need to add the code in Listing 15-19 to `Destination.m` to create the annotations. I show you later how that is used.

Listing 15-19: createAnnotations

```
- (NSArray*) createAnnotations {

  NSMutableArray* annotations =
              [[NSMutableArray alloc] initWithCapacity:2];
  [annotations addObject:city];
  [annotations addObject:airport];

  return annotations;
}
```

Normally, you wouldn't hardcode the `initWithCapacity` number — you would get that from a count of some array of model objects. I can do it that way here because I know exactly how many annotations I am adding.

You need to update the `Destination` interface by adding the code in bold in Listing 15-20.

Even though `annotations` is a mutable array, I return it as a simple array because that is all that is needed.

Listing 15-20: Update the Destination Interface

```
#import <Foundation/Foundation.h>
#import <MapKit/MapKit.h>
@class City;
@class Airport;

@interface Destination : NSObject {

  NSString  *destinationName;
  City      *city;
  Airport   *airport;
}
- (id) init:(NSString *) theDestination;
- (CLLocationCoordinate2D)initialCoordinate;
- (NSString *) mapTitle;
- (NSArray *) createAnnotations;

@end
```

So far so good. `Annotation` has adopted the `MKAnnotation` protocol, declared a `coordinate` property, and implemented `title` and `subtitle` methods using `@synthesize`, and you've added data by creating `City` and `Airport`. The `Destination` object then creates an array of these annotations. The only thing left to do is to get the annotations from the `Destination` and then send the array to the Map view to get the annotations displayed.

To do that, add the code in bold in Listing 15-21 to the `viewDidLoad:` method in `MapController.m` so that a message gets sent to `Destination` to create the annotations and send the annotations to the Map view to display them.

Listing 15-21: Adding to viewDidLoad

```
- (void)viewDidLoad {

  - (void)viewDidLoad {

  [super viewDidLoad];
  mapView.showsUserLocation = YES;
  CLLocationCoordinate2D initialCoordinate =
  [destination initialCoordinate];
  [self updateRegionLatitude:initialCoordinate.latitude
                   longitude:initialCoordinate.longitude
               latitudeDelta:.1 longitudeDelta:.1];
  self.title = [destination mapTitle];
  NSArray* destinationAnnotations =
                        [destination createAnnotations];
  annotations = [[NSMutableArray alloc]
         initWithCapacity:[destinationAnnotations count]];
  [annotations
```

```
                addObjectsFromArray:destinationAnnotations];
   [mapView addAnnotations:destinationAnnotations];
}
```

The MapController sends the addAnnotations: message to the Map view, passing it an array (the argument specified by the method) of objects that conform to the MKAnnotation protocol; that is, each one has a coordinate property and an optional title (and subtitle) method if you want to actually display something in the annotation callout.

The Map view places annotations on the screen by sending its delegate the mapView:viewForAnnotation: message. This message is sent for each annotation object in the array. Here you can create a custom view or return nil to use the default view. (If you don't implement this delegate method — which you won't, in this case — the default view is used; these are the red pins you see on the map.)

Creating your own Annotation views is beyond the scope of this book — although I show you how to use a MapKit-supplied Annotation view MKPinView to create draggable annotations later in this chapter.

For the time being, the default annotation view is fine for your purposes. It displays a pin in the location specified in the coordinate property of the annotation delegate and when the user touches the pin, the optional title and subtitle text will display if the title and subtitle methods are implemented in the annotation delegate.

You'll also have to add the new instance variable to MapController.h.

```
NSMutableArray        *annotations;
```

TECHNICAL STUFF

You can also add callouts to the Annotation callout, such as a Detail Disclosure button (the one that looks like a white chevron in a blue button in a Table view cell) or an Info button (like the one you see in many of the utility apps), without creating your own Annotation view. Again, another exercise for you, if you're feeling frisky.

If you compile and build your project, you can check out one of the annotations you just added in Figure 15-11.

You may be asking yourself at this point, "Why is Destination creating the annotation instead of MapController and why not have the data in Destination rather than City and Airport?'

To start with, you don't want any data at all in MapController. That's the purview of the model classes.

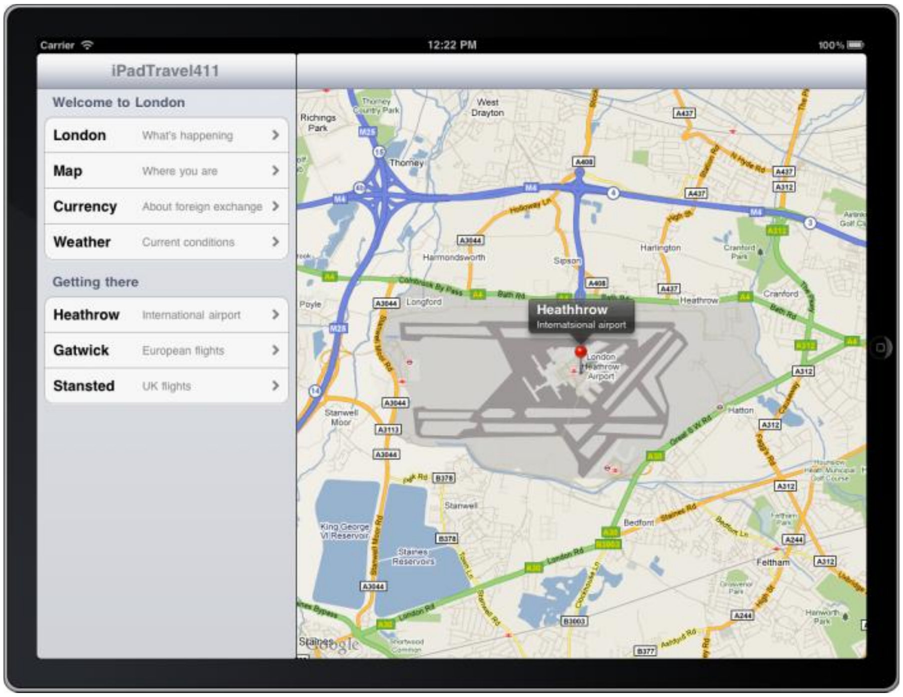


Figure 15-11: An annotation.

Given that, you have a couple of options when it comes to creating the model objects needed by the view controllers. One way is to have the view controllers create the model objects they'll use. For example, the `AirportController` would create the `Airport` object, and so on. That eliminates the indirection you saw in the previous section; you know, having to go through the `Destination` object to the `Airport` object that has the data.

Although this does work, and I've actually done that in past versions, I'd like you to consider a different approach that results in a more extensible program. (I explain this in detail in *Objective-C For Dummies,* so if you're curious, you may want to pick up a copy of that book.)

One of the advantages of the MVC design pattern I explain in Chapter 7 is that it allows you to separate these three groups in your application (model objects, view objects, and controller objects) and work on them separately. If each group has a well-defined interface, it encapsulates many of the kinds of changes that are often made so that they don't affect the other groups. This is especially true of the model and view controller relationship.

If the view controllers have minimal knowledge about the model, you can change the model objects with minor impact on the view controllers.

As I said, what makes this possible is a well-defined interface, which I show you how to develop in this section. You'll create an interface between the model and the controllers by using a technique called *composition,* which is a useful way to create interfaces.

I'm a big fan of composition, because it's another way to hide what's really going on behind the curtain. It keeps the objects that use the composite object ignorant of the objects the composite object uses, and it actually makes the components blissfully unaware of each other, allowing you to switch components in and out at will.

The `Destination` class provides the basis for such an architecture, and while I won't fully implement it here, you'll understand the structure and have no trouble extending it on your own.

But just to be reasonable, as you will see, for the rather simple model data such as Currency and Weather, I will keep the data in the `Destination` class. Just don't forget that, in a more robust program where there was more to currency and weather than supplying a URL (as you will see), you should create classes for them as well. I would.

Going to the Current Location

Although you can pan to the user location on the map, in this case it's kind of annoying, unless you're actually coding this at or around London. To remove at least that annoyance from your life, I want to show you how easy it is to add a button to the toolbar bar to zoom you in to the current location and then back to the map region and span you're currently displaying.

1. **Add the following code to add the button in the `MapController` method `viewdidLoad`.**

   You have quite a bit of code there already, so this is just what to add:

```
locateButton =
  [[UIBarButtonItem alloc] initWithTitle:@"Location"
  style:UIBarButtonItemStyleBordered target:self
                     action:@
        selector(goToLocation:)];
NSMutableArray *items = [[toolbar items] mutableCopy];
[items addObject:locateButton];
[self.toolbar setItems:items animated:YES];
[items release];
```

When the user taps the Locate button you create here, you've specified that the `goToLocation:` message is to be sent (`action:@selector(goToLocation:)`) to the `MapController` (`target:self`).

`viewDidLoad` is called only the first time the view is loaded, and this is the place we want to insert the Locate button. (You'll notice it plays well with the Root View button when you are in portrait mode.) Because we're creating this view controller/view only once, this button will stay around for the duration of the application. As I explain in Chapter 18, because you will have to manage the toolbar as you move from view controller to view controller, you want to save a reference to it as an instance variable. So, go ahead and add the following to `MapController.h`:

```
UIBarButtonItem *locateButton;
```

2. **Add the `goToLocation:` method to `MapController.m`.**

```
- (IBAction)goToLocation:(id)sender {

  MKUserLocation *annotation = mapView.userLocation;
  CLLocation *location = annotation.location;
  if (nil == location)
    return;
  CLLocationDistance distance =
                MAX(4*location.
        horizontalAccuracy,500);
  MKCoordinateRegion region =
         MKCoordinateRegionMakeWithDistance
            (location.coordinate, distance, distance);
  [mapView setRegion:region animated:NO];
  locateButton.action = @selector(goToDestination:);
  locateButton.title = @"Destination";
}
```

When the user presses the Locate button, you first check to see if the location is available. (It may take a few seconds after you start the application for the location to become available.) If not, you simply return. (You could, of course, show an alert informing the user what's going on and try again in 10 seconds or so — I leave that up to you.)

If it's available, you compute the span for the region you'll be moving to. In this case, the code

```
CLLocationDistance distance =
              MAX(4*location.horizontalAccuracy,1000);
```

computes the span to be four times the `horizontalAccuracy` of the device (but no less than 1,000 meters). `horizontalAccuracy` is a radius of uncertainty given the accuracy of the device; that is, the user is somewhere within that circle.

You then call the `MKCoordinateRegionMakeWithDistance` function that creates a new `MKCoordinateRegion` from the specified coordinate and distance values. `distance` and `distance` correspond to `latitudinalMeters` and `longitudinalMeters`, respectively. (I'm using the same value for both arguments here.)

If you didn't want to change the span, you could have simply set the Map view's `centerCoordinate` property to `userLocation`, and — as I mention earlier in the "It's about the region" section — that would have centered the region at the `userLocation` coordinate without changing the span.

3. **When the user taps the button, change the title on the button to the Map title and change the `@selector` to `(goToDestination:)`.**

   The result is that the next time the user touches the button, the `goToDestination:` message will be sent.

4. **Add the `goToDestination:` message.**

```
- (IBAction) goToDestination:(id)sender {

  CLLocationCoordinate2D initialCoordinate =
                      [destination initialCoordinate];
  [self updateRegionLatitude:
       initialCoordinate.latitude longitude:
       initialCoordinate.longitude
                latitudeDelta:.1 longitudeDelta:.1];
  locateButton.title = @"Location";
  locateButton.action = @selector(goToLocation:);
}
```

   This step sets the region and toggles the button title back to `Location` and the selector back to `goToLocation:`.

5. **Add both method declarations to the `MapController.h` file.**

```
- (IBAction) goToLocation:(id)sender;
- (IBAction) goToDestination:(id)sender;
```

You can see the result of touching the Locate button in Figure 15-12.

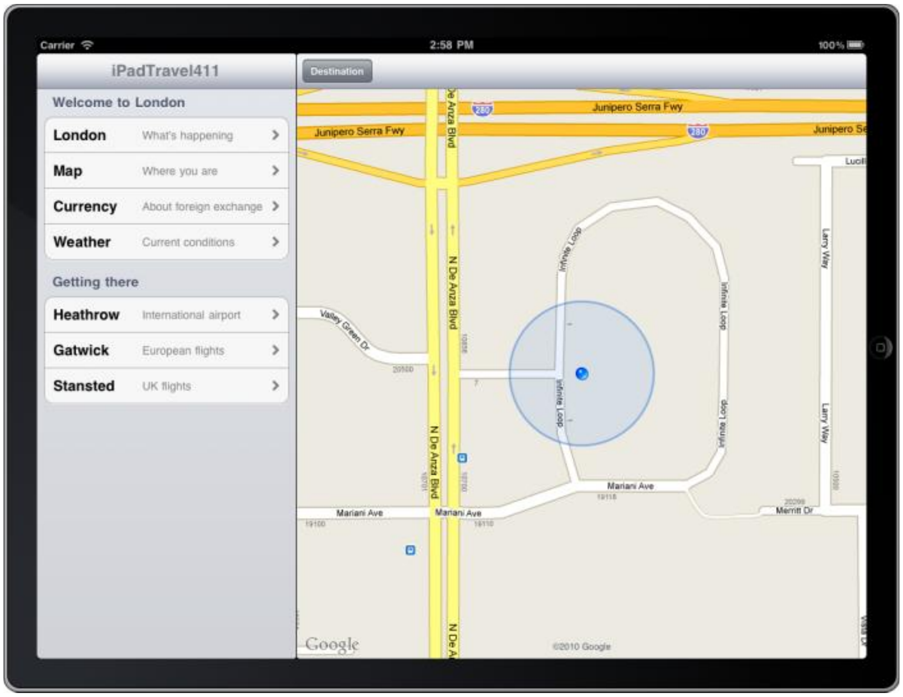


Figure 15-12: Go to the current location.

Displaying Multiple Annotations

Although what you see in the Map view is pretty darn good, under some circumstances (like when you just arrived at the airport, for example) it would really be better if the user could automatically see both Heathrow and London (and their respective annotations) on the map without hard-coding the region as you have been doing so far. This isn't that difficult to do. Add the following code in Listing 15-22 to `MapController.m`.

Listing 15-22: Computing the Region Based on the Annotations

```
- (MKCoordinateRegion)
                 regionForAnnotationGroup:(NSArray*) group {

  double maxLonWest= 0;
  double minLonEast = 180;
  double maxLatNorth = 0;
  double minLatSouth = 180;

  for (Annotation *location in group) {
    if (fabs(location.coordinate.longitude) >
                                    fabs(maxLonWest))
      maxLonWest = location.coordinate.longitude;
    if (fabs(location.coordinate.longitude) <
                                          fabs(minLonEast))
      minLonEast = location.coordinate.longitude;
    if (fabs(location.coordinate.latitude) >
                                         fabs(maxLatNorth))
      maxLatNorth = location.coordinate.latitude;
    if (fabs(location.coordinate.latitude) <
                                         fabs(minLatSouth))
      minLatSouth = location.coordinate.latitude;
  }

  double centerLatitide =
        maxLatNorth - (((maxLatNorth) - (minLatSouth))/2);
  double centerLongitude =
           maxLonWest - (((maxLonWest) - (minLonEast))/2);

  MKCoordinateRegion region;
  region.center.latitude =  centerLatitide;
  region.center.longitude = centerLongitude;
  region.span.latitudeDelta =
                        fabs(maxLatNorth - minLatSouth);
  if (fabs(maxLatNorth - minLatSouth) <= .005)
                        region.span.latitudeDelta = .01;
  region.span.longitudeDelta =
                          fabs(maxLonWest - minLonEast);
  if (fabs(maxLonWest - minLonEast) <= .005)
                       region.span.longitudeDelta = .01;

  return region;
}
```

To make this work, you have to add to `mapController.m`, `#import "Annotation.h"`, and the declaration to `MapController.h`, as follows:

```
- (MKCoordinateRegion)
              regionForAnnotationGroup:(NSArray*) group;
```

If you don't, you get an error message:

```
incompatible type for argument 1 of 'setRegion:animated:'
```

REMEMBER

I warn you about this because it isn't that easy to figure out what's really happening the first time you do this. It's because the message you're sending hasn't been declared in the interface. Keep this one in mind.

This code computes the region that includes both annotations. Frankly, this is really beyond the scope of this book, but it's handy to be able to know how to do this, so I include it here. I just summarize how it works and leave it for you to go through it step by step.

In general terms, here's how the code works: You want both annotations to fit on one screen, so the code goes through each annotation and determines the maximum north and south latitudes and the maximum east and west longitudes. The only trick here is that, because latitude and longitude can be negative, it uses a function `fabs` to get the absolute value of a floating point number. After that, it's simply a matter of finding the center latitude and longitude and setting the region `center` and then taking the maximum west and maximum east longitude and the maximum north and maximum south latitude and using that as the `span`. (You also decrease the span by `.005` in both directions to make sure that no pins are right on the edge of the screen.) I decided that I didn't want the `span` to ever be less than `.005`, so if it is, I arbitrarily make it `.01`.

All that is left is to change the `MapController` `viewDidLoad` method to use this method rather than the `Destination` `initialCoordinate` and `updateRegionLatitude::::`.

To do that, add the stuff in bold and then delete the BUI code in `view DidLoad` as shown in Listing 15-23.

Listing 15-23: viewDidLoad Now Results in Both Annotations Being Visible

```
- (void)viewDidLoad {

  [super viewDidLoad];
  mapView.showsUserLocation = YES;
  CLLocationCoordinate2D initialCoordinate =
                                        [destination
          initialCoordinate];
  [self updateRegionLatitude:initialCoordinate.latitude
         longitude: initialCoordinate.longitude
                     latitudeDelta:.1 longitudeDelta:.1];
  self.title = [destination mapTitle];
  NSArray* destinationAnnotations =
                               [destination
          createAnnotations];
  annotations = [[NSMutableArray alloc]
           initWithCapacity:[destinationAnnotations
          count]];
  [annotations  addObjectsFromArray:
          destinationAnnotations];
  [mapView addAnnotations:destinationAnnotations];
  [mapView setRegion:[self regionForAnnotationGroup:
                   destinationAnnotations] animated:NO];
  locateButton =
      [[UIBarButtonItem alloc] initWithTitle:@"Location"
      style:UIBarButtonItemStyleBordered target:self
                        action:@selector(goToLocation:)];
  NSMutableArray *items = [[toolbar items] mutableCopy];
  [items addObject:locateButton];
  [self.toolbar setItems:items animated:YES];
```

```
   [items release];
}
```

You need to also make the changes in bold to `goToDestination:` in order to compute the region rather than just get by using the hard-coded one.

```
- (IBAction) goToDestination:(id)sender {

   CLLocationCoordinate2D initialCoordinate =
                                [destination initialCoordinate];
   [self updateRegionLatitude:initialCoordinate.latitude
                 longitude:initialCoordinate.longitude
                          latitudeDelta:.1 longitudeDelta:.1];
   [mapView setRegion:
       [self regionForAnnotationGroup:annotations]
                                                    animated:NO];
   locateButton.title = @"Location";
   locateButton.action = @selector(goToLocation:);
}
```

At this point, you can also delete the `initialCoordinate` declaration and method in `Destination.h` and `.m`, respectively, and the `updateRegionLatitude::::` declaration and method in `MapController.h` and `.m`, respectively.

Figure 15-13 shows the results of your work.

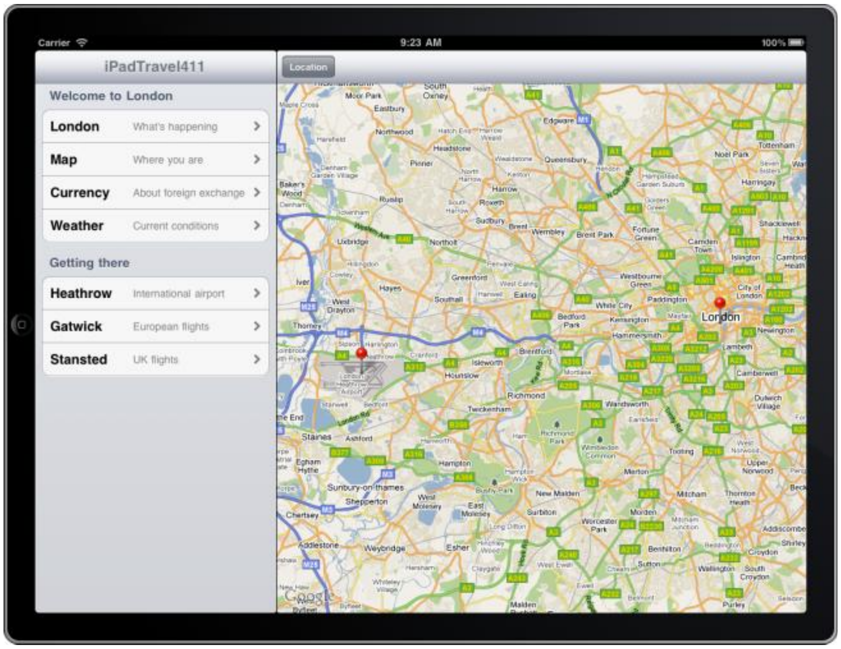


Figure 15-13: A better way to compute a region.

After you make regions something you compute rather than something you hard-code, all of a sudden you're confronted with another issue — you want the map to be able to be displayed in both landscape and portrait modes. The problem is that if the application opens in landscape mode and computes the correct region, when you switch to portrait mode, the annotations may no longer visible.

So this is a good time to introduce you to `didRotateFromInterfaceOrientation`. This message is sent to the `MapController` when the user interface orientation changes. Fortunately, all you have to do to get this to work for you is add the code in Listing 15-24 to `MapController.m`.

There is another nuance here, though. You don't want to set the region if you're showing the user location and then let the view controller implement the default behavior for the device rotation. Although you could have used an instance variable to let you know when you were displaying the user location, I take the easy way out, as you can see in Listing 15-24, and check to see what the button says instead.

Listing 15-24: Accounting for User Rotation Changes

```
- (void)didRotateFromInterfaceOrientation:
        (UIInterfaceOrientation)fromInterfaceOrientation {

   if ([locateButton.title isEqualToString:@"Location"])
      [mapView setRegion:
   [self regionForAnnotationGroup:annotations]
                                            animated:NO];
}
```

Geocoding

Seeing where I am on the map is all fine and dandy, but stickler that I am, I'd also like to know the exact street address. (If I have the address, I could also write some code to turn the iPad's current address into an Address Book contact, but I'll allow you the pleasure of figuring that out on your own.)

Being able to go from a coordinate on a map to a street address is called *reverse geocoding,* and thankfully the ability to do that is supplied by the `MapKit`. *Forward geocoding* (also called just geocoding), which converts an address to a coordinate, doesn't come with the `MapKit`, although many free and commercial services that *can* do that are available.

TIP

Keep in mind that the location may not be completely accurate — remember that `horizontalAccuracy` business in the "Going to the Current Location" section, earlier in this chapter? For example, because my office is very close to my property line, my location sometimes shows up with my next-door neighbor's address.

Adding reverse geocoding to iPhoneTravel411 will enable you to display the address of the current location. Just follow these steps:

1. **Import the reverse geocoder framework into `MapController.h` (it is actually part of the MapKit framework, so you don't have to add a new framework) and have `MapController` adopt the `MKReverseGeocoderDelegate` protocol.**

   ```
   #import <MapKit/MKReverseGeocoder.h>

   @interface MapController : DetailViewController
         <MKMapViewDelegate, MKReverseGeocoderDelegate> {
   ```

2. **Add an instance variable to hold a reference to the geocoder object, and another one to hold the annotation you're reverse geocoding for.**

   ```
    MKReverseGeocoder    *reverseGeocoder;
   Annotation           *selectedAnnotation;
   ```

   You'll use this `reverseGeocoder` to release the `MKReverseGeocoder` after you get the current address.

   `selectedAnnotation` will be assigned by `gotoLocation:`. It will also be used later in the "But What If I Don't Want to Go to London?" section, when you need to reverse geocode an annotation you're dragging. You also need to add an `@class Annotation;` statement to `MapController.h`.

3. **Add the method `reverseGeocoder:didFindPlacemark:` to `MapController.m`.**

   ```
   - (void)reverseGeocoder:(MKReverseGeocoder *) geocoder
             didFindPlacemark:(MKPlacemark *) placemark {

     NSDictionary* addressDictionary =
                           placemark.addressDictionary;

     NSMutableString* addressString;
     if ([addressDictionary objectForKey:@"Street"]) {
       addressString =
         [[NSMutableString alloc] initWithString:
             [addressDictionary objectForKey:@"Street"]];
       selectedAnnotation.subtitle =
                [addressDictionary objectForKey:@"City"];
   ```

```
  }
  else {
    addressString =
       [[NSMutableString alloc] initWithString:
            [addressDictionary objectForKey:@"City"]];
    selectedAnnotation.subtitle =
            [addressDictionary objectForKey:@"State"];
  }
  selectedAnnotation.title = addressString;
  [addressString release];
  [reverseGeocoder release];
  reverseGeocoder = nil;
}
```

The `reverseGeocoder:didFindPlacemark:` message to the delegate is sent when the `MKReverseGeocoder` object successfully obtains *placemark* information for its coordinate. An `MKPlacemark` object stores placemark data for a given latitude and longitude. Placemark data includes the properties that hold the country, state, city, and street address (and other information) associated with the specified coordinate, for example. (Several other pieces of data are available that you might also want to examine.)

- `country`: Name of country
- `administrativeArea`: State
- `locality`: City
- `thoroughfare`: Street address
- `subThoroughfare`: Additional street-level information, such as the street number
- `postalCode`: Postal code

There's also another property that is available: `addressDictionary` that you use here. A number of keys are available, such as the following:

- `CountryCode`
- `Street`: The street number and name
- `SubAdministrativeArea`
- `SubThoroughfare`
- `City`
- `ZIP`
- `State`
- `Thoroughfare`
- `Country`
- `FormattedAddressLines`

These are similar to the corresponding properties, but the addition of `Street` and `FormattedAddressLines` makes it a little easier.

In this implementation, you're setting the user location annotation (`userLocation`) title (supplied by `MapKit`) to the value of the `Street` key (street address) in the `addressDictionary`. You assign the subtitle the value of the `City` key.

You notice that you do engage in some error checking here. If there's a street address, you set the title to the street address and the subtitle to the city.

```
if ([addressDictionary objectForKey:@"Street"]) {
    addressString =
      [[NSMutableString alloc] initWithString:
         [addressDictionary objectForKey: @"Street"]];
    selectedAnnotation.subtitle =
            [addressDictionary objectForKey:@"City"];
}
```

If there's no street address, you set the title to the city and subtitle to the state.

```
addressString =
  [[NSMutableString alloc] initWithString:
          [addressDictionary objectForKey: @"City"]];
selectedAnnotation.subtitle =
            [addressDictionary objectForKey:@"State"];
```

You might also want to do an additional level of checking here if there's no city, and so on.

You may be wondering about why I'm spending so much time on geocoding because, as of now, the only geocoding you do is for the user location. But as you soon see, you will also geocode the new location if the user moves the pin on the map.

Finally you release the geocoder and set the instance variable to `nil`.

TECHNICAL STUFF

A `placemark` is also an annotation and conforms to the `MKAnnotation` protocol, whose properties and methods include the placemark coordinate and other information. Because they're annotations, you can add them directly to the Map view.

The `reverseGeocoder:didFailWithError:` message is sent to the delegate if the `MKReverseGeocoder` couldn't get the placemark information for the coordinate you supplied to it. (This is a required `MKReverseGeocoderDelegate` method.)

4. **Add the method `reverseGeocoder:didFailWithError:` to `MapController.m`.**

```
- (void)reverseGeocoder:(MKReverseGeocoder *)
          geocoder didFailWithError:(NSError *) error{

  NSLog(@"Reverse Geocoder Failure! due to error in
        domain: %@ with error code: %u, description:
        %@, and reason: %@",
        error.domain, error.code,
        [error localizedDescription],
        [error localizedFailureReason]);
  [reverseGeocoder release];
  reverseGeocoder = nil;
}
```

This message is sent to its delegate when the geocoder fails. This can happen for a variety of reasons — the service is down or it can't find an address for the coordinate. You get back an `error` object which can have some useful information. I'll leave you to explore the detail of the error information on your own.

While I simply log a message here, you may want to expand the user interface to inform the user of what's happening. Although that isn't important in this case — you can always just leave the annotation as "Current location" — when you start dragging annotations, as you will in the next section, you might want to develop a plan for what to display in the annotation if the geocoder fails.

Finally you release the geocoder and set the instance variable to `nil`.

Of course, in order to get the reverse geocoder information, you need to create an `MKReverseGeocoder` object. Make the `MapController` a delegate, send it a `start` message, and then release it when you're done with it.

Allocate and start the reverse geocoder and add the `MapController` as its delegate in the `MapController`'s `goToLocation:` method by adding the code in bold to `goToLocation` in `mapController.m`.

```
- (IBAction)goToLocation:(id)sender {

  MKUserLocation *annotation = mapView.userLocation;
  CLLocation *location = annotation.location;
  if (nil == location)
    return;
  CLLocationDistance distance =
                 MAX(4*location.
        horizontalAccuracy,500);
  MKCoordinateRegion region =
        MKCoordinateRegionMakeWithDistance
             (location.coordinate, distance,
        distance);
  [mapView setRegion:region animated:NO];
  locateButton.action = @selector(goToDestination:);
  locateButton.title = @"Destination";
  selectedAnnotation =
                   (Annotation *)mapView.userLocation;
  reverseGeocoder = [[MKReverseGeocoder alloc]
             initWithCoordinate:location.coordinate];
  reverseGeocoder.delegate = self;
  [reverseGeocoder start];
}
```

Notice how you initialize the `MKReverseGeocoder` with the coordinate of the current location. You also have to cast `mapView.userLocation` to an `Annotation`. While `MKUserLocation` (the class of `mapView.userLocation`) is not a sub class of Annotation, they both adopt the same protocol, and because it is the only thing you will be using in the protocol methods and properties, this is a safe cast.

5. **Release the `MKReverseGeocoder` by adding the code in bold to `goToDestination:`.**

```
- (IBAction) goToDestination:(id)sender {

  if (reverseGeocoder) {
    reverseGeocoder.delegate = nil;
    [reverseGeocoder release];
    reverseGeocoder = nil;
  }
  [mapView setRegion:[self regionForAnnotationGroup:
                           annotations] animated:NO];
  locateButton.title = @"Location";
  locateButton.action = @selector(goToLocation:);
}
```

You release the `MKReverseGeocoder` in this method because although you start the `MKReverseGeocoder` in the `goToLocation:` method, it actually doesn't return the information in that method. It operates asynchronously; when it either constructs the placemark or gives up, it sends the message `reverseGeocoder:didFindPlacemark:` or `reverseGeocoder:didFailWithError:`, respectively. If you're returning to the original Map view, however, you no longer care whether it succeeds or fails because you no longer need the placemark, and you release the `MKReverseGeocoder`, set the instance variable to `nil`, and set the delegate to `nil`.

Figure 15-14 shows the result of your adventures in reverse geocoding.

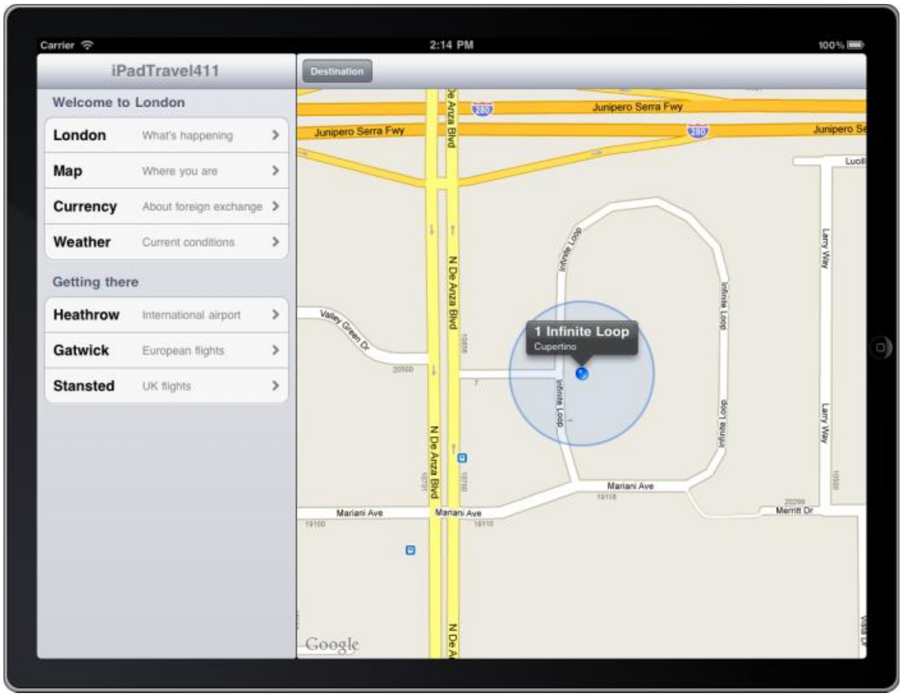


Figure 15-14: Reverse geocoding.

But What If I Don't Want to Go to London?

As you're aware, all the destination coordinates and locations are hard-coded in iPadTravel411. But in your own apps, you probably would want to give the user control over what's displayed.

In this app, one could argue that it's not really a problem; it's a travel guide to London, after all. But in reality, London is a big place, and the user might want to be able to adjust the destination more precisely.

Although you could create a modal dialog to allow the user to add or change a map location, that's beyond the scope of this book. But one thing I show you is how to create a draggable annotation — in this case, the `City` annotation — so that the user can move his or her destination to exactly where he or she wants to go.

The way to do that is pretty simple. All you do is set an Annotation view property. Of course, to do that you're going to need access to the Annotation view. So, instead of using the default view, as you have been doing, I show you how to use an SDK-supplied Annotation view — `MKPinAnnotationView`. In addition, you'll also be able to set the pin color as well as animate the pin to drop on to the map.

All you have to do is add the code in Listing 15-25 to `MapController.m`. `mapView:viewForAnnotation:` is a `MKMapView` delegate method that is automatically invoked before the Map view displays the annotation and gives you a chance to customize the view accordingly.

Listing 15-25: mapView:viewForAnnotation:

```
- (MKAnnotationView *)mapView:(MKMapView *)aMapView
         viewForAnnotation:(id <MKAnnotation>)annotation {

  if ([annotation isKindOfClass:[MKUserLocation class]])
    return nil;
  MKPinAnnotationView* pinView = (MKPinAnnotationView*)
 [mapView dequeueReusableAnnotationViewWithIdentifier:
                             @"CustomPinAnnotationView"];
  if (!pinView) {
      pinView = [[[MKPinAnnotationView alloc]
           initWithAnnotation:annotation
           reuseIdentifier:@"CustomPinAnnotation"]
                                          autorelease];
    if ([annotation isKindOfClass:[City class]]) {
      pinView.pinColor = MKPinAnnotationColorRed;
      pinView.draggable =YES;
    }
    else
      pinView.pinColor = MKPinAnnotationColorGreen;
    pinView.animatesDrop = YES;
    pinView.canShowCallout = YES;
  }
  else
    pinView.annotation = annotation;
  return pinView;
}
```

You start by checking to see whether the annotation is a `MKUserLocation`. If it is, you just use the built-in view.

```
if ([annotation isKindOfClass:[MKUserLocation class]])
    return nil;
```

The next thing you do is check to see whether there's a view lying around that you can use. This is the same kind of mechanism you use to reuse Table cells in Chapter 14 (except this time, just to show you, you code the reuse identifier in the message instead of using a constant). If there isn't a view available, you create one, initializing it with the annotation.

```
if (!pinView) {
  pinView = [[[MKPinAnnotationView alloc]
     initWithAnnotation:annotation
     reuseIdentifier:@"CustomPinAnnotation"] autorelease];
```

If the annotation is a `City` object, you set the pin color to red — that's the "customary" color for a destination. You also make the pin draggable. Setting the `draggable` property to `YES` (the default value of this property is `NO`) does what you'd expect it to do: makes an annotation draggable by the user. However, if you do that, the annotation object must also implement the `setCoordinate:` method. Of course, you've already done that when you made annotations an updateable property earlier in this chapter.

If it's not a `City` object — it's an `Airport`, for example — you set the pin color to green, the customary color for an origin. You don't want that pin to be draggable, so you won't bother there with the `draggable` property.

```
if ([annotation isKindOfClass:[City class]]) {
      pinView.pinColor = MKPinAnnotationColorRed;
      pinView.draggable =YES;
    }
    else
      pinView.pinColor = MKPinAnnotationColorGreen;
```

Finally, you make the pin drop animated and allow it to show a callout, which will be handled by the Pin view.

```
pinView.animatesDrop = YES;
pinView.canShowCallout = YES;
```

If there were a reusable Pin view, you assign the annotation and, in either case, return the Pin view.

```
else
  pinView.annotation = annotation;
  return pinView;
```

You also have to add `#import "City.h"` to `MapController.m`.

Of course, if the user does move the destination, it would be nice to change the title and subtitle to the new location. Fortunately, there's another delegate method that gives me a chance to do just that: `mapView:annotationView:didChangeDragState:fromOldState:`.

Listing 15-26 shows the code you need to add to `mapController.m` in order to get this title/subtitle change stuff to work for you.

Listing 15-26: mapView:annotationView:didChangeDragState:fromOldState:

```
- (void)mapView:(MKMapView *)mapView annotationView:
   (MKAnnotationView *)annotationView
   didChangeDragState:(MKAnnotationViewDragState)newState
        fromOldState:(MKAnnotationViewDragState)oldState {

  if (newState == MKAnnotationViewDragStateEnding) {
    reverseGeocoder = [[MKReverseGeocoder alloc]
           initWithCoordinate:
                    annotationView.annotation.coordinate];
    selectedAnnotation = annotationView.annotation;
    reverseGeocoder.delegate = self;
    [reverseGeocoder start];
  }
}
```

The `mapView:annotationView:didChangeDragState:fromOldState:` message is constantly sent to the delegate as the user drags the annotation hither and yon. Although you're constantly being updated, you really don't care where the user has dragged it until the dragging is over.

```
if (newState == MKAnnotationViewDragStateEnding) {
```

When the dragging is over, though, you save the annotation in `selected Annotation` and then create a geocoder to go with the new location.

```
selectedAnnotation = annotationView.annotation;
reverseGeocoder.delegate = self;
[reverseGeocoder start];
```

The geocoder does its thing and updates the `title` and `subtitle` appropriately.

WARNING!

It's my responsibility to warn you about something one more time. If the geocoder does fail (which it sometimes will), what you will see in the annotation is the last location you used to update the annotation `title` and `subtitle`. I'll leave it up to you to add some code to `reverseGeocoder:didFailWith Error:` to deal with that problem.

You can see the result in Figure 15-15.

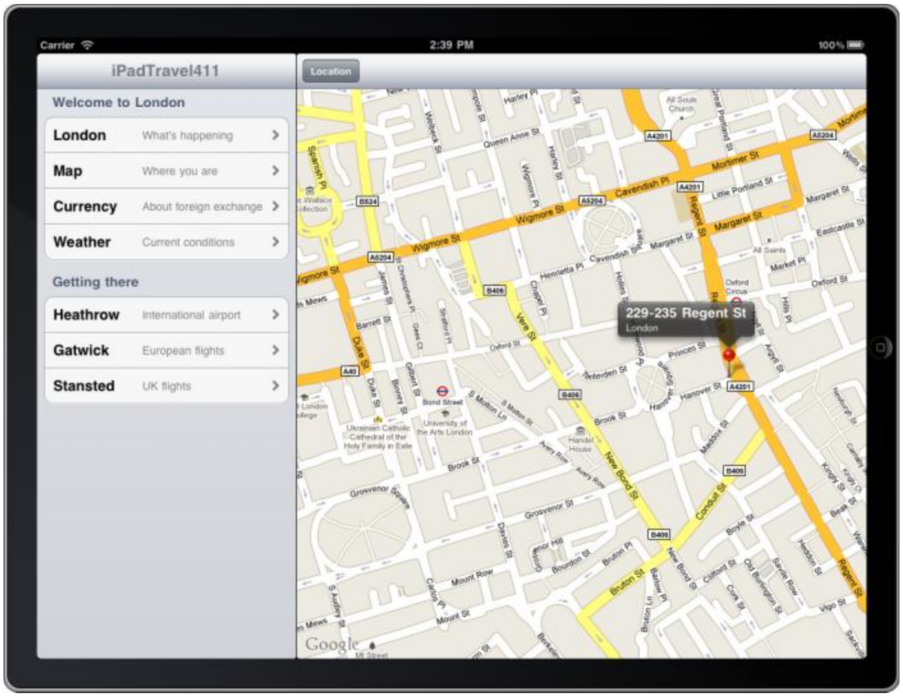


Figure 15-15: Staying at a friend's.

Chapter 16

Adding the Stuff

In This Chapter

- Taking a look at more model content
- Accessing data on the Internet
- Saving and reading files

Now that you've done a stellar job of creating the structure necessary to display the content of your app and you're also able to display a nice Map view when the application is launched, you probably want to be able to see something a bit more substantial actually happen when you press one of those buttons, so it's time to add more content.

In this chapter, I show you how to add three more view controllers to display content — and even how to get that content. I show you how to display content stored as a resource (part of your application) or stored on a Web server, and I even show you how to display a Web page.

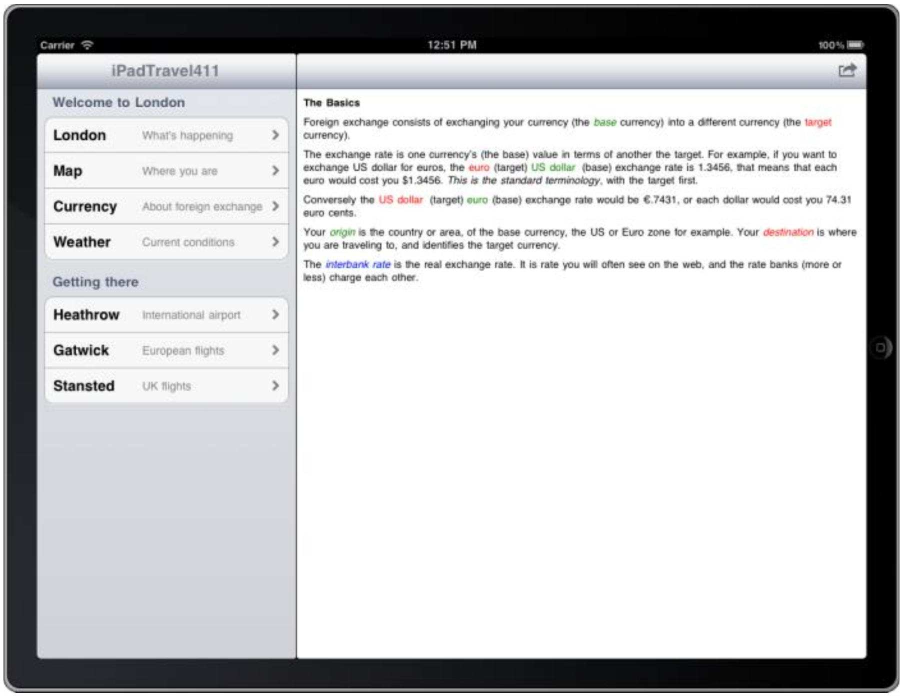
This being London, the first thing you really want to do is check the weather. But before you can do that, you need to find out how the selection mechanism in a Table view works.

Responding to a Selection

At some point, you have to make sure that something actually happens when a user makes a selection. To do that, all you really need to do is implement the `tableview:didSelectRowAtIndexPath:` method to set up a response to a user tap in the Main view. This method, too, is already in the `MasterViewController.m` file, courtesy of the template.

When the user taps a Table view entry, what happens next depends on what you want your Table view to do for you.

You could display some new content in the Detail view, as you can see in Figure 16-1.



The Basics

Foreign exchange consists of exchanging your currency (the *base* currency) into a different currency (the target currency).

The exchange rate is one currency's (the base) value in terms of another the target. For example, if you want to exchange US dollar for euros, the euro (target) US dollar (base) exchange rate is 1.3456, that means that each euro would cost you $1.3456. *This is the standard terminology*, with the target first.

Conversely the US dollar (target) euro (base) exchange rate would be €.7431, or each dollar would cost you 74.31 euro cents.

Your *origin* is the country or area, of the base currency, the US or Euro zone for example. Your *destination* is where you are traveling to, and identifies the target currency.

The *interbank rate* is the real exchange rate. It is rate you will often see on the web, and the rate banks (more or less) charge each other.

Figure 16-1: Displaying new content in the Detail view.

Alternatively, you could display content in the Master view, as you can see in Figure 16-2. I explain how to do that in Chapter 18.

You can actually do a lot of other things — play a movie, play a song, or do anything else the device is capable of — but I don't go into that here.

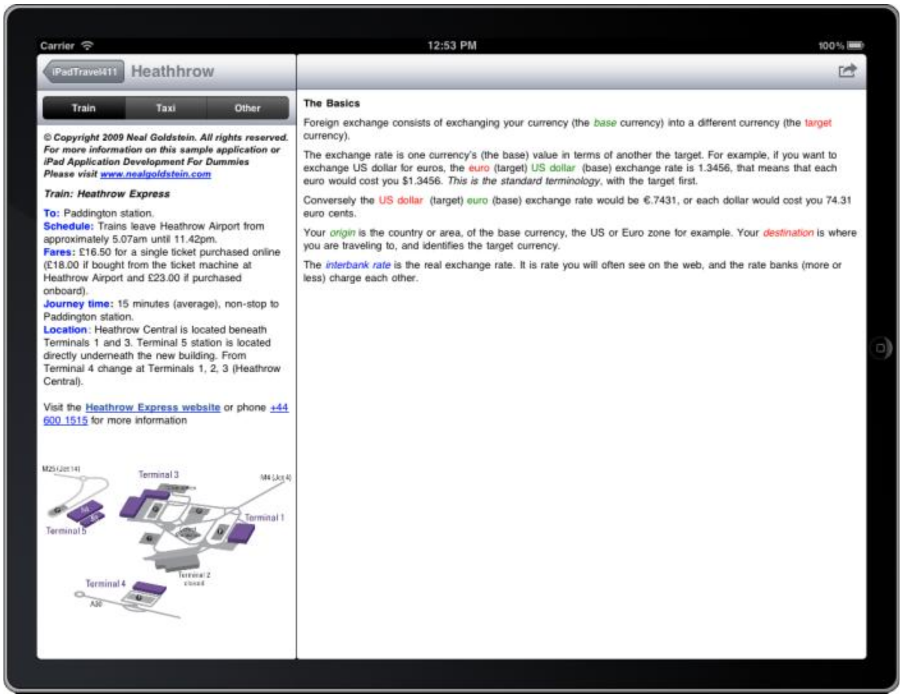
To move from one view to another view, first you need to create a new view controller for that view; then you launch it so it creates and installs the view on the screen.

This is all done in the final Table view method you'll need to work with: `tableView:didSelectRowAtIndexPath:`. The code in Listing 16-1 gives you an overview of what you'll need to do.

Listing 16-1: Selecting a Row

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

  [tableView deselectRowAtIndexPath:indexPath
                                          animated:YES];
  int menuOffset =
              [self menuOffsetForRowAtIndexPath:indexPath];
  DetailViewController *targetController = nil;
```

```
   switch (menuOffset) {
     case 0:
       //do something
       break;
     case 1:
       //do something
       break;
     case 2:
       //do something
       break;
     case 3:
       //do something
       break;
     case 4:
       //do something
       break;
     case 5:
       //do something
       break;
     case 6:
       //do something
       break;
   }
 }
```



© Copyright 2009 Neal Goldstein. All rights reserved. For more information on this sample application or iPad Application Development For Dummies Please visit www.nealgoldstein.com

Train: Heathrow Express

To: Paddington station.
Schedule: Trains leave Heathrow Airport from approximately 5.07am until 11.42pm.
Fares: £16.50 for a single ticket purchased online (£18.00 if bought from the ticket machine at Heathrow Airport and £23.00 if purchased onboard).
Journey time: 15 minutes (average), non-stop to Paddington station.
Location: Heathrow Central is located beneath Terminals 1 and 3. Terminal 5 station is located directly underneath the new building. From Terminal 4 change at Terminals 1, 2, 3 (Heathrow Central).

Visit the Heathrow Express website or phone +44 600 1515 for more information

The Basics

Foreign exchange consists of exchanging your currency (the *base* currency) into a different currency (the target currency).

The exchange rate is one currency's (the base) value in terms of another the target. For example, if you want to exchange US dollar for euros, the euro (target) US dollar (base) exchange rate is 1.3456, that means that each euro would cost you $1.3456. *This is the standard terminology*, with the target first.

Conversely the US dollar (target) euro (base) exchange rate would be €.7431, or each dollar would cost you 74.31 euro cents.

Your *origin* is the country or area, of the base currency, the US or Euro zone for example. Your *destination* is where you are traveling to, and identifies the target currency.

The *interbank rate* is the real exchange rate. It is rate you will often see on the web, and the rate banks (more or less) charge each other.

Figure 16-2: Displaying content in the Master view.

Here's what happens when a user makes a selection in the Main view:

1. **You deselect the row the user selected.**

   ```
   [tableView deselectRowAtIndexPath:indexPath
                                             animated:YES];
   ```

   It stands to reason that if you want your app to move on to a new view, you have to deselect the row where you currently are.

2. **You compute the offset (based on section and row) into the menu array.**

   ```
   int menuOffset =
             [self menuOffsetForRowAtIndexPath:indexPath];
   ```

   You need to figure out where you want your app to land, right?

3. **You do something based on the row selected.**

   ```
   switch (menuOffset) {
   ```

And that's what you do next.

Putting the Map in the Selection Mechanism

To refresh you memory, right now the Map controller is created in the `viewDidLoad` method in `RootViewController.m`. I had you add the code to do that back in Chapter 15.

Now it's time for it to take its rightful place in the `tableView:didSelectRowAtIndexPath:` method.

To start with, you need to fill in the blanks in the current `tableView:didSelectRowAtIndexPath:` method I highlight in Listing 16-1 by replacing that code with the code in Listing 16-2.

Listing 16-2: Selecting the Map

```
- (void)tableView:(UITableView *)tableView
       didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

  [tableView deselectRowAtIndexPath:indexPath
                                             animated:YES];
  DetailViewController *targetController = nil;
  int menuOffset =
```

```
            [self menuOffsetForRowAtIndexPath:indexPath];
  switch (menuOffset) {
    case 0:
      //do something
      break;
    case 1:
      targetController =
       [[menuList objectAtIndex:menuOffset]
                              objectForKey:kControllerKey];
      if ([targetController isKindOfClass:
                                         [NSNull class]]) {
        targetController = [[MapController alloc]
           initWithDestination:kAppDelegate.destination];
        [[menuList  objectAtIndex:menuOffset]
                         setObject:targetController
                                     forKey:kControllerKey];
        [targetController release];
      }
      break;
    case 2:
      //do something
      break;
    case 3:
      //do something
      break;
    case 4:
      //do something
      break;
    case 5:
      //do something
      break;
    case 6:
      //do something
      break;
  }
 [[kAppDelegate splitViewController] setViewControllers:
    [NSArray arrayWithObjects:self.navigationController,
                                  targetController, nil]];
 kAppDelegate.splitViewController.delegate =
                                         targetController;
}
```

Here’s what happens when a user makes a selection in the Main view:

1. **You deselect the row the user selected.**

   ```
   [tableView deselectRowAtIndexPath:indexPath
                                        animated:YES];
   ```

   It stands to reason that if you want your app to move on to a new view, you have to deselect the row where that’s currently selected.

2. **You compute the offset (based on section and row) into the menu array.**

   ```
   int menuOffset =
           [self menuOffsetForRowAtIndexPath:indexPath];
   ```

   The `tableview:didSelectRowAtIndexPath:` method is called when the user taps a row in a section. The `indexPath` argument has the section and row information of which entry was tapped. You recall from Chapter 14 that `menuOffsetForRowAtIndexPath:` will compute that for you.

3. **You're going to use a `switch` statement to get to the right controller:**

   ```
   switch (menuOffset) {
   ```

4. **You check to see whether the controller for that particular view has already been created.**

   ```
   targetController =
         [[menuList objectAtIndex:menuOffset]
                              objectForKey:kControllerKey];
   if ([targetController isKindOfClass:[NSNull class]]) {
   ```

   If you recall, when you set up the `menuList` in Chapter 14, you initialized the controller objects with `NSNull`, a singleton object, with a single class method `null`, that you can use to represent null values in collection objects. You use the `NSObject isKindOfClass:` method to check to see if it's a class of that type.

5. **If no controller exists, you create and initialize a new controller.**

   ```
   targetController = [[MapController alloc]
          initWithDestination:kAppDelegate.destination];
   ```

6. **If you created a new view controller, you save a reference to the newly created controller in the dictionary for that row and then release the controller because it's retained by the `menuList`.**

   ```
   [[menuList  objectAtIndex:menuOffset]
                   setObject:targetController
                                  forKey:kControllerKey];
    [targetController release];
   ```

7. **You then replace the detail controller in the Split view controller with your new view controller and then make this new view controller a delegate of the Split view controller.**

   I explain in Chapter 14 that the `UISplitViewController` class deals with view controllers that simply manage the presentation of two side-by-side view controllers — it is, in this respect, a container controller. Using this class, you create a view controller on the left (the *Master view,* as it's called), which presents a list of items, and one on the right, which presents the details, or content, of the selected item (the *Detail view,* as it's called).

```
[[kAppDelegate splitViewController]
  setViewControllers:[NSArray arrayWithObjects:
   self.navigationController, targetController, nil]];
kAppDelegate.splitViewController.delegate =
                                    targetController;
```

8. **You need to modify `viewDidLoad` — go into `viewDidLoad` and delete the previous code you used to create the `mapController` (the bold, underlined, and italic code, affectionately known as BUI) and then add the line of code in bold:**

```
- (void)viewDidLoad {

  [super viewDidLoad];

...

DetailViewController *mapController =
[[MapController alloc]
                    initWithDestination:destination];


[[kAppDelegate splitViewController setViewControllers:
 [NSArray arrayWithObjects:self, mapController, nil]];
kAppDelegate.splitViewController.delegate =
                                      mapController;
NSIndexPath *indexPath =
         [NSIndexPath indexPathForRow:1 inSection:0];
[self tableView:((UITableView *) self.tableView)
                  didSelectRowAtIndexPath:indexPath];
```

You take advantage of the same mechanisms that are used when the user taps an entry in the Table view. You invoke the `didSelectRowAtIndex-Path:` method, which already knows how to display a particular view represented by the `indexPath` — that is, section and row. You create the index path you need by sending the `indexPathForRow:inSection:` message to the `NSIndexPath` class.

If you wanted to refine this a bit, you could also use constants for the section and row that represent the initial view. This makes it easier later if you want to change which view the user sees at app launch. Of course, after that, unless the app is purged from the background, the app will resume where it was. I show you how to have the app resume where it was (even if the app has been purged) in Chapter 19.

How's the Weather Over There?

If the user selects Weather from the Main view, what the user sees *does* depend on whether the device is online or in stored data mode. (I explain that in Chapter 19.) If the device is online, the user sees a Web page from a Web site with the weather information, as illustrated in Figure 16-3. When in stored data mode, the user gets a message stating that weather data is unavailable when offline.

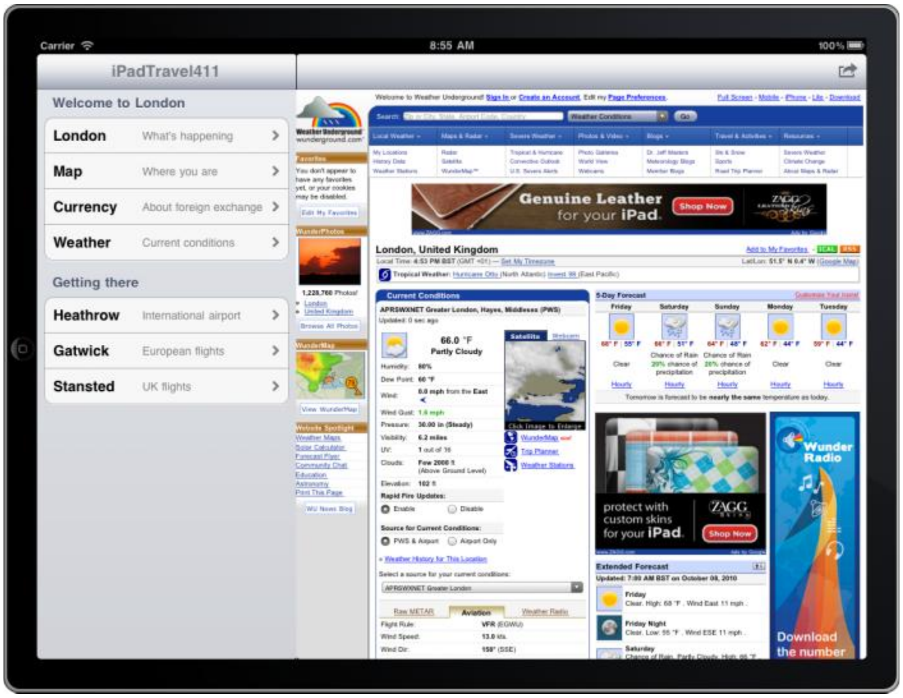


Figure 16-3: Gee, rain expected.

To begin this process, you need to create a view controller that will display a Web site in its view. To make *that* happen, you need to code your view controller interface, your implementation files, and your nib files.

Adding the controller and nib file

So many files, so little time. Actually, after you get a rhythm going, cranking out the various view controller, nib, and model files necessary to fill your application architecture with content isn't *that* much work. And even though I want to start with what happens when the user taps Weather (because it allows me to also explain a bit about navigating between Detail views in your program), now is as good a time as any to create the additional controllers I'll have you implement in this chapter.

Okay, check out how easy it is to come up with the view controller and nib files:

1. **In the IPadTravel411Project window, select the View controllers group and then choose File➪New File from the main menu (or press ⌘+N) to get the New File dialog.**
2. **In the leftmost pane of the dialog, first select Cocoa Touch Classes under the iOS heading, then select the *UIViewController subclass* template in the topmost pane, and then make sure that the following are selected:**
    - With XIB for User Interface
    - Targeted for iPad
3. **Click Next.**

   You see a new dialog asking for some more information.
4. **Enter** WeatherController **in the File Name field and then click Finish.**

   Of course, if you look at the choices in the Master view controller, you see there's more to life than complaining about the weather. So, to get it out of the way, you should add the remaining controllers you'll be using.
5. **Repeat Steps 1 through 4 for `CityController` and `CurrencyController`.**

   When you're done, in the View controllers group in your Groups & Files list, you should see `CityController.h`, `CityController.m`, `CurrencyController.h`, `CurrencyController.m`, `WeatherController.h`, and `WeatherController.m` in addition to what you already have there.

The next thing you need to do is take care of some plumbing in each of the `Controller.h` files, and after that, you need to set up the nib files.

You need to import `Destination` (you'll be using it to get the data needed to be displayed in the view) and `DetailViewController` because you're also going to make each of the classes a subclass of `DetailViewController`, just as you did the `MapController`, and you need to tell the compiler about the `Destination` class.

You also need to make the class a `WebViewDelegate` (I explain why shortly) and add three instance variables: `destination`, the `webView IBOutlet` (just as you added `destination` and `mapView` to the `MapController`), and `backButton`, which I explain in the "Cruising the Web" section, later in this chapter. You also declare an initialization method, `initWithDestination:`, again, just like you do with the `MapController`.

When you're done, `WeatherController.h` should look like Listing 16-3. (The stuff you add is all shown in bold, and you should delete the BUI code.) Do the same for the `City` and `Currency` controllers as well.

Listing 16-3: WeatherController interface

```
#import "DetailViewController.h";
@class Destination;


@interface WeatherController : UIViewController
          DetailViewController <UIWebViewDelegate> {

           Destination  *destination;
  IBOutlet UIWebView    *webView;
           UIButton     *backButton;
}
- (id) initWithDestination:(Destination *) theDestination;
- (void) computeFramesForOrientation:
             (UIInterfaceOrientation)interfaceOrientation;

@end
```

Be sure to save the file.

After it's saved — and only then — Interface Builder can find the new outlet.

Setting up the nib file

For the iPadTravel411 application, you want to use a `UIWebView` to display the Web site or any other data you're after. (For the reasoning behind that choice, check out Chapter 13.) You'll set up the `UIWebView` by using Interface Builder, but you'll also need a reference to it from the `WeatherController` so it can load the Web site you want. To do that, you need to create an *outlet* (a special kind of instance that can refer to objects in the nib) in the view controller, just as you did back in Chapter 14 when you were working on the `MapController`. (You took care of that outlet creation business when you made the modifications to the interface files for `WeatherController`, `CityController`, and `CurrencyController`.)

Now it's time to make the necessary connections in Interface Builder so the outlet reference will be filled in automatically when your application is initialized.

Here's how you to connect the outlets — it's the same thing you did in Chapter 14 to set up the `MapController`, so if you're a little hazy, you might want to go back and review what you did there:

1. **Use the Groups & Files list on the left in the Project window to drill down to the `WeatherController.xib` file; then double-click the file to launch it in Interface Builder.**

   TIP

   If the Attributes Inspector window is not open, choose Tools➪Inspector or press ⌘+Shift+1. If the View window isn't visible, double-click the View icon in the `WeatherController.xib` window.

   If for some reason you can't find the `WeatherController.xib` window (you may have minimized it, whether by accident, on purpose, or whatever), you can get it back by choosing Window➪WeatherController.xib or whichever nib file you're working on.

2. **Select File's Owner in the `WeatherController.xib` window.**

   Its type should already be set to `WeatherController`. If not, retrace your steps to see where you may have made a mistake.

   You need to be sure that the File's Owner is `WeatherController`. You can set the File's Owner from the Class drop-down menu in the Identity Inspector.

3. **Drag in a toolbar from the Library to the top of the View window. Make sure it is a toolbar and not a navigation bar. It looks like it belongs on the bottom of the view, but on the iPad it can be at the top. Select and delete the item button in the toolbar; you won't need it.**

   Be sure to do all this work in portrait mode, as it is in Figure 16-4.

4. **In the View Attributes Inspector window, be sure to deselect Autoresize Subviews.**

5. **Drag in a Web View below the toolbar and resize it to the size of the view remaining after you added the toolbar.**

   When you're done, your screen should look like Figure 16-4. (In Figure 16-4, I have the View selected and that's what's showing in the View Attributes Inspector window.)

6. **Back in the `WeatherController.xib` window, right-click File's Owner to call up a connections panel with a list of connections.**

   You can get the same list using the Connections tab in the Attributes Inspector.

7. **Drag from the little circle next to the `webView` outlet in the list onto the Web view.**

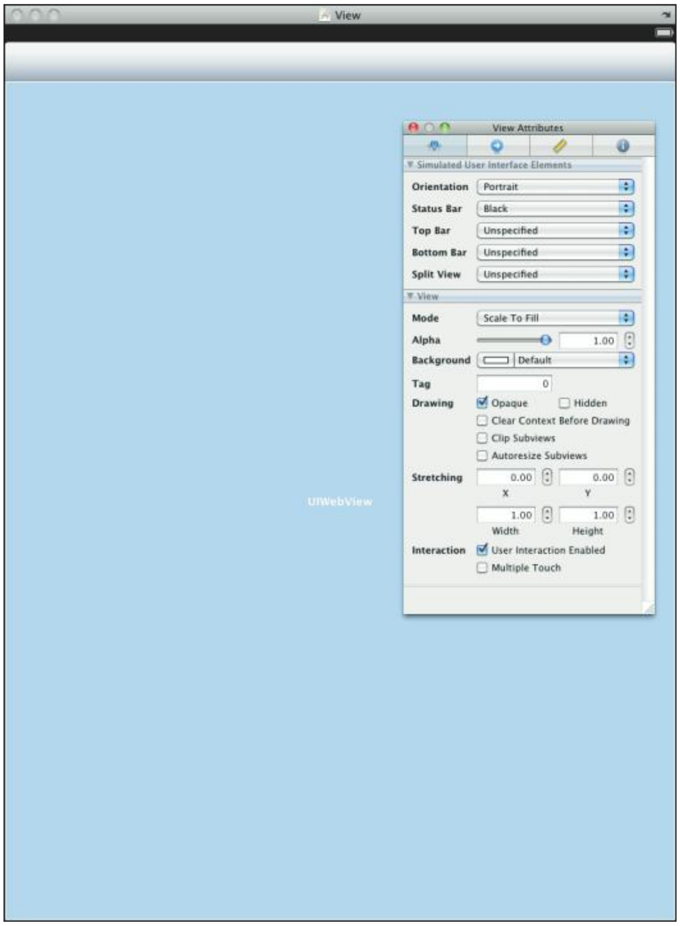
   Doing so connects the `webView` outlet of the `WeatherController` to the Web view.

8. **Drag from the little circle next to the `toolbar` outlet in the list onto the toolbar you dragged in from the Library.**

   Doing so connects the `toolbar` outlet of the `WeatherController` to the toolbar.

9. **Save the file.**

   The `WeatherController` now has its outlet to the Web view and toolbar connected, but you won't yet be able to receive the delegate messages as a `WebViewDelegate` — I show you an alternative way to do that in the next section where I also show you what methods you need to implement.



▼ Simulated User Interface Elements

Orientation	Portrait
Status Bar	Black
Top Bar	Unspecified
Bottom Bar	Unspecified
Split View	Unspecified

▼ View

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

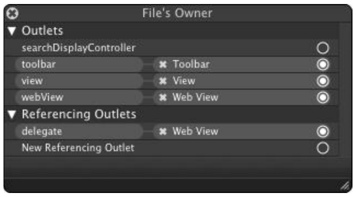
Figure 16-4: The Interface Builder windows.

REMEMBER

Only after the file's saved will the changes you made be reflected in your application.

When you're done, the connections panel should look like Figure 16-5.

10. **And, as you may have guessed, repeat Steps 1 to 9 for `CityController.xib` and `CurrencyController.xib`.**



File's Owner

▼ Outlets

searchDisplayController	
toolbar	Toolbar
view	View
webView	Web View

▼ Referencing Outlets

delegate	Web View
New Referencing Outlet	

Figure 16-5: WeatherController connections all in place.

Of course, even though it's nice that you have all these controllers, you still need to have them add some actual content. At this point, I have you focus on doing that in `WeatherController`.

Loading the Web View

First start with an initialization method. Add the code in Listing 16-4 to `WeatherController.m`. You also need to add `#import "Destination.h"` to the file to be able to use it.

Listing 16-4: Initialize WeatherController

```
- (id) initWithDestination:(Destination *)
                                          theDestination {

  if (self = [super initWithNibName:
                          @"WeatherController" bundle:nil]) {
    destination = theDestination;
  }
  return self;
}
```

What do these few lines of code do for you?

1. **First it invokes the superclass's `initWithNibName:bundle:` method:**

   ```
   [super initWithNibName:@"WeatherController"
                                          bundle:nil]
   ```

   The first thing *this* method does is invoke its superclass's initialization method. I pass it the nib filename (the one I just created in a previous section) and `nil` as the bundle, telling it to look in the main bundle.

   Note that the message to `super` precedes the initialization code added in the method. This sequencing ensures that initialization proceeds in the order of inheritance. Calling the superclass's `initWithNibName:bundle:` method initializes the controller, loads and initializes the objects in the nib file (views and controls, for example), and then sets all its outlet instance variables and Target-Action connections for good measure.

2. **The `init…:` methods all return a pointer to the object created.**

   While not the case here, the reason you assign whatever comes back from an `init…:` method to `self` is that some classes actually return a different class than what you created. The assignment to `self` becomes important if your class is derived from one of those kinds of classes. Keep in mind as well that an `init…:` method can also return `nil` if there's a problem initializing an object. If you're creating an object where that is a possibility, you have to take that into account. (Both of those situations are beyond the scope of this book.)

3. **After the superclass initialization is completed, the `Weather Controller` is ready to do its own initialization, including saving the `aDestination` argument to the `destination` instance variable.**

To load the Web view, the place to start is in the method `viewDid-Load`. This method was included for you in `WeatherController.m` by the `UIViewController` subclass template (albeit, commented out). Simply uncomment this method and add the code in Listing 16-5 to `WeatherController.m`.

Listing 16-5: viewDidLoad

```
- (void)viewDidLoad {

  [super viewDidLoad];
  webView.delegate = self;
  webView.scalesPageToFit = YES;
  [webView loadRequest:[NSURLRequest
                 requestWithURL:[destination weather]]];
}
```

The first thing you do in Listing 16-5 is make `WeatherController` the Web view's delegate. In Chapter 15, I have you do this in Interface Builder, but it's easy to forget to do, and doing it in code documents the fact that you did do it.

Next, I assign a Web view property `scalesPageToFit` to `YES`, which tells the Web view to scale the page to fit the view (so you don't have to scroll all over the place).

Then, as you can see, you send a message to the `Destination` object (which you passed in when you initialized the `WeatherController` in the previous section) to find out where the data that the Web view needs is located. This location is returned in the form of an `NSURL`, an object that includes the utilities necessary for downloading files or other resources from Web and FTP servers.

This method then creates an `NSURLRequest` that the Web view needs in order to be able to load the data from the `NSURL`. The `NSURLRequest` is what the WeatherController needs to send to the Web view in the load-Request: message, which tells it to load the data associated with that particular NSURL.

The NSURLRequest class encapsulates a URL as well as any protocol-specific properties, all the time keeping things nicely protocol-independent. It also provides a number of other things that are out of this book's scope — including the set of classes and protocols that provide the underlying capability for an application to access the data specified by a URL. Seeing that this is the preferred way to access files both locally and on the Internet, at some point you should explore the URL-loading system on your own.

This small task is all it takes to access the Web site — with the exception of the code in Destination that returns the NSURL. To do that bit of business, add the method in Listing 16-6 to Destination.m. (Add the declaration to Destination.h as well.)

Listing 16-6: The weather method

```
- (NSURL *) weather {

  NSURL *url =  [NSURL URLWithString:
    @"http://www.wunderground.com/global/stations/
                                                03772.html"];
  return url;
}
```

This is the same mechanism you used in the MapController where the Destination object returns the data (or the address of the data) that the view controller needs.

Be sure to type the URL string on one line.

That URL you see is one I use for weather for London from www.wunderground.com. Of course, these things change from time to time, and it may or may not work when you try it. If not, check my Web site (www.nealgoldstein.com) to find out what I'm currently using. Again be sure to type the URL string all on one line in Xcode.

Finally, just as you did with the MapController, you're going to have to manage the toolbar and view sizes in WeatherController. Add the code in Listing 16-7 to WeatherController.m and delete the bold, underlined and italic code shown in Listing 16-8 from WeatherController.m.

Listing 16-7: Adding computeFramesForOrientation:

```
- (void) computeFramesForOrientation:
           (UIInterfaceOrientation)interfaceOrientation {

  [self computeFrames:webView forOrientation:interface
           Orientation];
}
```

Listing 16-8: Deleting shouldAutorotateToInterfaceOrientation:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
             (UIInterfaceOrientation)interfaceOrientation {
    //  Overriden to allow any orientation.
  return YES;
}
```

If you do forget to do this, the super view (`DetailViewController`) will never get the `shouldAutorotateToInterfaceOrientation:` message and then send the `computeFramesForOrientation:` message. As a result, it will not resize correctly.

Cruising the Web

Although the Web page I've chosen for my Weather view (refer to Figure 16-3) does have a lot of weather-y information, there's also quite a bit more information available using the links on the page. So, in my design, I wanted to be able to tap a link on the page to get that additional information. It's actually easy to do in a Web view (in fact, the Web view does it for you), but you need to add a Back button after the user goes someplace in order to get back to this main page. So I have you create another button and label it Weather so the user knows he or she can use it to get back to the previous view.

Of course, you don't want to display a Back button if there's nothing to go back to — you want to display the button only after the user has clicked a link, and then remove it when the user is back to the original Web page you first displayed.

All this is accomplished in the `WebView:shouldStartLoadWithRequest:navigationType:` method. This message is sent to the Web view's delegate at exactly the right time — before the view starts to load but after the user has selected the link. Add the code in Listing 16-9 to `WeatherController.m`.

Listing 16-9: When the User Touches a Link

```
- (BOOL)webView:(UIWebView *)webView
    shouldStartLoadWithRequest:(NSURLRequest *)request
  navigationType:(UIWebViewNavigationType)navigationType {

  if (navigationType ==
          UIWebViewNavigationTypeLinkClicked) {
    if (!backButton) {
      backButton =
      [[UIBarButtonItem alloc] initWithTitle:@"Weather"
          style:UIBarButtonItemStyleBordered target:self
                               action:@selector(goBack:)];
      NSMutableArray *items =
                            [[toolbar items] mutableCopy];
      [items addObject:backButton];
      [toolbar setItems:items animated:YES];
      [backButton release];
    }
  }
  return YES;
}
```

Point for point, here's what you're doing with this code:

1. **First, check to see whether the user has touched an embedded link.**

```
if (navigationType ==
              UIWebViewNavigationTypeLinkClicked) {
```

2. **If he or she has, create and add the Back button (if there isn't one already there).**

```
if (!backButton) {
  backButton =
    [[UIBarButtonItem alloc] initWithTitle:@"Weather"
      style:UIBarButtonItemStyleBordered target:self
                             action:@selector(goBack:)];
```

   In this method, you allocate the button and then assign it to an instance variable `backButton`. The `action:@selector(goBack:)` argument is the standard way to specify Target-Action. It says when the button is tapped, send the `goBack:` message to the `target: self`, which is the `WeatherController`. I show you how to implement this shortly.

3. **Add the Back button to the toolbar.**

```
NSMutableArray *items = [[toolbar items] mutableCopy];
    [items addObject:backButton];
    [toolbar setItems:items animated:YES];
```

4. **Return `YES` to tell the Web view to load from the Internet.**

Next, add the delegate method `goBack:` to `WeatherController.m`. (You specified `goBack:` as the selector when you created the Back button.) Listing 16-10 has the details.

Listing 16-10: goBack to Where You Once Belonged

```
- (IBAction) goBack:(id)sender{

    [webView goBack];
}
```

The `UIWebView` actually implements much of the behavior you need here. It sends the Web view a message to load the previous page.

Finally, you want to get rid of the Back button when you're displaying the original page. The code to do that is in Listing 16-11.

Listing 16-11: You Don't Need the Back Button Any Longer

```
- (void)webViewDidFinishLoad:(UIWebView *) aWebView {

   if (([aWebView  canGoBack] == NO) && (backButton)) {
     NSMutableArray *items = [[toolbar items] mutableCopy];
     [items removeObject:backButton];
     backButton = nil;
     [toolbar setItems:items animated:YES];
   }
}
```

This message is sent after the view has loaded. At this point, you check to see whether there's anything to go back *to* — the Web view keeps track of those sorts of things. If not, you remove the button from the toolbar.

TIP

If you were to look in the `UIWebViewDelegate` Protocol Reference, you would find this method:

```
- (void)webViewDidFinishLoad:(UIWebView *)webView
```

If you add it *that way,* when you compile your app, you get a compiler warning that says

```
Local declaration of 'webView' hides instance variable
```

This warning is referring to your instance variable and the argument name in `webViewDidFinishLoad:`. Although this makes no difference because both `webView` variables refer to the same object, you should get rid of the warning. You can do one of two things. You can change your instance variable name, or you can simply change the name in the method:

```
- (void)webViewDidFinishLoad:(UIWebView *) aWebView
```

This is what I did in Listing 16-11.

WARNING!

That being said, the Apple Human Interface Guidelines say, “In addition to displaying Web content, a Web view provides elements that support navigation through open Web pages. Although you can choose to provide Web page navigation functionality, it’s best to avoid creating an application that looks and behaves like a mini Web browser.” As far as I’m concerned, making it possible to select links in a Web view and return back to the originating page doesn’t do that, but if you really didn’t want to enable the user to follow links, Listing 16-12 shows you how to disable links. (As you’ll see, there will be other times you’ll want to disable links, like when you’re in stored data mode, and I explain that in Chapter 19.)

If you’ve decided to follow Apple’s suggestion and aren’t making your app act as a mini browser, you have to disable the links that are available in the content. You can do that in the `shouldStartLoadWithRequest:` method in the `WeatherController.m` file by coding it as shown in Listing 16-12.

Listing 16-12: Disabling Links

```
- (BOOL)webView:(UIWebView *) webView
       shouldStartLoadWithRequest:(NSURLRequest *) request
       navigationType:
             (UIWebViewNavigationType)navigationType {

  if (navigationType ==
                        UIWebViewNavigationTypeLinkClicked)
    return NO;

  else return YES;
}
```

Responding to a Selection

At this point, you probably can’t wait to see the results of your handiwork — the weather in London. You have to go back to `tableView:didSelectRowAtIndexPath:` in `RootViewController.m` to make that happen.

Before you do though, add the following import statements so that you can initialize and create the controllers you just created:

```
#import "WeatherController.h"
#import "CityController.h"
#import "CurrencyController.h"
```

Now that you have the `WeatherController` set up, you can allow the user to select it in the `RootViewController`.

All you need to do is replace `case : 3` in `tableView:didSelectRowAtIndexPath:` which now looks like this:

```
case 3:
  //do something
  break;
```

with this:

```
case 3:
  targetController = [[menuList objectAtIndex:menuOffset]
                              objectForKey:kControllerKey];
  if ([targetController isKindOfClass:[NSNull class]]) {
    targetController = [[WeatherController alloc]
            initWithDestination:kAppDelegate.destination];
    [[menuList  objectAtIndex:menuOffset]
        setObject:targetController forKey:kControllerKey];
    [targetController release];
  }
  break;
```

This is exactly what you did to create the `MapController` when you did it at the end of the `viewDidLoad` method.

If you were to compile and run your app in landscape mode only, it would work exactly as you expect. You can select Map or Weather, you can go to a map location, click a link on the Weather Web page, go back to Map, go back to the same page in Weather, and then return to the main Weather view. The only thing you would notice is that the Web view really isn't scaled to fit the view.

The reason for that is you never did send a message to adjust the view sizes that I explain in Chapter 15. When the application is launched, the `shouldAutorotateToInterfaceOrientation:` message is sent, which then sends the `computeFramesForOrientation:` message, and so on. But when you make a selection in the Master view, no such message is sent. You're going to have to do that on your own. (To refresh your memory on this topic, review the section on managing the views in Chapter 15.)

That's not so hard.

But if you were to try to do almost anything in portrait view, you would have a real mess on you hands.

The Split view controller displays both the Master view and the Detail view in landscape orientations, but only the Detail view controller is displayed in portrait orientations. When the Master view controller is hidden, in the `split`

ViewController:willHideViewController: withBarButtonItem: forPopoverController: method, you add a button that is *supplied to you* by the Split view controller.

I explain this in Chapter 14. The user can then use that button to display the Master view controller in a popover. All this is accomplished by making the Detail view controller a delegate of the Split view controller, which sends its delegate messages at the appropriate times to add and remove the button.

This works just fine for the first Detail view controller you see in the portrait view. However, because it's inevitable that selecting a new view means you also select a new view controller (as well as its toolbar), you have to move the Root List button from where it was to the new toolbar.

Although this is some (not a lot of) additional work, by the time you're done, you'll have an appreciation of how this Split view controller works and be able to modify any application that isn't a boring-off-the-shelf-created-from-a-template app to meet your needs.

To make all this work, all you need to do is add the code in bold in Listing 16-13 to the end of `tableView:didSelectRowAtIndexPath:` in `RootViewController.h`. More specifically, add it right after the new view controller code you just added to the Detail view of the Split view controller.

Of course there are a few other methods to write as well, but you'll get to that).

Listing 16-13: Adding to tableView:didSelectRowAtIndexPath:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

…
[kAppDelegate splitViewController] setViewControllers:
    [NSArray arrayWithObjects:self.navigationController,
                                  targetController, nil]];
kAppDelegate.splitViewController.delegate =
                                         targetController;
[targetController computeFramesForOrientation:
                                 [self appOrientation]];
  if (currentController)
                     [self setupToolbar:targetController];
  if (currentController.popoverController) {
    targetController.popoverController =
                      currentController.popoverController;
    [currentController.popoverController
                              dismissPopoverAnimated:YES];
  }
  currentController = targetController;
}
```

Now I lead you through this slowly. After you're done with this, you'll really understand how this whole Split view controller popover business works.

Computing the view and toolbar sizes

The first thing you do is send the `computeFramesForOrientation:` message to the controller you just created. (I explain this in Chapter 15, so if you're a bit hazy on what's going to happen, you might want to review that chapter.) The only difference here is that the message is not being sent in response to a device rotation, so you'll have to pass in the current state of the device.

You get that by sending the `appOrientation` message that you'll find in Listing 16-14. (You have to add it to `RootViewController.m` and add its declaration to `RootViewController.h`.)

Listing 16-14: appOrientation

```
- (UIInterfaceOrientation) appOrientation {

  UIInterfaceOrientation orientation;
  if  ([[currentController.toolbar items]count]) {
    UIBarButtonItem * existingButton =
       [[currentController.toolbar items]objectAtIndex:0];
    if ([existingButton.title
                              isEqualToString:@"Root List"])
         orientation = UIDeviceOrientationPortrait;
    else orientation = UIDeviceOrientationLandscapeRight;
  }
  else orientation = UIDeviceOrientationLandscapeRight;
  return orientation;
}
```

You have a couple of ways to get the device orientation. For example, you have a nice little property in the `UIViewController` class by the name of `interfaceOrientation`. Unfortunately, that property, at least in this version of the SDK, doesn't get set until after the first device rotation. That's good, but not good enough.

There's also an `orientation` property of the `UIDevice` class. The only problem is that you have to enable orientation notifications on the device and then end them, and that get's a bit complicated.

Fortunately, for your purposes, there's an easier way: Have the code look to see whether there are bar button items and whether the first bar button item has the title "Root List." (Root List is the title you give it in the `DetailViewController` implementation of the `splitViewController:`

`willHideViewController:withBarButtonItem: forPopover Controller:` method.)

You are using the button title that you set back in the `splitViewController: willHideViewController:withBarButtonItem: forPopover Controller:` method. If you changed it there, you'll have to change it in here as well.

```
if  ([[currentController.toolbar items]count]) {
    UIBarButtonItem *existingButton =
       [[currentController.toolbar items]objectAtIndex:0];
    if ([existingButton.title
                            isEqualToString:@"Root List"])
```

If you know there are bar button items and that the first bar button item has the title Root List, you know the app is in portrait mode, and you set the orientation variable to that.

```
orientation = UIDeviceOrientationPortrait;
```

Portrait mode is the one you care about, so for all other orientations you just set the mode to `UIDeviceOrientationLandscapeRight` and return that, as follows:

```
    else orientation = UIDeviceOrientationLandscapeRight;
  }
  else orientation = UIDeviceOrientationLandscapeRight;
  return orientation;
}
```

You then go through the same code as you did previously to set up the view and toolbar sizes.

Setting up the toolbar

With your sizing issues out of the way, you next look at the buttons in the toolbar.

If your selection is taking place from the popover, you now have a button in the toolbar that got inserted in the current view with the title Root List. This was the button passed into the `DetailViewController`'s implementation of the `splitViewController:willHideViewController:withBar ButtonItem:forPopoverController:` delegate method. This button is all set up to display the popover, and you need to move that button from the current view controller's toolbar to the new view controller toolbar you're displaying.

REMEMBER

You may have plans for other buttons on the new view controller's toolbar; the logic in this method would take care of those as well.

If the selection isn't taking place from the popover, you need to look at the toolbar as well. If the app was in portrait mode the last time the view was displayed, then there will be an old Root List button. You need to remove that before you display the view.

To deal with this particular scenario, start by seeing whether there's a current controller (refer back to Listing 16-13). (This is not an initial application launch.) If there is one, you need to set up the toolbar.

```
if (currentController)
                     [self setupToolbar:targetController];
```

To do that, add the code in Listing 16-15 to `RootViewController.m`. You also have to add the declaration to `RootViewController.h`.

Listing 16-15: Setting Up the Toolbar

```
- (void) setupToolbar:(DetailViewController *)
                                     targetController  {

  NSMutableArray *items =
           [[targetController.toolbar items] mutableCopy];
  if (!items) items =
                  [NSMutableArray arrayWithCapacity:2];
  if ([self appOrientation] ==
                          UIDeviceOrientationPortrait) {
    if ([[targetController.toolbar items] count]) {
      UIBarButtonItem *buttonItem =
        [[targetController.toolbar items]objectAtIndex:0];
      if ([buttonItem.title isEqualToString:@"Root List"])
        [items removeObjectAtIndex:0];
    }
    [items insertObject:[[currentController.toolbar items]
                            objectAtIndex:0] atIndex:0];
    [targetController.toolbar
                           setItems:items animated:YES];
  }
  else {
    if ([[targetController.toolbar items] count]) {
      UIBarButtonItem *buttonItem =
        [[targetController.toolbar items]objectAtIndex:0];
      if ([buttonItem.title
                        isEqualToString:@"Root List"]){
        [items removeObjectAtIndex:0];
        [targetController.toolbar
```

```
                                    setItems:items animated:YES];
      }
    }
  }
}
```

The first thing you do in Listing 16-15 is see whether any toolbar items are available. (There could very well be one lying around if you've previously created this view controller.) If there is one, you copy the array; if not, you create a new one.

```
NSMutableArray *items =
           [[targetController.toolbar items] mutableCopy];
  if (!items) items =
                    [NSMutableArray arrayWithCapacity:2];
```

The next thing you do is check to see whether the app is in portrait orientation.

```
if ([self appOrientation] ==
                           UIDeviceOrientationPortrait) {
```

If the app is in portrait mode, check to see whether any buttons are on the toolbar. (There won't be if you have just created the controller and/or if the controller was created in landscape.) Actually, this is the general case as well as being the case in this chapter. (In the next chapter, you'll be adding a Print button to the toolbar when it's created, so there will always be a button in there.)

```
if ([[targetController.toolbar items] count]) {
```

If there are buttons, you want to remove the previous Root List button if there is one — it's no longer valid.

```
UIBarButtonItem *buttonItem =
        [[targetController.toolbar items]objectAtIndex:0];
      if ([buttonItem.title isEqualToString:@"Root List"])
        [items removeObjectAtIndex:0];
```

Next, you add the current button from the current controller. As I said, this button is all set up to display the popover; you need to move that from the current view controller's toolbar to the new view controller toolbar you want to display.

```
[items insertObject:[[currentController.toolbar items]
                             objectAtIndex:0] atIndex:0];
    [targetController.toolbar
                            setItems:items animated:YES];
```

If the app isn't in portrait mode, you want to see whether the new toolbar has the Root List button. (It'll be there if the app was in portrait mode the last time you displayed it.) If the Root List button is there, go ahead and remove it. (There is no need for a Root List button in landscape mode.)

```
if ([[targetController.toolbar items] count]) {
      UIBarButtonItem *buttonItem =
        [[targetController.toolbar items]objectAtIndex:0];
      if ([buttonItem.title
                           isEqualToString:@"Root List"]){
        [items removeObjectAtIndex:0];
        [targetController.toolbar
                             setItems:items animated:YES];
```

Managing the popover

The final thing you do is check to see whether there's a popover associated with the current controller (the one you're replacing). The Split view controller has associated this popover with a Detail view controller, and because you're replacing that Detail view controller, you need to now update the new one with the popover reference. (Again, refer to Listing 16-13.)

```
if (currentController.popoverController) {
```

The `popoverController` is a `DetailViewController` property that's set when the `UIPopoverControllerDelegate` messages are sent. (For more on these popover messages, see Chapter 14.)

If a popover is associated with the current controller, you want to update the new controller with it.

```
if (currentController.popoverController) {
    targetController.popoverController =
                     currentController.popoverController;
```

Assuming again that the popover is present, after this bit of updating you want to go ahead and dismiss it. Updating the `popoverController` instance variable and then dismissing the Popover controller may seem odd, but you need to do this to keep everything in sync.

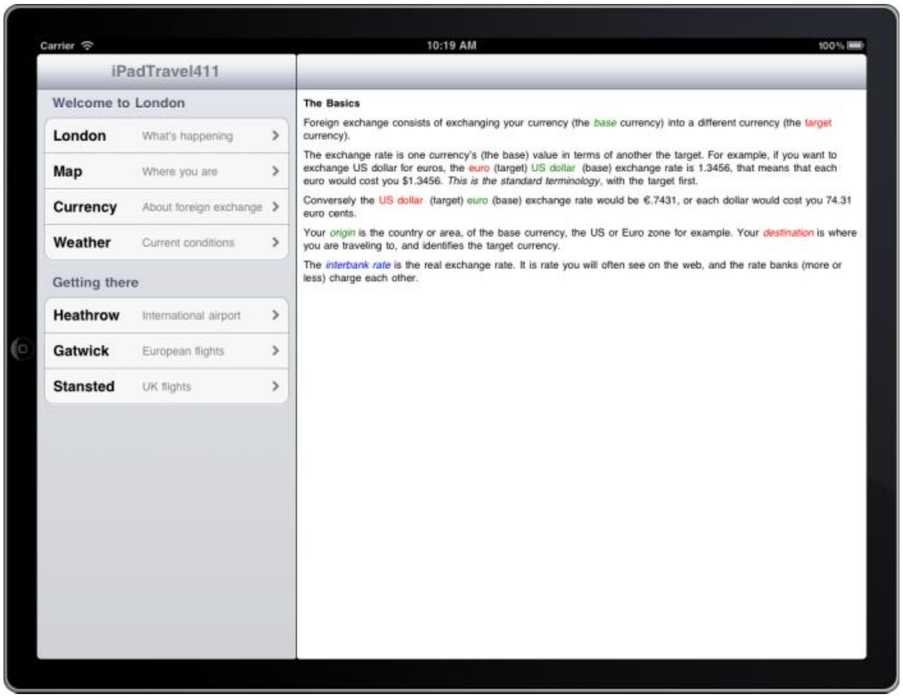
You then save the current controller in the last statement so you can find the popover later to dismiss it.

You also need to add a new instance variable to `RootViewController.h`.

```
DetailViewController  *currentController;
```

The Currency Implementation Model

If the user selects Currency from the Main view in the iPadTravel411 application, he or she will see some very basic information about exchange rates, as illustrated in Figure 16-6. Because this information changes rarely (if ever), I'm going to include this information in the application. The way to do this is to include it as a resource.



The Basics

Foreign exchange consists of exchanging your currency (the *base* currency) into a different currency (the target currency).

The exchange rate is one currency's (the base) value in terms of another the target. For example, if you want to exchange US dollar for euros, the euro (target) US dollar (base) exchange rate is 1.3456, that means that each euro would cost you $1.3456. *This is the standard terminology*, with the target first.

Conversely the US dollar (target) euro (base) exchange rate would be €.7431, or each dollar would cost you 74.31 euro cents.

Your *origin* is the country or area, of the base currency, the US or Euro zone for example. Your *destination* is where you are traveling to, and identifies the target currency.

The *interbank rate* is the real exchange rate. It is rate you will often see on the web, and the rate banks (more or less) charge each other.

Figure 16-6: Reading about how currencies work.

The `CurrencyController` and view follows the same pattern laid down by the `WeatherController` implementation. Do the following to `CurrencyController.m`:

1. Add `#import "Destination.h"` to `CurrencyController.m`.
2. Add the code in Listing 16-4: Initialize WeatherController to `CurrencyController.m`, replacing `WeatherController` with `CurrencyController`.
3. Add the code in Listing 16-7: Adding `computeFramesForOrientation:` to `CurrencyController.m`.
4. Delete the code in Listing 16-8: Deleting `shouldAutorotateToInterfaceOrientation:` in `CurrencyController.m`.

5. Add the Web navigation code in Listings 16-9 to 16-11 to `CurrencyController.m`.

   a. Listing 16-9: When the User Touches a Link

   b. Listing 16-10: `goBack` to Where You Once Belonged

   c. Listing 16-11: You Don't Need the Back Button Any Longer

You might be wondering why I don't just create a superclass and have `WeatherController`, `CurrencyController`, and `CityController` inherit all that code from it. The answer is *you* should, but I didn't because it makes things a little more complex, and things in this chapter are complex enough. I leave this as an exercise for you to do after you've wrapped you head around all this info.

Now you're ready to add the content.

Adding the content

The content for the `Currency` view is in a file I created called `Currencies.html`. To make it available to the application, you need to include it in the application bundle itself,

Now, you can add it to your bundle in one of two ways (you did this earlier when you added the picture file in Chapter 9):

- Open the Project window and drag an `.html` file into the Groups & Files list.

  It's a good idea to create a new group within your project as a snug little home for the file. (I named my new group `Static data`.)

  Or

- Select Project➪Add to Project and then use the dialog that appears to navigate to (and select) the file you want.

This file is available, along with all the other code and files from this book, on my Web site at `www.nealgoldstein.com` on the Downloads page available from the Support page.

The only thing interesting here is that you're going to use some data that you've included with your application as a *resource* (which you can think of as an included file, although it doesn't live in the iPad file system but rather is embedded in the application itself).

Loading the Currency view

To load the `Currency` Web view, the place to start is in the method `viewDidLoad`. This method was included for you in `CurrencyController.m` by the `UIViewController` subclass template (albeit, commented out). Simply uncomment this method and add the code in Listing 16-16 to `CurrencyController.m`. This is similar to the `viewDidLoad` method you added to `WeatherController`.

Listing 16-16: viewDidLoad

```
- (void)viewDidLoad {

  [super viewDidLoad];
  webView.scalesPageToFit = NO;
  [webView loadRequest:[NSURLRequest requestWithURL:
                        [destination currencyBasics]]];

}
```

All this code does is send a message to the `Destination` object (which you passed in when you initialized the `CurrencyController` in the previous section) to find out where the data that the Currency view needs is located. You also notice that, in this case, you set the `scalesPageToFit` property to `NO`. I do that because the program I used to create the view content already took into account the page size.

`loadRequest` is a `UIWebView` method that connects to a given URL and downloads whatever is there. In the case of `WeatherController`, that was a Web site; in this case, it's a resource in your bundle. In the case of the `CityController`, as you'll soon see, it will be an `.html` file on a server.

Of course, you have to give `loadRequest` the file URL. You do this in `Destination.m` in the `currencyBasics` method. Add the code in Listing 16-17 to `Destination.m` and the declaration to `Destination.h`.

Listing 16-17: The currencyBasics Method

```
- (NSURL *)currencyBasics {

  NSString *filePath = [[NSBundle mainBundle]
           pathForResource:@"Currencies" ofType:@"html"];
  NSURL * currencyData= [NSURL fileURLWithPath:filePath];
  return currencyData;
}
```

To get the URL for the resource you just added, you use `pathForResource`, which is an `NSBundle` method. (You used an `NSBundle` method when you got the application name in the `RootViewController` to set the title on the main window back in Chapter 15.) Just give `pathForResource` the name and the file type.

TIP

Be sure you provide the right file type; otherwise, this technique won't work.

Launching the CurrencyController

To launch the `CurrencyController` when the user selects Currency in the Master view, all you have to do is replace the code in `case 2` in `tableView:didSelectRowAtIndexPath:` in `RootViewController.m`.

The idea is to replace this:

```
case 2:
  //do something
  break;
```

with the bolded stuff here:

```
case 2:
  targetController = [[menuList objectAtIndex:menuOffset]
                           objectForKey:kControllerKey];
  if ([targetController isKindOfClass:[NSNull class]]) {

    targetController = [[CurrencyController alloc]   init
           WithDestination:kAppDelegate.destination];
    [[menuList  objectAtIndex:menuOffset]
       setObject:targetController forKey:kControllerKey];
    [targetController release];
  }
  break;
```

If this looks familiar, that's because it is. This is exactly what you did to launch the `MapController`.

Adding the City

If the user selects City from the Main view in the iPadTravel411 application, he or she should see information about what's going on in London (although in the file I provide, there's not much to speak of going on).

How selecting City works is pretty much the same as how selecting Currency works, except this time you're going to use a model object — `City`, which you're already using as an annotation — that contains the location of the URL. You'll see how this all nicely fits together in a moment.

Because `CityController` follows the same pattern laid down by the `Weather` implementations, start off by doing the following to `CityController.m`:

1. Add `#import "Destination.h"` to `CityController.m`.
2. Add the code in Listing 16-7: Adding `computeFramesForOrientation:` to `CityController.m`.
3. Delete the code in Listing 16-7: Deleting shouldAutorotateToInterface Orientation: in `CityController.m`.
4. Add the Web navigation code in listings 16-9 to 16-11 to `CityController.m`.
    a. Listing 16-9: When the User Touches a Link
    b. Listing 16-10: `goBack` to Where You Once Belonged
    c. Listing 16-11: You Don't Need the Back Button Any Longer

Now you're ready to add the content.

Loading the City view

To load the Web view, the place to start is in the method `viewDidLoad`. This method was included for you in `CityController.m` by the `UIViewController` subclass template (albeit, commented out). Simply uncomment this method and add the code in Listing 16-18 to `CityController.m`. This is similar to the `viewDidLoad` method you added to `CurrencyController`.

Listing 16-18: The viewDidLoad Method

```
- (void)viewDidLoad {

  [super viewDidLoad];
  webView.scalesPageToFit = NO;
  [webView loadRequest:[NSURLRequest requestWithURL:
                     [destination cityHappenings]]];
}
```

Just as with the CurrencyController, all this code does is send a message to the Destination object to find out where the data that the City view needs is located. You'll also notice that in this case, you again set the scalesPageToFit property to NO. I do that because the program I used to create the view content already took into account the page size.

As I mention earlier when discussing the Currency view, loadRequest is a UIWebView method that connects to a given URL and downloads whatever is there. It should come as no surprise that you have to give loadRequest the file URL, which is again done in Destination.m — this time in the cityHappenings method.

This bit of coding isn't all that different from the implementation of viewDidLoad in WeatherController and CurrencyController. Add the code in Listing 16-19 to Destination.m, and add its declaration to Destination.h.

Listing 16-19: The cityHappenings Method

```
- (NSURL *) cityHappenings {

  return [city cityHappenings];
}
```

Well now, this is at least a little different. Now you're getting the information from the City model object instead of from Destination.

```
return [city cityHappenings];
```

Back in Chapter 15, I explain why it's a good idea to have model objects that are used by the main model object — Destination. I didn't do this with Currency and Weather because it was so trivial, but in the case of the City view, the City model object also does something besides return the URL — it also has the information necessary for the annotation. So here it makes sense to have it own and return the URL information as well.

Add the code in Listing 16-20 to City.m, and add its declaration to City.h.

Listing 16-20: The City Model Object Returns the NSURL

```
- (NSURL *) cityHappenings {

  return [NSURL URLWithString:
                   @"http://nealgoldstein.com/City.html"];
}
```

This is no different than what you did with the Weather NSURL back in the "Loading the Web View" section, earlier in this chapter — whether it's weather info or city info, all you're doing is grabbing some data from the Web.

Launching the CityController

To launch the CityController when the user selects City in the Master view, all you have to do is replace the code in case 0 in tableView:didSelectRowAtIndexPath: in RootViewController.m.

In other words, what now looks like this:

```
case 0:
  //do something
  break;
```

should be changed so it looks like this:

```
case 0:
  targetController = [[menuList objectAtIndex:menuOffset]
                            objectForKey:kControllerKey];
  if ([targetController isKindOfClass:[NSNull class]]) {

    targetController = [[CityController alloc]
            initWithDestination:kAppDelegate.destination];
    [[menuList  objectAtIndex:menuOffset]
        setObject:targetController forKey:kControllerKey];
    [targetController release];
  }
  break;
```

Third time's the charm — this is exactly what you did to launch the MapController and WeatherController.

A Checkpoint

Build and run your application and you'll now find that all of the selections you can make in the first section of the Master view, both in landscape mode and in portrait mode, work as expected.

Of course, there are a few more things left to do, like printing, implementing preferences, and saving state, and oh yeah, all that stuff in the second section.

On to printing!

Chapter 17

Printing from Your iPad App

In This Chapter

- ▶ How printing works
- ▶ Providing printing options
- ▶ Printing various views
- ▶ Using the Printer Simulator

Over the course of the last few chapters, you've learned quite a lot about creating apps that go beyond the ho-hum. In this chapter, you add to your knowledge by adding a great new iPad feature: printing. Not only is it one of the great additions to iOS 4.2, but it also turns out to be pretty simple to use (a definite break from Chapter 16).

While printing opens up a world of opportunity for apps in general, there are some very definite uses for it in an app like IiPhoneTravel411. The user might want to print out a map or a page on what's happening in London. This is especially helpful for many users without 3G and even those who do have 3G who would prefer not to pay roaming rates. It also moves the iPad one step closer to being able to replace a laptop for many users.

In iPadTravel411, you'll implement printing for most of the views. You'll also find out how to set some parameters so you can print maps in landscape mode, for example, and everything else in portrait mode.

Printing on the iPad

Okay, I'll be the first to admit that, at this point, only a few printers support iPad printing. How few? This few:

- HP Photosmart Premium Fax e-All-in-One Printer series — C410
- HP Photosmart Premium e-All-in-One Printer series — C310
- HP Photosmart Plus e-All-in-One Printer series — B210

TIP

That doesn't sound too promising, but in beta releases of iOS 4.2, printing to a shared printer on a Mac running Mac OS X 10.6.5 was also supported. This support was removed in the Golden Master release. Why it was removed is the subject of a host of unsubstantiated rumors — but don't be surprised if you see printing to a shared printer pop up again at some point in the future.

And, of course, as with anything starting with an i from Apple, you can count on more and more printers supporting iPad printing.

WARNING!

One more caveat: Printing is supported only on those iOS devices that support multitasking.

TIP

For testing purposes, you can save a lot of trees by using the Printer Simulator with your app. I explain how to do that in the section "The Printer Simulator," later in this chapter.

The beauty of iPad printing is that it's pretty simple for the user. All the user needs to do is tap a button — usually found in a navigation bar or toolbar of the view or item the user wants to print. (See Figure 17-1.) The application then presents a Printer Options popover, as shown later on in Figure 17-2. That allows a user to select a printer (and some other options such as page range and copies) and then get something printed by tapping the Print button.

Your application then needs to provide printing output from its content or provide printable data or file URLs. The requested print job is spooled and then control returns to your application. If the destination printer is currently not busy, printing begins immediately. If the printer is already printing or if there are jobs before it in the queue, the print job remains in the print queue until it moves to the top of queue and is printed.

Gee — just like it does on a real computer.

Adding the Print button

The first thing printing-related that a user sees is a Print button. The Print button is often a bar-button item on a navigation bar or a toolbar. The Print button should logically apply to the content the application is presenting; if the user taps the button, the application should print that content. Although the Print button can be any custom button, it's recommended that you use the system item-action button shown in Figure 17-1. The system item-action button is the one on the far-right side of the toolbar.

It looks nice enough, but how do you actually get it there? More specifically, where in your code do you make the changes necessary to get that nice Print button to appear?

Because you want all of your views in the various view controllers derived from `DetailViewController` to be able to print, you'll add the code to the `viewDidLoad` method of the `DetailViewController`. So uncomment out the `viewDidLoad` method to `DetailViewController.m` and add the code in bold in Listing 17-1.

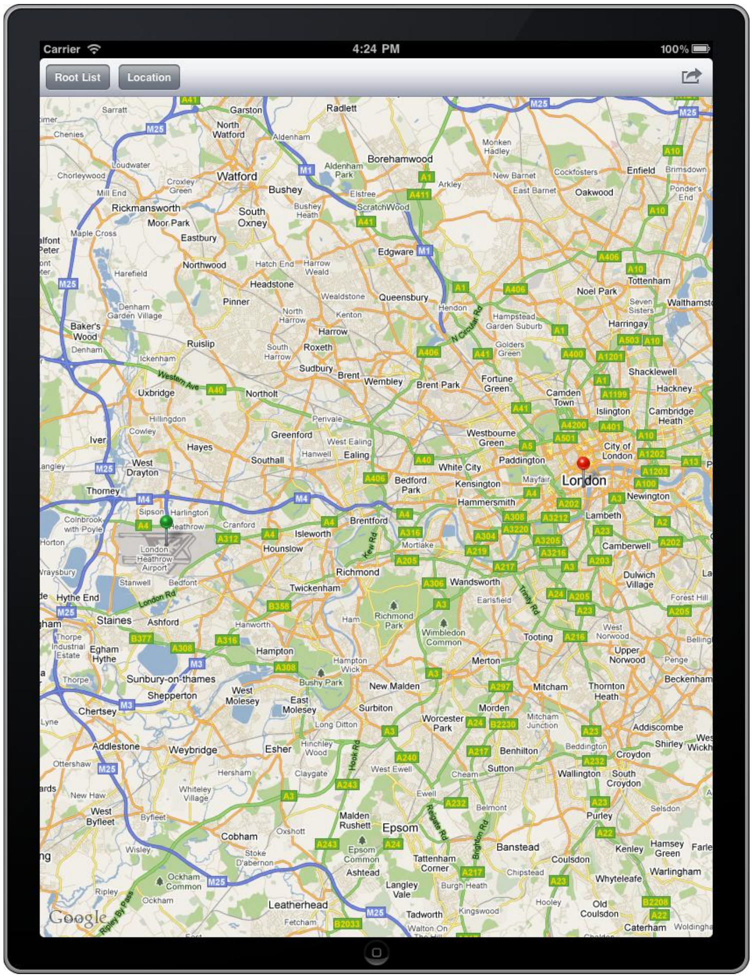


Figure 17-1: The system item-action button used for printing.

Listing 17-1: Adding a Print Button

```
- (void)viewDidLoad {

   [super viewDidLoad];
   UIBarButtonItem *flexibleSpace =
     [[[UIBarButtonItem alloc] initWithBarButtonSystemItem:
          UIBarButtonSystemItemFlexibleSpace
                    target:nil action:nil] autorelease];

   printButton= [[[UIBarButtonItem alloc]
    initWithBarButtonSystemItem:UIBarButtonSystemItemAction
         target:self action:@selector(print:)]autorelease];
   NSMutableArray *items = [[toolbar items] mutableCopy];
   [items addObject:flexibleSpace];
   [items addObject:printButton];
   [self.toolbar setItems:items animated:YES];
   [items release];
   [super viewDidLoad];
}
```

The first thing you do is allocate a flexible space “button,” `UIBarButton SystemItemFlexibleSpace`. This simply adds a blank space which forces the Print button to the right (whether or not there are other buttons in the toolbar).

You then create a standard Print button:

```
printButton= [[[UIBarButtonItem alloc] initWithBar
          ButtonSystemItem:UIBarButtonSystemItem
          Action target:self action:@selector(print:)]
          autorelease];
```

and add it to the toolbar:

```
NSMutableArray *items = [[toolbar items] mutableCopy];
[items addObject:flexibleSpace];
[items addObject:printButton];
[self.toolbar setItems:items animated:YES];
[items release];
[super viewDidLoad];
```

To round things off, you have to add the instance variable to `DetailViewController.h` as well:

```
UIBarButtonItem *printButton;
```

One side effect of the way I have you do this is that now the Location button for the Map view and the Back button for the Web views (when it is needed) appear after the Print button. You can change that if you’d like, but I kind of like it this way.

One thing I also want to direct your attention to is the fact that — because of all of the work you did in the last chapter — adding this button works seamlessly in all views and in all orientations. Good job!

The print methods

When a user taps the Print button, your controller (a `DetailViewController` subclass) receives the action (`print:`) message. So the next thing you need to do is add the method `print:`.

This code is identical for all the controllers that have a Web view. (I know, another argument for the kind of superclass I mention in Chapter 16, in the section on the currency implementation model.) Add the code in Listing 17-2 to `WeatherController.m`, `CityController.m`, and `CurrencyController.m`.

Listing 17-2: The print Method for Web Views

```
- (void)print:(id)sender {

  [self print:webView orientation:
                        UIPrintInfoOrientationPortrait];
}
```

All this method does is turn around and send itself the `print;orientation:` message, which you implement next in the `DetailViewController`. The reason the method exits at all is to send two arguments to the superclass method. The first is the view you want printed, and the second the print orientation. For all the Web views, both the view and orientation are the same — you want the orientation to be portrait — but for the Map view, you want to pass a Map view to be printed *and* print it in landscape. Add the code in Listing 17-3 to `MapController.m` to accomplish that.

Listing 17-3: The print Method for the Map View

```
- (void)print:(id)sender {

  [self print:mapView orientation:
                       UIPrintInfoOrientationLandscape];
}
```

I could have accomplished customizing the print orientation on a class-by-class basis in a lot of ways — including subclassing or by using instance variables in the `DetailViewController` class that was initialized by each derived class. I chose to do it this way to give you an idea of how you could have a bit more control on a class-by-class basis.

The UIPrintInteractionController

The `print:orientation:` method in the `DetailViewController` is where the action is.

To let the `print: orientation:` method do its thing, add the code in Listing 17-4 to `DetailViewController.m` and add the method declaration to `DetailViewController.h`.

Listing: 17-4: The Main Printing Code

```
- (void) print:(UIView *) theView
        orientation:(UIPrintInfoOrientation) orientation {

  UIPrintInteractionController *controller =
     [UIPrintInteractionController sharedPrintController];
  if(!controller){
    NSLog(@"Couldn't open Print Interaction Controller!");
    return;
  }
  UIPrintInfo *printInfo = [UIPrintInfo printInfo];
  printInfo.jobName = @"iPadTravel411";
  printInfo.outputType = UIPrintInfoOutputGeneral;
  printInfo.orientation = orientation;
  if (orientation == UIPrintInfoOrientationPortrait)
    printInfo.duplex = UIPrintInfoDuplexLongEdge;
  else
    printInfo.duplex = UIPrintInfoDuplexShortEdge;
  controller.printInfo = printInfo;
  controller.showsPageRange = YES;
  controller.printFormatter =
                           [theView viewPrintFormatter];
  void (^completionHandler)
       (UIPrintInteractionController *, BOOL, NSError *) =
       ^(UIPrintInteractionController *printController,
                         BOOL completed, NSError *error) {
    if(!completed && error) {
      NSLog(@"FAILED! Error code: %u in domain: %@",
            error.code, error.domain);
    }
  };

  if (UI_USER_INTERFACE_IDIOM() ==
                                UIUserInterfaceIdiomPad)
  [controller presentFromBarButtonItem:printButton
        animated:YES completionHandler:completionHandler];
  else
    [controller presentAnimated:YES
                     completionHandler:completionHandler];
}
```

So this is how it works:

When a user taps the Print button, you get the shared instance of `UIPrintInteractionController`, which handles the heavy lifting for you. It is responsible for the Print Options popover, for example, which allows the user to select a printer, specify the number of copies (and possibly a range of pages), and choose single-sided or double-sided printing (if the printer supports duplex printing). When users make their selections and tap Print, the print job commences.

```
UIPrintInteractionController *controller =
     [UIPrintInteractionController sharedPrintController];
  if(!controller){
    NSLog
         (@"Couldn't open Print Interaction Controller!");
    return;
  }
```

If for some reason you can't open a Print Interaction Controller, you log the fact to the debugger console.

The next thing you do is create a `UIPrintInfo` object so you can set some print job parameters.

```
UIPrintInfo *printInfo = [UIPrintInfo printInfo];
```

A `UIPrintInfo` object contains information about a print job, including which printer to use, output type (normal, photo, grayscale), orientation (portrait or landscape), and any selected duplex mode.

First, you assign the job name:

```
printInfo.jobName = @"iPadTravel411";
```

Then you assign `UIPrintInfoOutputGeneral` to the `outputType`. `UIPrintInfoOutputGeneral` is a mix of text, graphics, and images in color. (Other choices are `UIPrintInfoOutputPhoto` and `UIPrint InfoOutputGrayscale`.)

```
printInfo.outputType = UIPrintInfoOutputGeneral;
```

You assign the orientation to whatever was passed in as the `orientation` argument.

```
printInfo.orientation = orientation;
```

You then set duplex printing (which is two-sided printing) so that it's available if the printer supports it. If you're doing portrait printing, you want to duplex along the long edge; if not, you want to use the short edge.

```
if (orientation == UIPrintInfoOrientationPortrait)
    printInfo.duplex = UIPrintInfoDuplexLongEdge;
  else
    printInfo.duplex = UIPrintInfoDuplexShortEdge;
```

You then assign the `printInfo` object to the `printInfo` property of the `UIPrintInteractionController` instance.

```
controller.printInfo = printInfo;
```

If you don't assign a `printInfo` object, `UIKit` assumes default attributes for the print job. (For example, the job name is the application name.)

You then tell the controller you want it to show the page range controls.

```
controller.showsPageRange = YES;
```

After that, you need to tell the controller what to print. You do that by assigning an object to one of the following `UIPrintInteractionController` properties:

- `printingItem`: You assign a single *print-ready object* — an `NSData`, `NSURL`, `UIImage`, or `ALAsset` object containing or referencing PDF data or image data.
- `printingItems`: You assign an array of print-ready objects. (See `printingItem`.)
- `printFormatter`: You assign a *print formatter* — an object that knows how to lay out content of a certain type.
- `printPageRenderer`: You assign a *page renderer* — a custom object that draws the content for printing.

You should only set one of these properties for any print job.

In Listing 17-4, you'll use a print formatter. Although you could also create a custom print formatter for complex content that allows you to specify the margins of printed content and the starting page for printing, for now you'll just use a pretty standard `UIViewPrintFormatter` that's a concrete subclass of `UIPrintFormatter`.

Certain classes in the SDK know how to draw their contents for printing, including `UIWebView`, `UITextView`, and `MKMapView`. These come supplied with a `UIViewPrintFormatter` instance that you're free to use, thank you very much. You get that instance by sending the viewPrintFormatter message to the view and then use that print formatter to print the view.

```
controller.printFormatter = [theView viewPrintFormatter];
```

The next thing you do is create a completion handler. The completion handler is a block of type UIPrintInteractionCompletionHandler that's invoked when a print job either completes successfully or is terminated because of an error. This completion handler can clean up the state (you don't do that here) and/or log error messages (you do that here).

```
void (^completionHandler)
       (UIPrintInteractionController *, BOOL, NSError *) =
  ^(UIPrintInteractionController *printController,
                         BOOL completed, NSError *error) {
    if(!completed && error){
      NSLog(@"FAILED! Error code: %u in domain: %@",
                               error.code, error.domain);
    }
  };
```

You then present the user interface.

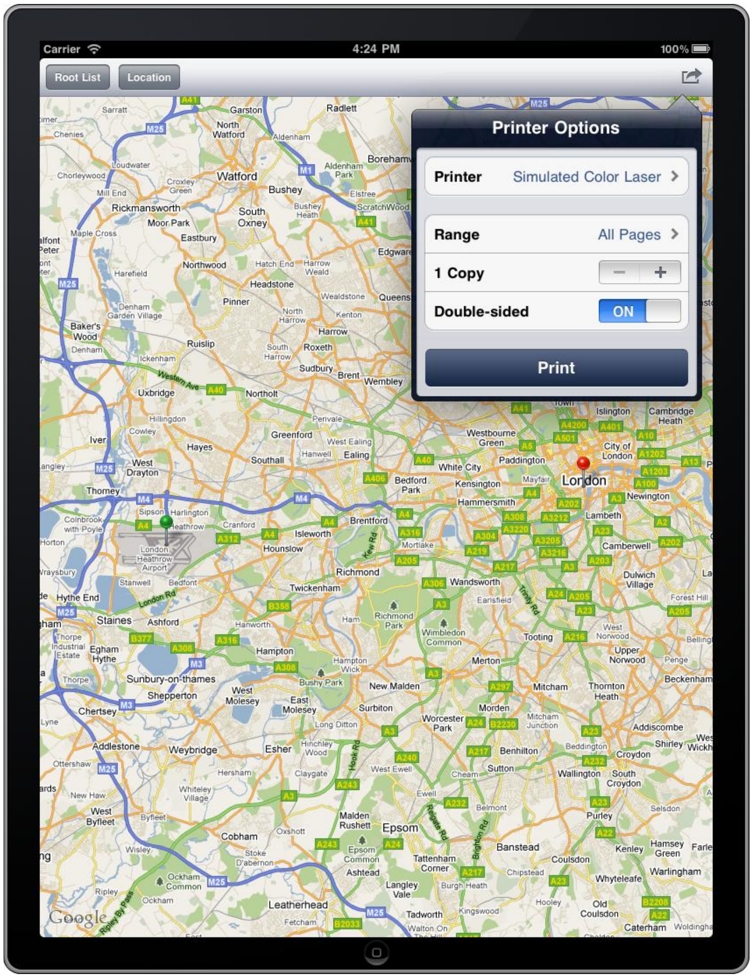
Because this is an iPad book, you're going to present an iPad user interface, but I wanted to show you how you would handle it if your app could also run on an iPhone or iPod touch.

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
    [controller presentFromBarButtonItem:printButton
           animated:YES completionHandler:completion
           Handler];
  else
    [controller presentAnimated:YES completionHandler:
           completionHandler];
```

UI_USER_INTERFACE_IDIOM() is a UIKit function that returns either UIUserInterfaceIdiomPhone (if the device is an iPhone or iPod touch) or UIUserInterfaceIdiomPad (if the device is an iPad). On an iPad, the UIKit framework displays a Popover view containing the Printer options, as shown in Figure 17-2. An application can animate this view to appear from the Print button (you do that here with the help of presentFromBarButtonItem: printButton) or from an arbitrary area of the application's user interface.

Just for your information, on iPhone and iPod touch devices, UIKit displays a sheet of printing options that an application can animate to slide up from the bottom of the screen.

After a print job has been submitted and is either printing or waiting in the print queue, users can check on its status by double-tapping the Home button to access the Print Center in the multitasking UI. Users can tap a print job in the Print Center to get detailed information about the job (see Figure 17-3) and cancel jobs that are printing or waiting in the queue. The Print Center is a background system application that shows the order of jobs in the print queue, including those that are currently printing. It's available only while a print job is in progress.



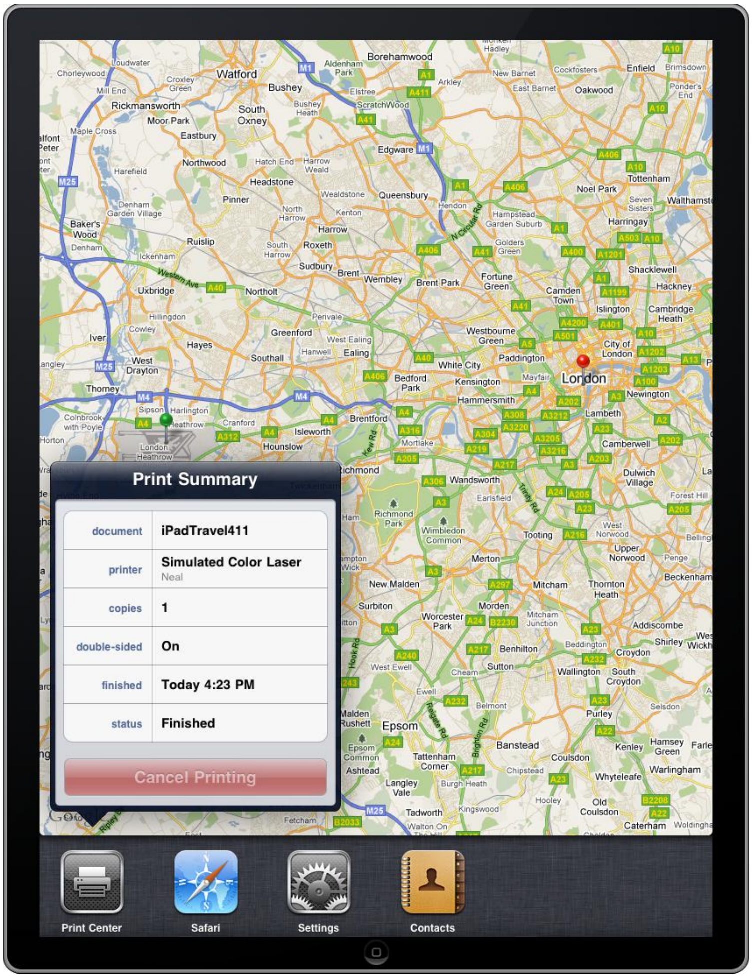
Printer Simulated Color Laser

Range All Pages

1 Copy

Double-sided ON

Figure 17-2: Printer Options popover view (iPad).


Print Summary

document	iPadTravel411
printer	Simulated Color Laser Neal
copies	1
double-sided	On
finished	Today 4:23 PM
status	Finished

Figure 17-3: The Print Center.

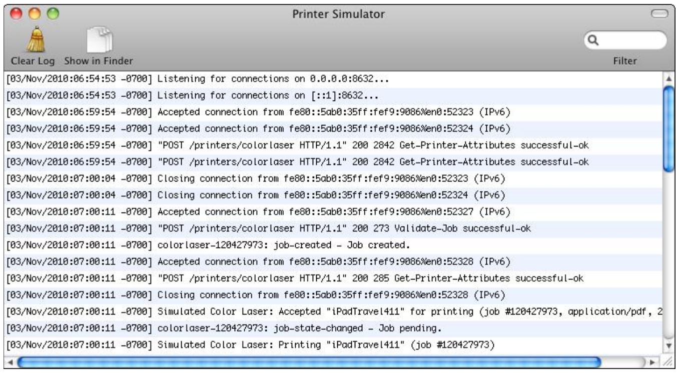
The Printer Simulator

The SDK for iOS 4.2 (and later) provides a Printer Simulator application that you can use to test your application's printing capabilities. The application simulates various printer types — inkjet, black-and-white laser, color laser, and so on — and uses the Preview application to display the output. The Printer Simulator also logs information from the printing system about each print job.

It's probably a good idea to use the Print Simulator to test printing, unless you really need a lot of Heathrow London maps, London weather reports, or information about currency to hand out to your friends and family.

If you want to have the Printer Simulator show up as an option in the Printing Options window, you need to start it first.

The way to start the Printer Simulator is by choosing File➪Open Printer Simulator in the Simulator itself. After you do that, the Printer Simulator window opens. (You can see some of the output it logs in the window in Figure 17-4.) I won't be going into what all of that means in this book.



```
[03/Nov/2010:06:54:53 -0700] Listening for connections on 0.0.0.0:8632...
[03/Nov/2010:06:54:53 -0700] Listening for connections on [::1]:8632...
[03/Nov/2010:06:59:54 -0700] Accepted connection from fe80::5ab0:35ff:fef9:9086%en0:52323 (IPv6)
[03/Nov/2010:06:59:54 -0700] Accepted connection from fe80::5ab0:35ff:fef9:9086%en0:52324 (IPv6)
[03/Nov/2010:06:59:54 -0700] "POST /printers/colorlaser HTTP/1.1" 200 2842 Get-Printer-Attributes successful-ok
[03/Nov/2010:06:59:54 -0700] "POST /printers/colorlaser HTTP/1.1" 200 2842 Get-Printer-Attributes successful-ok
[03/Nov/2010:07:00:04 -0700] Closing connection from fe80::5ab0:35ff:fef9:9086%en0:52323 (IPv6)
[03/Nov/2010:07:00:04 -0700] Closing connection from fe80::5ab0:35ff:fef9:9086%en0:52324 (IPv6)
[03/Nov/2010:07:00:11 -0700] Accepted connection from fe80::5ab0:35ff:fef9:9086%en0:52327 (IPv6)
[03/Nov/2010:07:00:11 -0700] "POST /printers/colorlaser HTTP/1.1" 200 273 Validate-Job successful-ok
[03/Nov/2010:07:00:11 -0700] colorlaser-120427973: job-created - Job created.
[03/Nov/2010:07:00:11 -0700] Accepted connection from fe80::5ab0:35ff:fef9:9086%en0:52328 (IPv6)
[03/Nov/2010:07:00:11 -0700] "POST /printers/colorlaser HTTP/1.1" 200 285 Get-Printer-Attributes successful-ok
[03/Nov/2010:07:00:11 -0700] Closing connection from fe80::5ab0:35ff:fef9:9086%en0:52328 (IPv6)
[03/Nov/2010:07:00:11 -0700] Simulated Color Laser: Accepted "iPadTravel411" for printing (job #120427973, application/pdf, 2
[03/Nov/2010:07:00:11 -0700] colorlaser-120427973: job-state-changed - Job pending.
[03/Nov/2010:07:00:11 -0700] Simulated Color Laser: Printing "iPadTravel411" (job #120427973)
```

Figure 17-4: The Printer Simulator window.

There's Much More to Printing

As you might expect, there's a lot more to printing. As I say earlier in this chapter, you can always create a custom print formatter for complex content that allows you to specify the margins of printed content and the starting page for printing.

The next step beyond that is to create a `UIPageRenderer` instead of a `UIViewPrintFormatter`. This gives you greater control over the content drawn for printing. It includes properties for the page count and for heights of headers and footers of pages. It also has several methods that you can override to draw specific portions of a page (such as the header, the footer, or the content itself) or even to integrate page renderer and print formatter drawing.

In a different direction, you can also create single print-ready `NSData`, `NSURL`, `UIImage`, or `ALAsset` objects that reference PDF data or image data.

Obviously that's an exercise for you to do on your own.

Chapter 18

Providing Content in the Master View

In This Chapter

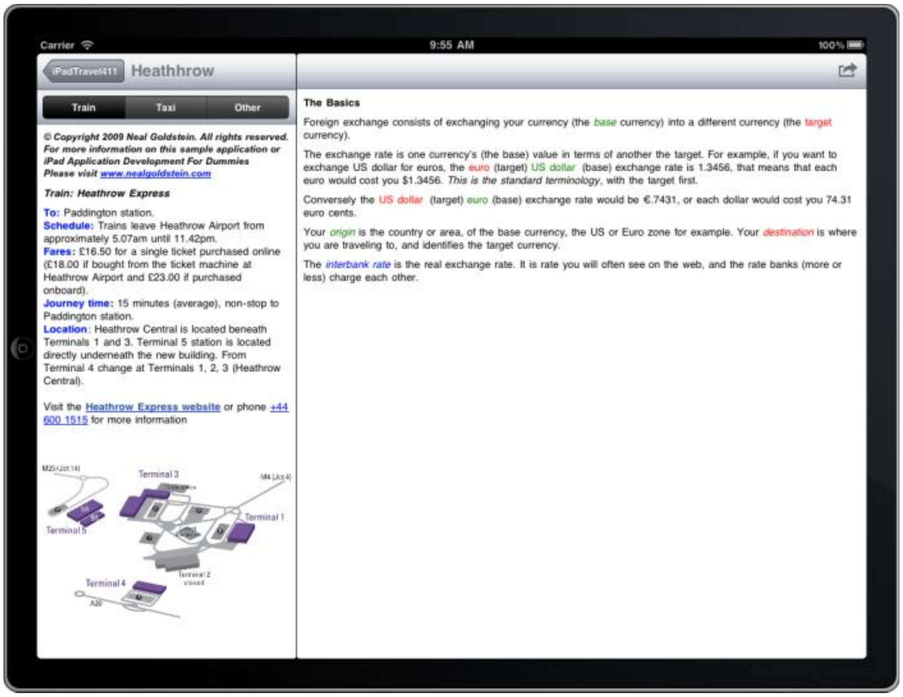
- Adding Content views to the Master view
- Taking another look at model objects
- Creating views without using Interface Builder
- Accessing data on the Internet
- Saving files

So far you've seen how to use the Master view controller to change something in the Detail view. But you can also push new controllers in the Master view to display more of a hierarchy or even more data. (More on that in the "Navigating the Navigation Controller" section, later in this chapter.) Figure 18-1 shows you how you can be getting information about transportation into London while looking at currency information about what it takes to turn dollars into pounds so you can pay for your Heathrow Express ticket. (The more data, the merrier, I say.)

In this chapter, you're going to create an `AirportController` that manages the airports and their views in the Master view.

The Airport Controller

At this point, adding a new controller should be second nature to you. I know it may be getting a bit tedious, but this is what you have to go through for almost all the views you'll have in your application.



© Copyright 2009 Neal Goldstein. All rights reserved. For more information on this sample application or iPad Application Development For Dummies Please visit www.nealgoldstein.com

Train: Heathrow Express

To: Paddington station.
Schedule: Trains leave Heathrow Airport from approximately 5.07am until 11.42pm.
Fares: £16.50 for a single ticket purchased online (£18.00 if bought from the ticket machine at Heathrow Airport and £23.00 if purchased onboard).
Journey time: 15 minutes (average), non-stop to Paddington station.
Location: Heathrow Central is located beneath Terminals 1 and 3. Terminal 5 station is located directly underneath the new building. From Terminal 4 change at Terminals 1, 2, 3 (Heathrow Central).

Visit the Heathrow Express website or phone +44 600 1515 for more information

The Basics

Foreign exchange consists of exchanging your currency (the *base* currency) into a different currency (the target currency).

The exchange rate is one currency's (the base) value in terms of another the target. For example, if you want to exchange US dollar for euros, the euro (target) US dollar (base) exchange rate is 1.3456, that means that each euro would cost you $1.3456. *This is the standard terminology*, with the target first.

Conversely the US dollar (target) euro (base) exchange rate would be €.7431, or each dollar would cost you 74.31 euro cents.

Your *origin* is the country or area, of the base currency, the US or Euro zone for example. Your *destination* is where you are traveling to, and identifies the target currency.

The *interbank rate* is the real exchange rate. It is rate you will often see on the web, and the rate banks (more or less) charge each other.

Figure 18-1: What is all this exchange rate stuff about anyway?

Adding the Airport controller and nib file

Here's the drill:

1. **Select the View controllers group and then choose File➪New File from the main menu (or press ⌘+N) to call up the New File dialog.**
2. **In the leftmost pane of the dialog, first select Cocoa Touch Classes under the iOS heading, select the UIViewController subclass template in the topmost pane, and then make sure that the following are all selected:**
    - With XIB for User Interface
    - Targeted for iPad
3. **Click Next.**

   You see a new dialog asking for some more information.
4. **Enter** AirportController **in the File Name field and then click Finish.**

The idea behind `AirportController` is that you want to display the transportation options open to the user. The first step down that road involves adding methods and instance variables to that controller.

Start by looking at the changes you need to make in `AirportController.h` to get an overview of sorts of where you're headed. Make the changes in bold you see in Listing 18-1.

Listing 18-1: AirportController.h

```
#import <UIKit/UIKit.h>
#import "DetailViewController.h"
@class Destination;

@interface AirportController : UIViewController
           DetailViewController <UIWebViewDelegate> {

  UIWebView           *webView;
  Destination         *destination;
  UIToolbar           *theToolbar;
  UISegmentedControl *segmentedControl;
  UIBarButtonItem     *backButton;
}
- (id)initWithDestination:(Destination *)aDestination
                              airportID:(int) theAirport;

@end
```

If you look at the Master view in Figure 18-1, you can see that you're going to be adding a toolbar with a segmented control — the Train-Taxi-Other business — as well as an image and a "back button," just as you have done in other controllers (`WeatherController` for example) that let you cruise the Web. In Listing 18-1, you create some instance variables (I explain why they're not `IBOutlet`s next) to take care of this.

Be sure to save the file before you continue.

Right off the bat, you're going to have to add the initialization method in Listing 18-2 to `AirportController.m`.

Listing 18-2: initWithDestination:airportID:

```
- (id)initWithDestination:(Destination *)aDestination
                             airportID:(int) theAirport {

  if (self = [super initWithNibName:@"AirportController"
                                            bundle:nil]) {
    destination = aDestination;
    self.title = [destination
                          returnAirportName:theAirport];
  }
  return self;
}
```

London is serviced by three separate airports (Heathrow, Gatwick, and Stansted), and I'm planning to provide information for all three. Because the kind of information I want to provide is basically the same, it would be nice to have to create only one view controller class that could then provide the information for any airport — that's the reason for the `airportID` and the `returnAirportName` message it sends to `destination`. Although I won't be showing you how to implement that in this book, it does provide the beginnings of the framework you need to do it on your own.

You also have to add a new method to `Destination.m` and its declaration to `Destination.h`, as shown in Listing 18-3. `returnAirportName:` is a step down the path I suggested earlier. Here you're having a "generic" airport object that doesn't know anything about its airport — the gatekeeper here being the `Destination` object. I'm fudging a bit because I'm going to implement only one airport, but you get the picture.

Listing 18-3: Returning the Name for an Airport

```
- (NSString *)returnAirportName:(int) theAirportID {

  return airport.title;
}
```

You'll notice here that I'm taking advantage of the fact that an `Airport`, which you have been using so far simply as an annotation, already has a name (`title`) associated with it. In the section "The Destination Model," later in this chapter, I show you how to nudge `Airport` into being even more of a model object of the kind I explain in Chapter 13.

With your preliminaries out of the way, you can start adding content by first taking care of the image. To do that, you add the image as a resource in your program bundle. The one I'm showing is a map of Heathrow Airport, and you can download that from my Web site or use any other image you want.

To make it available to the application, I need to include the image in the application bundle itself, although I could have downloaded it the first time the application ran. (But there's method in my madness. Including it in the bundle *does* give me the opportunity to show you how to handle this kind of data.)

You can add it to your bundle one of two ways:

- Open the Project window and drag the `Heathrow.png` file into the Groups & Files list, like you did with the icon in Chapter 9 or even the `Currencies.html` file in Chapter 16.

It's a good idea to put this is the same place as `Currencies.html`.

Or

- Choose Project➪Add to Project and then use the dialog that appears to find and select the file you want.

  You also want to select Copy when the dialog appears.

The only thing interesting here is that you're going to use some data (the Airport image) that you have included with your application as a resource (just as you did with the `currencies.html` file).

Up to now, you've been using Interface Builder to build your View controller views. Now, instead of using Interface Builder, I show you how to programmatically create the elements you need. It's a good idea for you to be able to do this programmatically to round out your understanding.

This is why, as you can see in Listing 18-1, instance variables `toolbar` and `webView` are not declared as `IBOutlets`. Since Interface Builder plays no role here, there is no reason to do so.

Setting up the view

Your `AirportController` is going to be getting the content for its views from the `Destination` object — content which it then passes on to the view itself. (No surprise here; you've been doing it this way all along.) You use the `viewDidLoad` method to get your view prepped for its big day. This particular method was included for you in `AirportController.m` by the `UIViewController` subclass template (albeit commented out). Just to make things interesting, this time delete the current incarnation of this method and replace it with the code in Listing 18-4.

Listing 18-4: Setting Things Up in viewDidLoad

```
- (void)viewDidLoad {

  [super viewDidLoad];

  CGRect webViewFrame = CGRectMake(0, kToolbarHeight,
                       kPopoverWidth, kAirportViewHeight);
  webView = [[UIWebView alloc]
                              initWithFrame:webViewFrame];
  [self.view addSubview:webView];
  webView.delegate = self;
```

(continued)

Listing 18-4 ***(continued)***

```
   theToolbar = [UIToolbar new];
   CGRect viewBounds = self.view.frame;
   viewBounds.size.width = kPopoverWidth;
   viewBounds.size.height = kToolbarHeight;
   [theToolbar setFrame:viewBounds];
   [self.view addSubview:theToolbar];

   segmentedControl = [[UISegmentedControl alloc]
           initWithItems:[NSArray arrayWithObjects:
                        @"Train", @"Taxi", @"Other", nil]];
   [segmentedControl addTarget:self action:@
           selector(selectTransportation:) forControl
           Events:UIControlEventValueChanged];
   segmentedControl.segmentedControlStyle =
                             UISegmentedControlStyleBar;
   segmentedControl.tintColor = [UIColor darkGrayColor];
   CGRect segmentedControlFrame = ((UIViewController *)
           kAppDelegate.rootViewController).view.frame;
   segmentedControlFrame.size.width =
                theToolbar.frame.size.width - kLeftMargin;
   segmentedControlFrame.size.height = kSegControlHeight;
   segmentedControl.frame = segmentedControlFrame;
   segmentedControl.selectedSegmentIndex = 0;

   UIBarButtonItem *choiceItem = [[UIBarButtonItem alloc]
                      initWithCustomView:segmentedControl];
   theToolbar.items = [NSArray arrayWithObject:choiceItem];
   [segmentedControl release];
   [choiceItem release];

   CGRect imageViewFrame = CGRectMake(kImageIndent, kTool
           barHeight+kAirportViewHeight,
            kPopoverWidth - (2*kImageIndent), kImageSize);
   UIImageView *imageView = [[UIImageView alloc]
                          initWithFrame:imageViewFrame];
   UIImage *webImage =
                      [UIImage imageNamed:@"Heathrow.png"];
   imageView.image = webImage;
   [self.view addSubview:imageView];

   self.contentSizeForViewInPopover =
                CGSizeMake(kPopoverWidth, kPopoverHeight);

 }
```

Here's what Listing 18-4 has in mind:

1. **Create a frame for the Web view.**

   You specify its origin in its superview as (`0`, `kToolbarHeight`), which puts it at the left edge and just below the toolbar. You specify its width and height as the width of a popover and a new constant you add to `Constants.h` — `kAirportViewHeight`. You then allocate the Web view, initialize it with the frame you created, and then add it to the view that comes with the controller. This is the general approach you take when you create your view elements programmatically and add them to a view.

```
CGRect webViewFrame = CGRectMake(0, kToolbarHeight,
                   kPopoverWidth, kAirportViewHeight);
webView = [[UIWebView alloc]
                          initWithFrame:webViewFrame];
[self.view addSubview:webView];
```

You also need to add the following constant to `Constants.h`.

```
#define kAirportViewHeight 420
```

2. **Make the `AirportController` the Web view's delegate.**

```
webView.delegate = self;
```

3. **Create the toolbar.**

This shows another approach to creating view elements. Here I create the toolbar but don't initialize it. (In fact, there is no initialization method.) I then create a frame based on its superview, adjust the height and width accordingly, assign it to the toolbar, and then add the toolbar to the view.

```
theToolbar = [UIToolbar new];
CGRect viewBounds = self.view.frame;
viewBounds.size.width = kPopoverWidth;
viewBounds.size.height = kToolbarHeight;
[theToolbar setFrame:viewBounds];
[self.view addSubview:theToolbar];
```

You also need to add the following constant to `Constants.h`.

```
#define kToolbarHeight 44
```

4. **Create the segmented control that will go inside the toolbar.**

```
segmentedControl = [[UISegmentedControl alloc]
        initWithItems: [NSArray arrayWithObjects:
                   @"Train", @"Taxi", @"Other", nil]];
segmentedControl addTarget:self action:@selector
        (selectTransportation:)
         forControlEvents:UIControlEventValueChanged];
segmentedControl.segmentedControlStyle =
                          UISegmentedControlStyleBar;
segmentedControl.tintColor = [UIColor darkGrayColor];
CGRect segmentedControlFrame = ((UIViewController *)
          kAppDelegate.rootViewController).view.frame;
segmentedControlFrame.size.width =
           theToolbar.frame.size.width - kLeftMargin;
segmentedControlFrame.size.height =
                                    kSegControlHeight;
segmentedControl.frame = segmentedControlFrame;
segmentedControl.selectedSegmentIndex = 0;
```

In the first line of code, you're creating a segmented control and an array that specifies the text for each segment. You then set the Target-Action parameters by saying that if a segment is tapped by the user (`UIControlEventValueChanged`), then the `selectTransportation:` message is sent to `self` — in this case, `self` is the `AirportController`. You then compute the size of the segmented control as you would for any other subview. The last line specifies the initial segment (`0`) selected when the view is created; before the view is displayed, the `selectTransportation:` message is sent to display the content associated with segment `0`. (You can see the code for `selectTransportation:` in all its glory in Listing 18-5.)

You also need to add the following constants to `Constants.h`.

```
#define kLeftMargin        16
#define kSegControlHeight  30
```

5. **Add the segmented control to the toolbar.**

```
UIBarButtonItem *choiceItem = [[UIBarButtonItem alloc]
                 initWithCustomView:segmentedControl];
theToolbar.items =
                 [NSArray arrayWithObject:choiceItem];
[segmentedControl release];
[choiceItem release];
```

You get the `choiceBar` (`UIToolbar`) to display controls by creating an array of instances of `UIBarButtonItem` and assigning the array to the `items` property of the `UIToolbar` object (your `choiceBar`). In this case, you create a `UIBarButtonItem` and initialize it with the segmented control you just created. You then create the array and assign it to `items`. This is similar to the way you have been handling buttons in the toolbar in the Detail view.

You then can release `choiceItem` because the `NSArray` has a reference to it.

6. **Add the Image view with its image.**

This should be old hat by now. You do a little adjusting to get the image nicely indented. The only new (and interesting) piece is the `imageNamed:` method. It looks in the system caches for an image object with the name and returns that object if it exists. If it isn't already in the cache, this method loads the image data from the bundle, caches it, and then returns the object. (This is the Heathrow map image you just added.)

```
CGRect imageViewFrame = CGRectMake(kImageIndent,
        kToolbarHeight+kAirportViewHeight,
        kPopoverWidth - (2*kImageIndent), kImageSize);
UIImageView *imageView = [[UIImageView alloc]
                       initWithFrame:imageViewFrame];
UIImage *webImage =
                [UIImage imageNamed:@"Heathrow.png"];
imageView.image = webImage;
[self.view addSubview:imageView];
```

You also need to add to `Constants.h`:

```
#define kImageSize   190
 #define kImageIndent 6
```

There's another Image view method that you could use which creates an Image view initialized with the specified image, and that's the `initWithImage:` method. But this method adjusts the frame of the receiver to match the size of the specified image — not what I'm after here.

7. **Finally, you'll have to set the size of the popover.**

```
self.contentSizeForViewInPopover =
           CGSizeMake(kPopoverWidth, kPopoverHeight);
```

As I mention in Chapter 14, you can set the size of the popover. And, later in that chapter, I have you change the popover size in the `viewDidLoad` method in `RootViewController.m`.

Because popovers normally get their size based on the size of the view in the view controller, you're going to have to specify the size in the `AirportController` just as you did in the case of the `RootViewController`.

That's the run-through through Listing 18-4, but don't forget that you have to add the following `#import` statements to `AirportController.m`.

```
#import "Destination.h"
#import "Constants.h"
#import "iPadTravel411AppDelegate.h"
```

At this point, you have the view set up and waiting for data, as well as the segmented control across the top that will allow the user to select `@"Train"`, `@"Taxi"`, `@"Other"`.

Responding to the user selection in the choice bar

You've set things up so that when the view is first created — or when the user taps a control — the `selectTransportation:` method in `AirportController` is called, allowing the `AirportController` to hook up what the view needs in order to display what the model has to offer. Listing 18-5 shows the necessary code for the `selectTransportation:` method in all its elegance. Add this to `AirportController.m` and its declaration to `AirportController.h`.

Listing 18-5: selectTransportation:

```
- (void)selectTransportation:(id) sender {

  [webView loadRequest:[NSURLRequest requestWithURL:
  [destination returnTransportation:
     (((UISegmentedControl *)
                           sender).selectedSegmentIndex)]]];
}
```

This code is executed when the user selects one of the segmented controls (`Train`, `Taxi`, `Other`) that you added to the view. `(((UISegmentedControl *) sender).selectedSegmentIndex)` gives you the segment number. If you'll notice, the controller has no idea — nor should it care — what was selected. It just passes what was selected on to the `Destination` object.

All this does is send a message to the `Destination` object to find out where the data the Web view needs is located, `[destination returnTransportation: (((UISegmentedControl *) sender).selectedSegmentIndex)]`, and then send a message to the Web view to load it. This is more or less what you did in Chapter 16 with `Weather`, `Currency`, and `City`, but I explain more about the mechanics of this shortly.

Finally, as you did with all the other classes you derived for `DetailViewController`, delete the method `shouldAutorotateToInterfaceOrientation:` in `AirportController.m`.

The Destination Model

You're starting to get all your pieces lined up. Now it's time to take a look at what happens when the controller sends messages to the model.

Take a look at how `returnTransportation:` works. Start by adding the code in Listing 18-6 to `Destination.m` and the declaration to `Destination.h`.

Listing 18-6: Returning the Transportation Link

```
- (NSURL *) returnTransportation:(int) aType {

  return [airport returnTransportation:aType];
}
```

As you can see, `Destination` simply turns around and gets the `NSURL` from the `Airport` object. This is the implementation of composition architecture I speak of in Chapter 13.

Now that you've seen all the things the `Airport` object is responsible for, you're going to have to build it.

Building the Airport

To continue work on the `Airport` object, take a look at the methods you'll need to implement. Add the code in bold in Listing 18-7 to `Airport.h`.

Listing 18-7: Airport.h

```
@interface Airport : Annotation {

}
- (NSURL *) returnTransportation:(int) transportationType;

@end
```

In Listing 18-8, you see the `returnTransportation:` method. Add this code to `Airport.m`.

Listing 18-8: Airport Model Method Used by Destination

```
- (NSURL *) returnTransportation:
                               (int) transportationType  {

    NSURL *url = [[NSURL alloc] autorelease];
    switch (transportationType) {
      case 0: {
        url = [NSURL URLWithString:
          @"http://nealgoldstein.com/ToFromiPad100.html"];
        [self saveAirportData:
```

(continued)

Listing 18-8 ***(continued)***

```
                    @"ToFromiPad100.html" withDataURL:url];
         break;
       }
       case 1: {
         url = [NSURL URLWithString:
           @"http://nealgoldstein.com/ToFromiPad101.html"];
         [self saveAirportData:
                    @"ToFromiPad101.html" withDataURL:url];
         break;
       }
       case 2: {
         url = [NSURL URLWithString:
           @"http://nealgoldstein.com/ToFromiPad102.html"];
         [self saveAirportData:
                    @"ToFromiPad102,html" withDataURL:url];
         break;
       }
     }
   return url;
 }
```

Okay, here's the blow-by-blow for Listing 18-8.

1. **When a message is sent to the model to return the data the view needs to display, it's passed the number of the segmented control that was tapped (`Train`, `Taxi`, `Other`).**

   It's the model's responsibility to decide what data is required here.

2. **The data for each of the choices in the segmented control is on a Web site — `www.nealgoldstein.com`, to be precise. The method constructs the `NSURL` object that the Web view uses to load the data.**

   The NSURL object is nothing fancy. To refresh your memory, it's simply an object that includes the utilities necessary for downloading files or other resources from Web and FTP servers or accessing local files.

   ```
   NSURL *url = [NSURL URLWithString:
         @"http://nealgoldstein.com/ToFromiPad100.html"];
   ```

3. **Then the `saveAirportData:` message is sent:**

   ```
   [self saveAirportData:
                       @"ToFromiPad100" withDataURL: url];
   ```

   The `saveAirportData` method in Listing 18-9 downloads and saves the file containing the latest data for whatever transportation method (`Taxi`, for example) the user selected. It's what will be displayed in the current view, and it'll be used later in Chapter 19 when I show you how to use cached data when the user doesn't want to access the data online. (Add the `saveAirportData` method to `Airport.m`.)

Listing 18-9: Saving Airport Data

```
- (void) saveAirportData:(NSString *) fileName
                              withDataURL:(NSURL *) url {

  NSData *dataLoaded = [NSData
                             dataWithContentsOfURL:url];
  if (dataLoaded == NULL)
                      NSLog(@"Data not found %@", url);
  NSArray *paths = NSSearchPathForDirectoriesInDomains
            (NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  NSString *filePath = [documentsDirectory
               stringByAppendingPathComponent:fileName];
  [dataLoaded writeToFile:filePath atomically:YES];
}
```

As you can see, the first thing I do here is go back out to the URL to get the data again. The method `dataWithContentsOfURL:` does what it sounds like. If you're not familiar with the `NSData` class, it and its mutable subclass (`NSMutableData`) are simply object-oriented wrappers — objects that hold any kind of data. Although this approach isn't particularly efficient, since I am getting the data twice, it allows me to show you how to load data from a Web site as an `NSData` object that you can later do something with. I've also added an `NSLog` message if the data can't be found. This is a placeholder for error-handling that I've left as an exercise for you to do on your own.

Writing to the file system on the iPad is pretty simple: You tell the system which directory to put the file in, specify the file's name, and then pass that information to the `writeToFile` method.

1. **You get the path to the `Documents` directory.**

   ```
   NSArray *paths = NSSearchPathForDirectoriesInDomains
             (NSDocumentDirectory, NSUserDomainMask, YES);
   NSString *documentsDirectory =[paths objectAtIndex:0];
   ```

   On the iPad, you really don't have much choice about where the file goes. Although there's a `/tmp` directory, I'm going to place this file in the `Documents` directory — because (as I explain in Chapter 2), this is part of my application's sandbox, so it's the natural home for all the app's files.

   `NSSearchPathForDirectoriesInDomains:` returns an array of directories; because I'm only interested in the `Documents` directory, I use the constant `NSDocumentDirectory`, and because I'm restricted to my home directory, `/sandbox`, the constant `NSUserDomainMask` limits the search to that *domain.* There will be only one directory in the domain, so the one I want will be the first one returned.

2. **You create the complete path by appending the path filename to the directory.**

   ```
   NSString *filePath = [documentsDirectory
                stringByAppendingPathComponent:fileName];
   ```

   `stringByAppendingPathComponent;` precedes the filename with a path separator (`/`) if necessary.

   TIP

   Unfortunately, this doesn't work if you're trying to create a string representation of a URL.

3. **You write the data to the file.**

   ```
    [dataLoaded writeToFile:filePath atomically:YES];
   ```

   `writeToFile:` is an `NSData` method and does what it implies. I'm actually telling the array here to write itself to a file, which is why I decided to save the location in this way in the first place. A number of other classes implement this method, including `NSData`, `NSDate`, `NSNumber`, `NSString`, and `NSDictionary`. You can also add this behavior to your own objects, and they could save themselves — but I don't get into that here. The `atomically` parameter first writes the data to an auxiliary file, and when that's successful, it's renamed to the path you've specified. This guarantees that the file won't be corrupted even if the system crashed during the write operation.

You may have noticed that I did not tell you to add the `saveAirportData: withDataURL:` declaration. The next section tells you why.

Making methods "private"

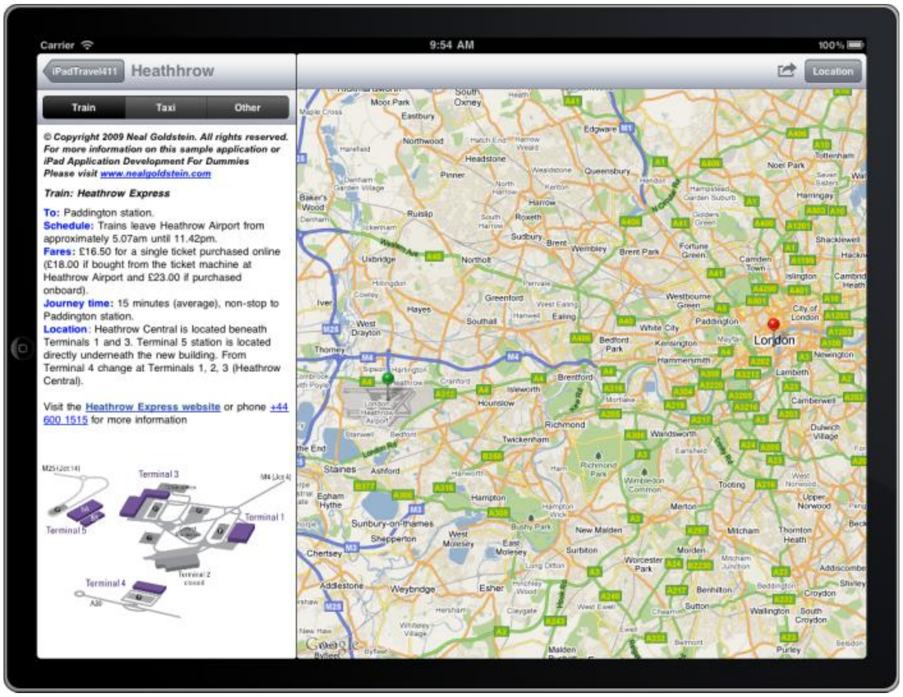
If you're coming from C++, you probably want this method to be private — there is no reason for any "external" object to send this message to the Airport object. The problem is that there's no private construct in Objective-C. But there is a workaround. To hide it, move its declaration to the implementation file and create an Objective-C class extension. Although categories are a way to add methods to an existing class (even to one to which you do not have the source, which isn't relevant here but is good to know), class extensions are like "anonymous" categories, and the methods they declare must be implemented in the main `@implementation` block for that class. This makes these methods (almost) "invisible" to other classes. Any more about categories is beyond the scope of this book, but I do explain them in detail in *Objective-C For Dummies*.

So don't declare `saveAirportData: withDataURL:`. Instead, add the following code to `Airport.m`, right before the `@implementation` statement.

```
@interface Airport  ()
- (void) saveAirportData:(NSString *) fileName
                              withDataURL:(NSURL *) url;
@end
```

As I said, you will use this saved data in Chapter 19.

When all is said and done (and with a bit more code), you will get what you see in Figure 18-2 when the user selects Heathrow in the Master view and keeps a map visible in the Detail view.



© Copyright 2009 Neal Goldstein. All rights reserved. For more information on this sample application or iPad Application Development For Dummies Please visit www.nealgoldstein.com

Train: Heathrow Express

To: Paddington station.
Schedule: Trains leave Heathrow Airport from approximately 5.07am until 11.42pm.
Fares: £16.50 for a single ticket purchased online (£18.00 if bought from the ticket machine at Heathrow Airport and £23.00 if purchased onboard).
Journey time: 15 minutes (average), non-stop to Paddington station.
Location: Heathrow Central is located beneath Terminals 1 and 3. Terminal 5 station is located directly underneath the new building. From Terminal 4 change at Terminals 1, 2, 3 (Heathrow Central).

Visit the Heathrow Express website or phone +44 600 1515 for more information

Figure 18-2: Airport transportation information and a map.

Selecting the airport

In Chapter 16, you add the ability for the user to make a selection in the Master view by adding the necessary code to `tableView:didSelectRowAtIndexPath`. You do the same thing here. Currently, you should see three `case` statements you have yet to implement:

```
case 4:
  //do something
  break;
case 5:
  //do something
  break;
case 6:
  //do something
  break;
```

Replace the `//do something` in each with the following:

```
targetController = [[menuList objectAtIndex:menuOffset]
                             objectForKey:kControllerKey];
if ([targetController isKindOfClass:[NSNull class]]) {
 targetController = [[AirportController alloc]
     initWithDestination:kAppDelegate.destination
                                             airportID:1];
 [[menuList  objectAtIndex:menuOffset]
       setObject:targetController forKey:kControllerKey];
 [targetController release];
}
```

You'll also need to import `AirportController.h`. (You should change the `airportID` for each.)

This is exactly what you did to add the view controllers in Chapter 16. But now, I also want you to add the code in bold in Listing 18-10 to `tableView:didSelectRowAtIndexPath`. ***Note:*** You need to add it to the very last part of the method (after the `switch` block) where you are placing what was the last section of code into the `else` clause of the `if` statement you are adding.

Listing 18-10: tableView:didSelectRowAtIndexPath

```
  if ([targetController isKindOfClass:
                            [AirportController class]])
    [[self navigationController] pushViewController:target
          Controller
                                             animated:YES];
  else {
    [[kAppDelegate splitViewController]
     setViewControllers:
      [NSArray arrayWithObjects:self.navigationController,
                              targetController, nil]];
    kAppDelegate.splitViewController.delegate =
                                        targetController;
    [targetController computeFramesForOrientation:
                               [self appOrientation]];
    if (currentController)
                    [self setupToolbar:targetController];
    if (currentController.popoverController) {
      targetController.popoverController =
                      currentController.popoverController;
      [currentController.popoverController
                             dismissPopoverAnimated:YES];
    }
```

```
     currentController = targetController;
   }
 }
```

Because the Airport controller will display its view in the Master view, you're not going to do what you did with the other views back in Chapter 16, where you replaced the Detail view in the Split view controller. Instead, you're going to determine whether the Target controller you created is an Airport controller.

```
if ([targetController isKindOfClass:
                            [AirportController class]])
```

`isKindOfClass` is an `NSObject` protocol method that `NSObject` implements. It returns a Boolean value that indicates whether the object is an instance of a given class (in this case, the `AirportController` class) or an instance of any class that inherits from that class.

If it is, you push it on the Master view controller's navigation controller stack.

```
[[self navigationController]
       pushViewController:targetController animated:YES];
```

Pushing the view on the stack will replace the current view controller with the new one. You might think this is the same thing as replacing the Detail view controller in the Split view controllers — except this time you're messing with the Master view.

Well, that's not really true.

Let me point you back to Chapter 15, where I was explaining the following bit of code:

```
  [[kAppDelegate splitViewController] setViewControllers:
    [NSArray arrayWithObjects:self.navigationController,
                                  mapController, nil]];
```

Back then I said, "You may be curious why you're using `self.navigationController` and not something that resembles the `RootViewController` here (like `self`). Answering that question involves understanding navigation using view controllers, and this is not the right place to do that. So be patient for now; you learn all about navigation controllers in Chapter 18."

Well, guess what? You're now in Chapter 18. *The secret can be revealed.*

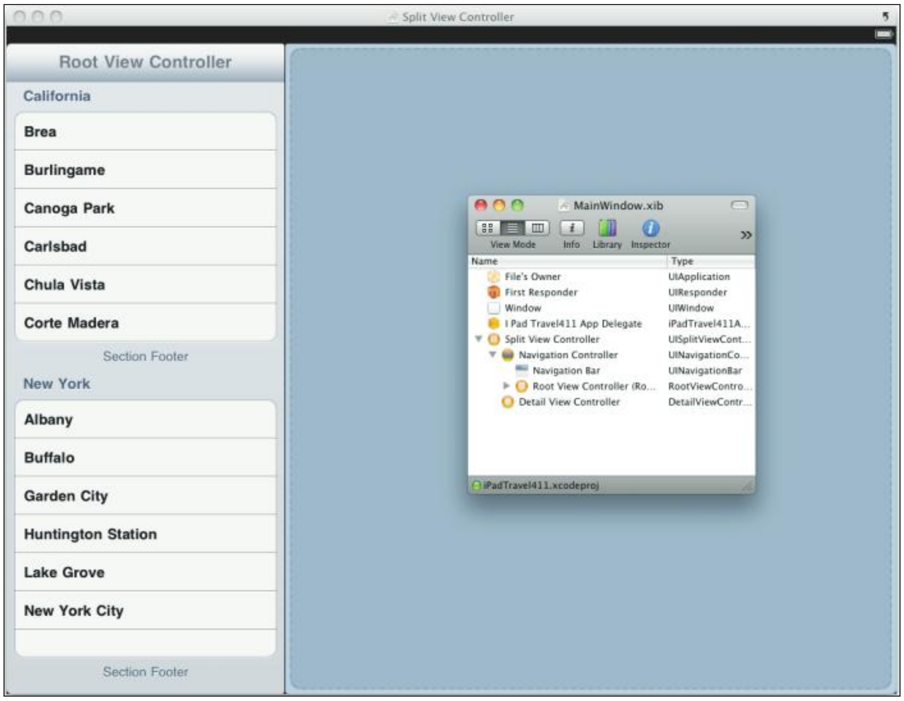
In your app, the Master view controller is not the `RootViewController` — it's a `UINavigationController`. The `UINavigationController` class implements a specialized view controller that manages the navigation of hierarchical content (which is what you need in order to go back and forth in the Master view from the Table view to another controller like `AirportController`. This specialized view controller is responsible for the *navigation bar* and a Back button — the kind you see so often in the iPad and especially iPhone applications — and the handling of the Back request.

You'll notice, by the way, that the `UIViewController` even has a `navigationController` property so it can find its navigation controller.

```
@property(nonatomic, readonly, retain)
          UINavigationController *navigationController
```

So all you need to do to implement the hierarchal navigation you need in the Master view is to tell the navigation controller to push the `AirportController` onto its stack. (I explain that in a second.) It does what it's told, displays the new view, and adds the Back button to the navigation bar (refer to Figures 18-1 and 18-2) so the user can return to the previous view control (in this case, the Table view) in the Master view.

This navigation controller is created for you by the template. When it creates the Split view controller, it initializes the Master view with a navigation controller, which in turn manages the `RootViewController`. You can see that for yourself in the nib file in Figure 18-3 or by double clicking `MainWindow.xib` in the Resources group of your project.



Name	Type
File's Owner	UIApplication
First Responder	UIResponder
Window	UIWindow
I Pad Travel411 App Delegate	iPadTravel411A...
Split View Controller	UISplitViewCont...
Navigation Controller	UINavigationCo...
Navigation Bar	UINavigationBar
Root View Controller (Ro...	RootViewContro...
Detail View Controller	DetailViewContr...

Figure 18-3: It's all about the navigation controller.

There as you can see, in the Master view side of the Split view, it says Root View Controller.

Now you can get back to what this `pushViewController animated:` does when you ask the navigation controller to do it to the `AirportController`.

Navigating the Navigation Controller

Table views are paired with navigation bars in order to give users the option of returning to a view higher up in the hierarchy (in this case, the Master view). In fact, that bar bearing the name iPadTravel411 in Figures 18-1 and 18-2 is the navigation bar that enables a user to navigate the hierarchy.

Here's what you need to know in order to make navigation bars work for you:

- The view below the navigation bar presents the current level of data.
- A navigation bar includes a title for the current view.
- If the current view is lower in the hierarchy than the top level, a Back button appears on the left side of the bar; the user can tap it to return to the previous level. The text in the Back button tells the user what the previous level was. In this case, it's the application's main view, so you'll see the previous view controller's title — iPadTravel411.
- A navigation bar may also have an Edit or Add button (on the right side) — used to enter editing mode for the current view or adding an entry respectively — or even custom buttons such as a button to launch a popover.

When the user taps a row of the Table view to get the Heathrow Express information, say, the root view controller pushes the next view controller onto the stack. The new controller's view (the Heathrow Express information) slides into place, and the navigation bar items are updated appropriately. When the user taps the Back button on the navigation bar, the current view controller pops off the stack, the Heathrow Express Information view slides off the screen, and the user lands (so to speak) back in the main (Table) view.

The navigation controller maintains a stack of view controllers, one for each of the views displayed, starting with the Master view controller. The only thing that makes the `RootViewController` (Master view controller) special is that it is the very first view controller that the Navigation controller pushes onto its stack when a user launches the application; it remains active until the user selects the next view to look at.

A *stack* is a commonly used data structure that works on the principle of last in, first out. Imagine an "ideal" boarding scenario for an airplane: You would start with the last seat in the last row and board the plane in back-to-front order until you got to the first seat in the first row — that would be the seat for the last person to board. When you got to your destination you'd deplane (is that really a word?) in the reverse order. That last person on — the person in row one, seat one — would be the first person off.

A computer stack is pretty much the same. Adding an object is called a *push* — in this case, when you select Heathrow, the view controller for the Heathrow view is pushed onto the stack. Removing an object is called a *pop* — touching the Back button pops the view controller for the Heathrow view. When you pop an object off the stack, it's always the last one you pushed onto it. The controller that was there before the push is still there and now becomes the active one — in this case, it's the `RootViewController`.

The navigation bar back button

This navigation bar Back button is of course different from the Back button you created in the Web view. You created that one to get back from a Web page to a previous Web page. This one takes you back from one view controller to a previous view controller.

The other Back button

In my design, I wanted the user to be able to tap a link in the Airport views to access a Web site such as the Heathrow Express one to get more information. (You can see such a link on the left in Figure 18-4.) When I do that, the iPadTravel411 application *replaces* the content of the view, instead of *creating* a new view controller. Tapping the link doesn't change the controller in any way, so the left button doesn't change; you can't use it to get back to a previous view — you only go back to the main view, as the control text tells you. To solve this, just as you did with the other view controllers that enable you to go out to the Web, you created another button and labeled it "Airport" so the user knows he or she can use it to get back to the previous view.

In the Weather controller and the other Web views, I showed you how to create this Back button to return from a selected link. You'll need to do the same thing in the `AirportController`, but instead of putting a button on a toolbar, you add a button to the navigation bar instead. Otherwise, this code works exactly the same way as it did in Chapter 16, so I won't spend too much time on it here.

Add the code in Listing 18-11 to `AirportController.m`.

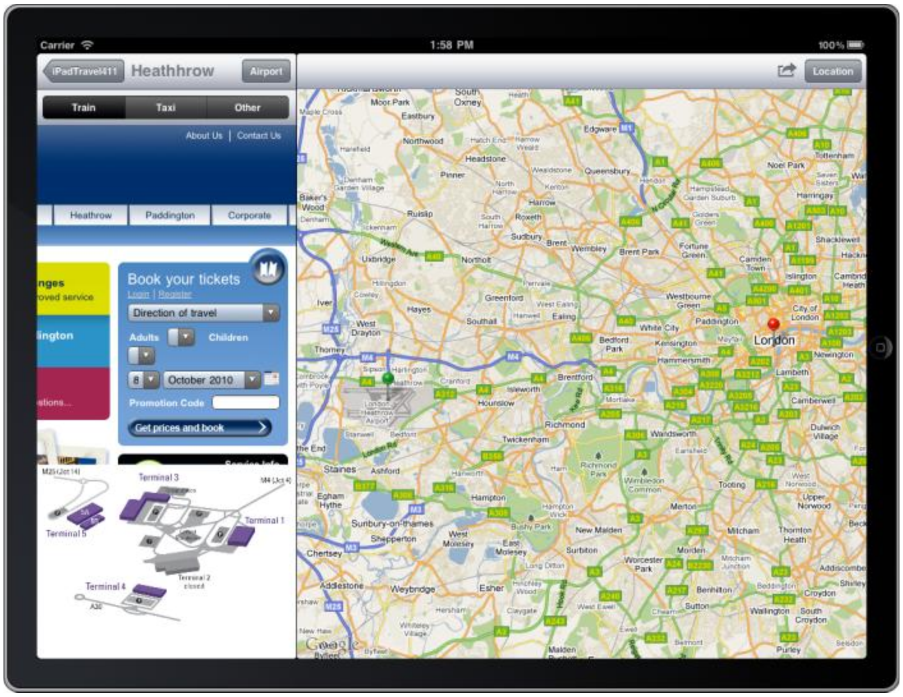


Figure 18-4: Going back to the Airport.

Listing 18-11: Still Cruising the Web

```
- (BOOL)webView:(UIWebView *)webView
   shouldStartLoadWithRequest:(NSURLRequest *)request
  navigationType:(UIWebViewNavigationType)navigationType {

  if (navigationType ==
                         UIWebViewNavigationTypeLinkClicked) {
    if (!backButton) {
      backButton =
      [[UIBarButtonItem alloc] initWithTitle:@"Airport"
         style:UIBarButtonItemStyleBordered target:self
                              action:@selector(goBack:)];
      self.navigationItem.rightBarButtonItem = backButton;
      [backButton release];
    }
  }
  return YES;
}


- (IBAction) goBack:(id)sender{
```

(continued)

Listing 18-11 ***(continued)***

```
   [webView goBack];
}

- (void)webViewDidFinishLoad:(UIWebView *) aWebView {

  if (([aWebView  canGoBack] == NO) && (backButton)) {
    self.navigationItem.rightBarButtonItem = nil;
    theToolbar.hidden = NO;
    backButton = nil;
  }
}
```

As I said, this is the same thing you do in the `WeatherController`, `CityController`, and `CurrencyController` examples — except instead of putting the button on the toolbar, you add it as the right bar button on the navigation bar, as follows:

```
self.navigationItem.rightBarButtonItem = backButton;
```

Getting Rid of a Pesky Compiler Warning

One last thing. You may notice the following error in your Build Results window:

```
The 'view' outlet of 'File's Owner' is connected to 'View'
     but 'view' is no longer defined on AirportController.
```

This happened when you made `AirportController` a subclass of `DetailViewController`.

To fix this error, double-click `AirportController.xib` and then right-click the File's Owner icon in the `AirportController.xib` window to bring up the (by now rather familiar) connections panel.

Even though the little circle is already filled in, drag from the little circle next to View in Outlets to the view in the View window or even the View icon in the `AirportController.xib` window.

Save the file, and the error goes away.

Chapter 19

Enhancing the User Experience

In This Chapter

- Getting back to where you once belonged
- Avoiding bankruptcy because of exorbitant roaming charges

One of the things about multitasking is that most of the time you don't *really* need to know how it works, but there are some things you need to be aware of so you can keep out of trouble. In this chapter, I have you add two more pieces of functionality — saving the place the user is in your app when he or she leaves it (referred to by computer types as *saving state*) and adding user preferences. Both of these exercises will give you some useful insight into how multitasking works, what you need to be aware of when an app may be running in background, and what really goes on when an app is moved to the background or even terminated.

Saving and Restoring State

In computer science terms, a *state* is a unique configuration of information in a program or machine. For your purposes, if the user leaves the application because he or she decided to play a game, you want the user, when he or she resumes the application, to be able to start exactly where he or she left off.

You may be wondering, "Why do I have to save state? Didn't you say way back in Chapter 8 that under iOS 4, when the user taps the Home button on a device, the application is suspended and when the user 'launches' it again, it starts right back up where it left off?"

Yes, I did say that, but there are two situations where that won't happen:

- The user is running a device that doesn't support multitasking.
- The device does support multitasking, but your app has been purged from memory.

If either of these situations crops up, you have to make sure you've saved any unsaved data — as well as the current state of your application — if you want to restore the application to its previous state the next time the user launches it. Now, in situations like this one, you have to use common sense to decide what *state* really means. Generally, you wouldn't need to restore the application to where the user last stopped in a scrollable list, for example. For purposes of explanation, I chose to save the last category view that the user selected in the Master view, which corresponds to a row in a section in the Table view.

I don't save the view that the user was in if he or she selected Airport in the Master view. I'll leave that addition as an exercise for you to do on your own.

So where do you save the state?

In those devices that don't support multitasking, when the user taps the Home button, iOS terminates your application and returns to the Home screen. The `applicationWillTerminate:` message is sent, and your application is terminated — no ifs, ands, or buts. That's where you'd do any necessary saving of state for devices that don't support multitasking.

In devices that *do* support multitasking, the `applicationWillTerminate:` message is not sent. Instead, when your app is moved into the background, the `applicationDidEnterBackground:` message is sent and you have to save any changes of state in this method in case your application is later purged.

Saving state information

Here's the sequence of events that go into saving the state:

1. **Add a new instance variable `lastView` and declare the `@property` in the `iPhoneTravel411AppDelegate.h` file. Also add the `saveState` declaration. (You'll implement that shortly.)**

   See the code in bold in Listing 19-1.

   I explain properties in Chapter 8.

   As you can see, `lastView` is a mutable array. You'll save the section as the first element in the array and the row as the second element. Because the array is mutable, it'll be easier to update when the user selects a new row in a section.

2. **Add the `@synthesize` statement to the `iPadTravel411AppDelegate.m` file to tell the compiler to create the accessors for you.**

   This addition is shown in Listing 19-2. (You guessed it — new stuff is bold.)

3. **Define the filename you'll use when saving the state information in the `Constants.h` file.**

   ```
   #define kState  @"LastState.state"
   ```

   You also have to add the `#import "Constants.h"` statement to the `iPadTravel411AppDelegate.h` file.

4. **Save the section and row that the user last tapped in the `iPadTravel411AppDelegate`'s `lastView` instance variable by adding the following code to the beginning of the `tableview:didSelectRowAtIndexPath:` method in the `RootViewController.m` file, as shown in Listing 19-3.**

   The `tableview:didSelectRowAtIndexPath:` method is called when the user taps a row in a section. The section and row information are in the `indexPath` argument of the `tableview:didSelectRowAtIndexPath:` method. All you have to do to save that information is to save the `indexPath.section` as the first array entry and save the `indexPath.row` as the second. (The reason I do it this way will become obvious when I show you how to write this to a file.)

   REMEMBER

   You're not going to save the state if the user is in an Airport view. As you can see, you determine that in Listing 19-3 by checking to see whether the selection was in the first section:

   ```
   if (indexPath.section == 0) {
   ```

5. **Save the section and row in the `saveState` method by adding the code in Listing 19-4 to `iPadTravelAppDelegate.m`.**

   In `saveState`, I'm saving the `lastView` instance variable (which contains the last section and row the user tapped) to the file `kState`, which is the constant I defined in Step 3 to represent the filename `LastState.state`.

   As you can see, reading or writing to the file system on the iPad is pretty simple: You tell the system which directory to put the file in, specify the file's name, and then pass that information to the `writeToFile` method. You've already done something similar in Chapter 18, and the blow-by-blow can be found there.

6. **Send the `saveState` message in both the `applicationDidEnterBackground:` and `applicationWillTerminate:` methods.**

   For some reason, in this version of the SDK, the template doesn't include the `applicationDidEnterBackground:` method for this template, although it does for the iPhone templates. Go figure. (Who knows? By the time you read this, things may have changed.) In Listing 19-5 add `applicationDidEnterBackground:` to `iPhoneTravel411AppDelegate.m`, and in Listing 19-6 update `applicationWillTerminate:` in `iPhoneTravel411AppDelegate.m`.

Listing 19-1: Adding the Instance Variable to iPadTravel411AppDelegate.h

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


@class RootViewController;
@class DetailViewController;
@class Destination;

@interface iPadTravel411AppDelegate :
                        NSObject <UIApplicationDelegate> {

    UIWindow *window;
    UISplitViewController  *splitViewController;
    RootViewController     *rootViewController;
    DetailViewController   *detailViewController;
    Destination            *destination;
    NSMutableArray         *lastView;
}

@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet
          UISplitViewController *splitViewController;
@property (nonatomic, retain) IBOutlet RootViewController
                                     *rootViewController;
@property (nonatomic, retain) IBOutlet
               DetailViewController *detailViewController;
@property (nonatomic, retain) Destination *destination;
@property (nonatomic, retain) NSMutableArray *lastView;
- (void) saveState;
@end
```

Listing 19-2: Adding the @synthesize to the iPadAppTravelDelegate.m

```
@implementation iPadTravel411AppDelegate

@synthesize window, splitViewController,
                 rootViewController, detailViewController;
@synthesize destination;
@synthesize lastView;
```

Listing 19-3: Saving indexPath

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

  if (indexPath.section == 0) {
    [kAppDelegate.lastView
        replaceObjectAtIndex:0 withObject:
            [NSNumber numberWithInteger:indexPath.section]];
    [kAppDelegate.lastView
        replaceObjectAtIndex:1 withObject:
             [NSNumber numberWithInteger:indexPath.row]];
}
```

Listing 19-4: Adding saveState

```
- (void) saveState {

 NSArray *paths = NSSearchPathForDirectoriesInDomains
             (NSDocumentDirectory, NSUserDomainMask, YES);
 NSString *documentsDirectory = [paths objectAtIndex:0];
 NSString *filePath = [documentsDirectory
                  stringByAppendingPathComponent:kState];
 [lastView writeToFile:filePath atomically:YES];
}
```

Listing 19-5: Add applicationDidEnterBackground:

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

  [self saveState];
}
```

Listing 19-6: Update updateapplicationWillTerminate:

```


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

  [self saveState];
}
```

You also have to add `#import "Constants.h"` to `iPadTravel411AppDelegate.m`.

Restoring the state

Now that I've saved the state, I need to restore it when the application is launched. I do this in `RootViewController`'s `viewDidLoad` method:

```
NSIndexPath *indexPath = [NSIndexPath indexPathForRow:1
                                             inSection:0];
[self tableView:((UITableView *) self.tableView)
                        didSelectRowAtIndexPath:indexPath];
```

All I'm going to do is remove the hard-coded values and use `lastView`, which is the index path of the last view you stored in the previous section. Update `viewDidLoad` in `RootViewController.m` to match what you see in Listing 19-7. This code should be at the very end of the method. (Bold underline italic (BUI) shows deletions, the bolded code shows additions.)

Listing 19-7: Updating viewDidLoad

```
- (void)viewDidLoad {
…
   NSIndexPath * indexPath = 
                  [NSIndexPath indexPathForRow:1 inSection:0];

   NSArray *paths = NSSearchPathForDirectoriesInDomains
              (NSDocumentDirectory, NSUserDomainMask, YES);
   NSString *documentsDirectory = [paths objectAtIndex:0];
   NSString *filePath = [documentsDirectory
                    stringByAppendingPathComponent:kState];
   kAppDelegate.lastView =[[NSMutableArray alloc]  initWith
           ContentsOfFile:filePath];

   NSIndexPath *indexPath;
   if ( kAppDelegate.lastView != nil) {
      indexPath = [NSIndexPath indexPathForRow:
      [[kAppDelegate.lastView objectAtIndex:1]intValue]
          inSection:[[kAppDelegate.lastView
                                 objectAtIndex:0] intValue]];
   }
   else {
     kAppDelegate.lastView = [[NSMutableArray
       arrayWithObjects:[NSNumber numberWithInteger:0],
               [NSNumber numberWithInteger:1],nil] retain];
    indexPath =
               [NSIndexPath indexPathForRow:1 inSection:0];
   }
   [self tableView:((UITableView *) self.tableView)
                          didSelectRowAtIndexPath:indexPath];
}
```

You need to read in the file (`LastState.state`) that contains the `lastView` information. Reading is the mirror image of writing. You create the complete path, including the filename, just as you did when you saved the file. This time you send the `initWithContentsOfFile:` message instead of `writeToFile:`, which allocates the `lastView` array and initializes it with the file. If the result is `nil`, there's no file, meaning that this is the first time the application is being used. In that case, you create the array with the value of section and row set to `-0` and `-1`.

TIP

`initWithContentsOfFile:` is an `NSData` method similar to `writeTo-File:`. The classes that implement `writeToFile:` and those that implement `initWithContentsOfFile:` are the same.

Respecting User Preferences

Figure 19-1 shows you the Settings screen for my iPadTravel411 application. There you can see that I've added the one preference for iPadTravel411, and in this chapter, I show you how to implement it. Any other preferences you might come up with I leave up to you.

The Use Stored Data preference tells the application to use the last version of the data that it accessed, rather than going out on the Internet for the latest information. Even though this does violate my I Want The Most Up To Date Information principle, it can save the user from excessive roaming charges, depending on his or her data plan. (This is, of course, only applicable for iPads with 3G and a data plan.)

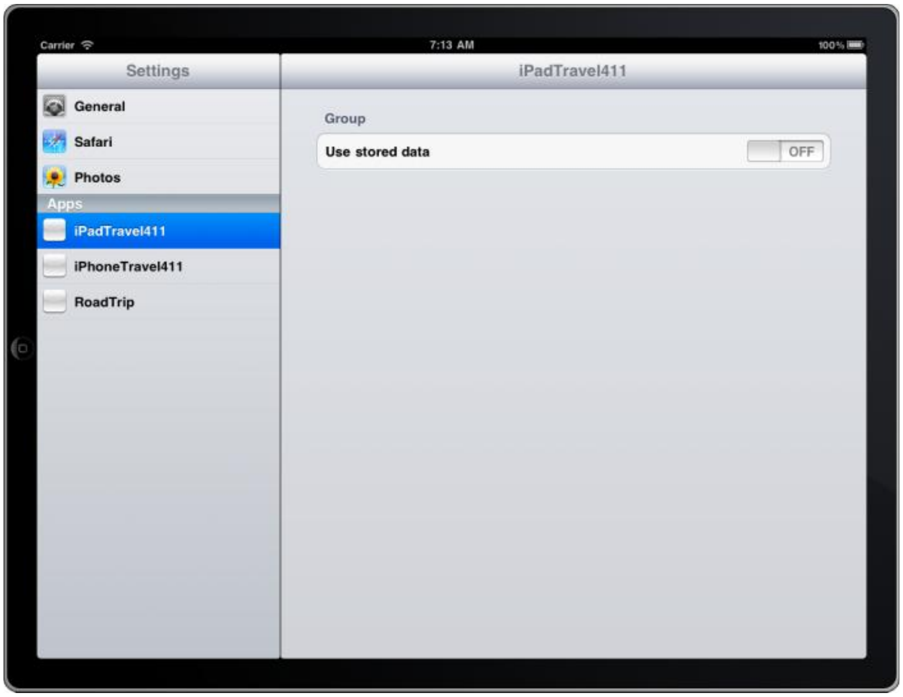
Of course, there are other reasons the user might have to use stored data — an Internet connection may not be available, for example. I'll leave that as an exercise for the reader.

When the user has selected Use Stored Data, I refer to the device as being in *stored data mode,* or *offline.*

TIP

No doubt it's way cool to put user preferences in Settings. Some programmers abuse this trick, though; they make you go into Settings, when it's just as easy to give the user a preference-setting capability within the program itself (as you did with the DeepThoughts app in the first part of this book). You should put something in Settings only if the user changes it infrequently. In this case, stored data doesn't change often so the Use Stored Data preference definitely belongs in Settings.

The Settings application uses a property list, called `Root.plist`, found in the Settings bundle inside your application. The Settings application takes what you put in the property list and builds a Settings section for your application in its list of application settings as well as the views that display and enable the user to change those settings. The next sections spell out how to put that Settings section to work for you.



Use stored data OFF

Figure 19-1: The required preferences.

Adding a Settings bundle to your project

For openers, you have to add a Settings bundle to your application. Here are the moves:

1. **In the Groups & Files list (on the left in the Xcode Project window), select the iPadTravel411 icon and then choose File➪New File from the main menu or press ⌘+N.**

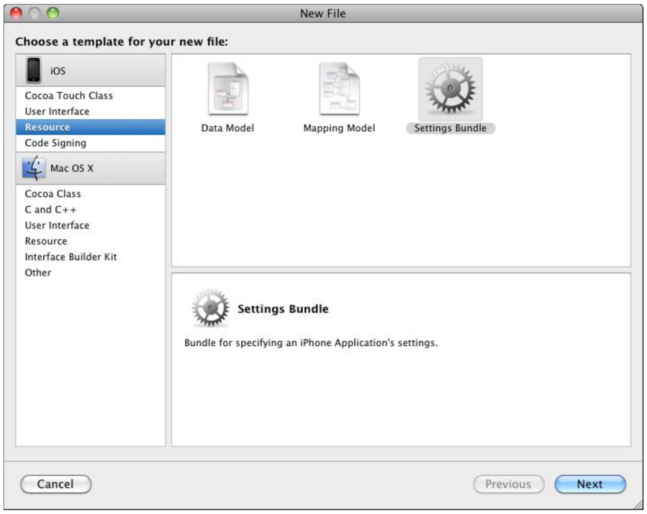
   The New File dialog appears.

2. **Choose Resource under the iOS heading in the left pane and then select the Settings Bundle icon, as shown in Figure 19-2.**
3. **Click the Next button.**
4. **Choose the default name of `Settings.bundle` and then press Return (Enter) or click Finish.**

   You should now see a new item called `Settings.bundle` in the Groups & Files list.

5. **Click the triangle to expand the `Settings.bundle`.**

   You see the `Root.plist` file as well as an `en.lproj` folder — the latter is used for dealing with localization issues, which I discuss in Chapter 14.



Settings Bundle

Bundle for specifying an iPhone Application's settings.

Figure 19-2: Creating the application bundle.

Setting up the property list

Property lists are widely used in iPad applications because they provide an easy way to create structured data using named values for a number of object types.

TIP

In my own applications, I use property lists extensively as a way to *parameterize* view controllers and models — I have the initialization data in a plist and initialize objects with that data.

Property lists all have a single root node — a Dictionary, which means it stores items using a key-value pair, just as an `NSDictionary` does. All dictionary entries must have both a key and a value. In this dictionary, there are two keys:

- `StringsTable`
- `PreferenceSpecifiers`

The value for the first entry is a string — the name of a strings table used for localization, which I don't get into here. The second entry is an array of dictionaries — one dictionary for each preference. (You probably need some time to wrap your head around that one; it'll become clearer as I take you through it.)

`PreferenceSpecifiers` is where you put a toggle switch so the user can choose to use (or not use, because it's a toggle) only stored data — I refer to that choice later as *stored data mode.* Here's how it's done:

1. **In the Groups & Files list of the Project window, select the disclosure triangle next to the `Settings.bundle` file to reveal the `Root.plist` file and then double-click the `Root.plist` file to open it in a separate window.**

   Okay, you don't *really* have to do this, but I find it easier to work with this file when it's sitting in its own window.

2. **In the `Root.plist` window you just opened, expand the disclosure triangles next to all the nodes by clicking all those triangles, as shown in Figure 19-3.**

   You can also expand everything by holding down the Option key when clicking a closed disclosure triangle, like the one next to `PreferenceSpecifiers`.

3. **Under the `PreferenceSpecifiers` heading in the `Root.plist` window, move to Item 0.**

   `PreferenceSpecifiers` is an array designed to hold a set of dictionary nodes, each of which represents a single preference. For each item listed in the array, the first row under it has a key of `Type`; every property list node in the `PreferenceSpecifiers` array must have an entry with this key, which identifies what kind of entry this is. The `Type` value for the current Item 0 — `PSGroupSpecifier` — is used to indicate that a new group should be started. The value for this key actually acts like a section heading for a Table view (like you created in Chapter 14). Double-click the value next to `Title` and delete the default `Group`, as I have in Figure 19-4 (or you can put in `IPadTravel411 Preferences`, or be creative if you like).

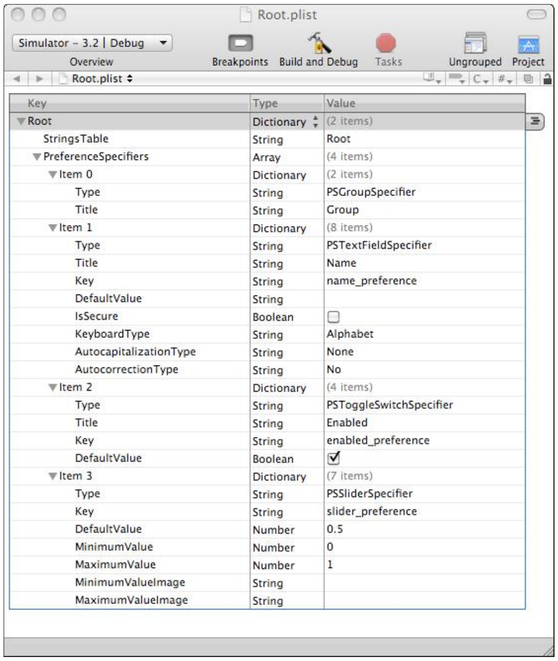
4. **Seeing that Item 2 is already defined as a toggle switch, you can just modify it by changing the `Title` value from `Enabled` to `Use stored data` and the key from `enabled_preference` to `useStoredDataPreference`.**

   This is the key you'll use in your application to access the preference.

5. **Continue your modifications to Item 2 by deselecting the `Boolean` check box next to `DefaultValue`.**

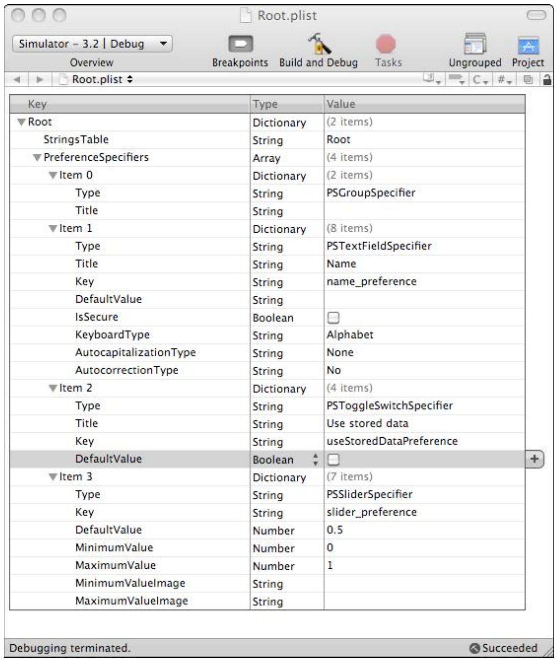
   I want the Use Stored Data preference initially set to Off because I expect most people will still want to go out on the Internet for the latest information, despite the high roaming charges involved.

   When you're done, the `Root.plist` window should look like Figure 19-4.



Key	Type	Value
▼Root	Dictionary	(2 items)
StringsTable	String	Root
▼PreferenceSpecifiers	Array	(4 items)
▼Item 0	Dictionary	(2 items)
Type	String	PSGroupSpecifier
Title	String	Group
▼Item 1	Dictionary	(8 items)
Type	String	PSTextFieldSpecifier
Title	String	Name
Key	String	name_preference
DefaultValue	String	
IsSecure	Boolean	☐
KeyboardType	String	Alphabet
AutocapitalizationType	String	None
AutocorrectionType	String	No
▼Item 2	Dictionary	(4 items)
Type	String	PSToggleSwitchSpecifier
Title	String	Enabled
Key	String	enabled_preference
DefaultValue	Boolean	☑
▼Item 3	Dictionary	(7 items)
Type	String	PSSliderSpecifier
Key	String	slider_preference
DefaultValue	Number	0.5
MinimumValue	Number	0
MaximumValue	Number	1
MinimumValueImage	String	
MaximumValueImage	String	

Figure 19-3: The default Root.plist file preferences.



Key	Type	Value
▼Root	Dictionary	(2 items)
StringsTable	String	Root
▼PreferenceSpecifiers	Array	(4 items)
▼Item 0	Dictionary	(2 items)
Type	String	PSGroupSpecifier
Title	String	
▼Item 1	Dictionary	(8 items)
Type	String	PSTextFieldSpecifier
Title	String	Name
Key	String	name_preference
DefaultValue	String	
IsSecure	Boolean	☐
KeyboardType	String	Alphabet
AutocapitalizationType	String	None
AutocorrectionType	String	No
▼Item 2	Dictionary	(4 items)
Type	String	PSToggleSwitchSpecifier
Title	String	Use stored data
Key	String	useStoredDataPreference
DefaultValue	Boolean	☐
▼Item 3	Dictionary	(7 items)
Type	String	PSSliderSpecifier
Key	String	slider_preference
DefaultValue	Number	0.5
MinimumValue	Number	0
MaximumValue	Number	1
MinimumValueImage	String	
MaximumValueImage	String	

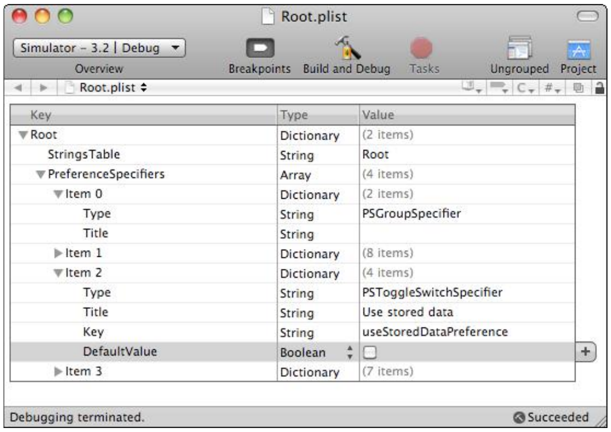
Debugging terminated. Succeeded

Figure 19-4: Preferences for IPhone-Travel411.

6. **Collapse the disclosure triangles next to items 1 and 3 (as shown in Figure 19-5) and then select those items one by one and delete them.**

   WARNING!

   The item numbers do change as you delete them, so be careful. That's why you need to leave the preference item you care about open, so you can see that you shouldn't delete it. Fortunately, Undo is supported here; if you make a mistake, press ⌘+Z to undo the delete.



Key	Type	Value
▼Root	Dictionary	(2 items)
StringsTable	String	Root
▼PreferenceSpecifiers	Array	(4 items)
▼Item 0	Dictionary	(2 items)
Type	String	PSGroupSpecifier
Title	String	
▶Item 1	Dictionary	(8 items)
▼Item 2	Dictionary	(4 items)
Type	String	PSToggleSwitchSpecifier
Title	String	Use stored data
Key	String	useStoredDataPreference
DefaultValue	Boolean	☐
▶Item 3	Dictionary	(7 items)

Figure 19-5: Delete these items.

Reading Settings in the Application

After you set it up so your users can let their preferences be known in Settings, you need to read those preferences back into the application. You do that in the `iPadTravel411AppDelegate`'s `application:didFinishLaunchingWithOptions:` method. But first, a little housekeeping.

You need to add `useStoredData` as an instance variable and then declare it as a property in the `iPadTravel411AppDelegate.h` file.

```
BOOL  useStoredData;
...
@property (nonatomic, readwrite) BOOL useStoredData;
```

Notice that the `@property` declaration is a little different than what you've been using so far. Up to now, all your properties have been declared `(nonatomic, retain)`. What's this `readwrite` stuff? Because `useStoredData` is not an object (it's a Boolean value), `retain` is not applicable.

Add the `@synthesize` statements to the `iPadTravel411AppDelegate.m` file in order to tell the compiler to create the accessors for you.

```
@synthesize useStoredData;
```

Just standard stuff here.

With your housekeeping done, it's time to add the necessary code. But first, I want to explain something about defaults in a world of multitasking.

As I explain in Chapter 8, when the user is (temporarily) done with your application, the application does not terminate. Instead, it goes into the background and becomes inactive. But while your application is sitting there in the background, life goes on, and the user may have done something that impacts your application, like change the settings.

Fortunately, iOS 4 provides a way for you to be informed of what has happened while your application dreams of electric sheep. While your app is suspended, the user could be doing all sorts of things — like getting on a plane to London and changing his or her preference to Use Stored Data rather than the Internet. Although I'm going to concentrate on changes in user preferences, here's a list of some of the other things that could potentially impact your app:

- An accessory is connected or disconnected.
- The device orientation changes.
- There is a significant time change.
- The battery level or battery state changes.
- The proximity state changes.
- The status of protected files changes.
- An external display is connected or disconnected.
- The screen mode of a display changes.
- Preferences that your application exposes through the Settings application are changed.
- The current language or locale settings change.

That's a lot to keep track of, but iOS 4 is very helpful in keeping things straight for you. Instead of saving all those events and pummeling your application senseless with everything the user has done for the last six weeks, it coalesces events and delivers a single event (of each relevant type) that nets out all the changes since your app was suspended.

The way you find out whether anything has changed is through the Notification system. *Notification* is a system that allows objects within an application to learn about changes that occur elsewhere in that application. Usually, objects get information by messages that come to them. But that means the object that sends the message must know what objects it needs to update whenever it does something that those objects care about. And face it, the object has no clue about your app.

That's where notification comes in. Notification is a broadcast model where you can register your objects to be notified of a particular event. (You can even post a notification, although I'm not going to get into that here.) Notifications are managed by a single object, `NSNotificationCenter`, which is accessed by using the class method `defaultCenter`.

These net changes are sent to you in the `NSUserDefaultsDidChangeNotification` notification. For example, in iPadTravel411, the user may have turned on the Use Stored Data option. If you didn't respond to that when your application became active again, you could potentially rack up a rather large roaming bill for the user.

If you do receive the `NSUserDefaultsDidChangeNotification` notification, the appropriate response would be to reload the settings data and have your app behave appropriately.

To get that notification, though, you need to register for the `NSUserDefaultsDidChangeNotification` in the `application:didFinishLaunchingWithOptions:` method. Right after that is where you want to read in the current user preferences and set your new instance variables accordingly. Listing 19-8 shows you how it's done. Add it to `RootViewController.m` in `viewDidLoad` right after the code you added earlier to save the state.

Listing 19-8: viewDidLoad before tableView:didSelectRowAtIndexPath:

```
- (void)viewDidLoad {
…
[[NSNotificationCenter defaultCenter]
  addObserver:self selector:@selector(setDefaults:)
     name:NSUserDefaultsDidChangeNotification object:nil];

if (![[NSUserDefaults standardUserDefaults]
        objectForKey:kUseStoredDataPreference])
                          kAppDelegate.useStoredData = NO;
else
    kAppDelegate.useStoredData =
      [[NSUserDefaults standardUserDefaults]
                    boolForKey:kUseStoredDataPreference];
[self tableView:((UITableView *) self.tableView) didSelect
           RowAtIndexPath:indexPath];
```

Here's what you want all that code to do for you:

1. **Register for the `NSUserDefaultsDidChangeNotification` and inform the Notification Center to send you the `setDefaults:` message in the event the user changes a preference.**

   ```
   [[NSNotificationCenter defaultCenter] addObserver:self
          selector:@selector(setDefaults:)
   name:NSUserDefaultsDidChangeNotification object:nil];
   ```

2. **Check to see whether the settings have been moved into `NSUserDefaults`.**

   ```
   if (![[NSUserDefaults standardUserDefaults]
               objectForKey:kUseStoredDataPreference]){
   ```

   The Settings application moves the user's preferences from Settings into `NSUserDefaults` only *after* the user visits the setting for the first time. (I want to point out that if the user visits the setting and doesn't change anything, you won't get a change notification either.)

3. **If the settings haven't been moved into `NSUserDefaults` yet, use the defaults of `NO` and `YES` (which corresponds to the default you used for the initial preference value).**

   ```
   useStoredData = NO;
   ```

4. **If the settings *have* been moved, you assign the correct value.**
5. **In either case, you then simply send the message to display the view as you did before.**

   ```
   [self tableView:((UITableView *) self.tableView)
                      didSelectRowAtIndexPath:indexPath];
   ```

You also have to add the following to `Constants.h`:

```
#define kUseStoredDataPreference
                                @"useStoredDataPreference"
```

When you registered for the `NSUserDefaultsDidChangeNotification`, you passed an argument — `selector:@selector(setDefaults:)` — which informed the Notification Center what message to send (and what object to send it to) when there was a change to the defaults. That being the case, it's time to implement `setDefaults:`. Add the `setDefaults:` method in Listing 19-9 to `RootViewControlller.m`.

Listing 19-9: setDefaults:

```
- (void)setDefaults:(NSNotification *)notification {

   BOOL newDefault = [[NSUserDefaults standardUserDefaults]
                     boolForKey:kUseStoredDataPreference];
   if (newDefault != kAppDelegate.useStoredData) {
    kAppDelegate.useStoredData = newDefault;
    if ((kAppDelegate.useStoredData) &&
         (([[kAppDelegate.lastView objectAtIndex:1]
                                         intValue] == 1) ||
        ([[kAppDelegate.lastView objectAtIndex:1]
                                        intValue] == 3))) {
      NSIndexPath *indexPath =
              [NSIndexPath indexPathForRow:2 inSection:0];
     [self displayOfflineAlert:
          [[menuList objectAtIndex:
          [[kAppDelegate.lastView  objectAtIndex:1]
                                                 intValue]]
                               objectForKey:kSelectKey]];
     [self tableView:((UITableView *) self.tableView)
                       didSelectRowAtIndexPath:indexPath];

   }
   else {
     NSIndexPath *indexPath = [NSIndexPath indexPathForRow:
       [[kAppDelegate.lastView objectAtIndex:1]intValue]
       inSection:[[kAppDelegate.lastView objectAtIndex:0]
                                                intValue]];
     [self tableView:((UITableView *) self.tableView)
                       didSelectRowAtIndexPath:indexPath];
   }
  }
}
```

First you need to check to see whether the user defaults have really changed — so you read in the defaults from the file. Even though you were notified, it may have been for other reasons besides the user changing defaults in the Settings application. (The other reasons you may get the notification are beyond the scope of this book and are usually changes made by the OS in the defaults domain.)

```
BOOL newDefault = [[NSUserDefaults standardUserDefaults]
                     boolForKey:kUseStoredDataPreference];
  if (newDefault != kAppDelegate.useStoredData) {
```

If it's a change to your app's settings and it's to stored data mode, you need to see what the last view was. If it's a Map or Weather view, you need to inform the user that those views are no longer available because you're now offline. You also need to replace either of those views with one that does not require Internet access.

This is especially important in the Map view since the user panning the map may initiate an attempt at Internet access. It's also important in the Weather view, or any other Web view, because you don't want the user to update the page or touch a link (although there are other ways to keep that from happening in a Web view that you implement in the "Airport and City in Stored Data Mode" section, later in this chapter).

```
if ((kAppDelegate.useStoredData) &&
        (([[kAppDelegate.lastView objectAtIndex:1]
                                        intValue] == 1) ||
       ([[kAppDelegate.lastView objectAtIndex:1]
                                       intValue] == 3))) {
```

If the current view is the Map (1) or Weather (3) (1 and 3 are their positions in the `menuList` array — you might want to use some constants here so as not to confuse yourself), you send an alert (see Listing 19-10) and replace the last Master view selection (the current one in the Detail view) with one of your choice — any one, at least, that does not require Internet access. Here I'm having you use Currency, although you should really create a separate view controller that displays information about the fact that the user is now offline and that the application's functionality is now limited to whatever can be done offline.

```
NSIndexPath *indexPath =
              [NSIndexPath indexPathForRow:2 inSection:0];
 [self displayOfflineAlert:[[menuList objectAtIndex:
 [[kAppDelegate.lastView  objectAtIndex:1 intValue]]
                            objectForKey:kSelectKey]];
 [self tableView:((UITableView *) self.tableView)
                     didSelectRowAtIndexPath:indexPath];
```

If the preference has changed to the Online line (the Use Stored Data option has been switched off), you just continue processing normally with the current selection.

```
NSIndexPath *indexPath = [NSIndexPath indexPathForRow:
 [[kAppDelegate.lastView objectAtIndex:1]intValue]
 inSection: [[kAppDelegate.lastView objectAtIndex:0]
                                          intValue]];
 [self tableView:((UITableView *) self.tableView)
                     didSelectRowAtIndexPath:indexPath];
```

The next place you need to use the stored data preference is when the user makes a selection in the Master view. Some selections simply don't work when you are not online — maps and weather.

If the device isn't online, you can't deliver the quality of the information a user needs. (Saved current weather conditions is an oxymoron.) For other selections — Map, for example — a network connection is required. (Right now, no caching is available.) In either case, you send an alert to the user (see Listing 19-10) informing him or her that the selection is unavailable. (You also need to add the declaration to `RootViewController.h`.)

In `tableview:didSelectRowAtIndexPath:`, you need to check to see whether the app is in stored data mode. If the user has selected Weather or Map, you present an alert (the same one as the one you use in `setDefaults:`). You also change the selection to Currency, which is the only one you know needs no online access. (See the discussion above about creating a special view controller for those situations.)

As you will see in the next section, the other selections can use cached data, but only after the first time the views are displayed so the data can be cached. You should actually keep track of what has been cached, and its date, and inform the user if there is no cached data or how old the data is. Another thing for you to put on your plate.

Adding the code in bold in Listing 19-10 to `RootViewController.m` will send an alert when the user tries to select either Map or Weather when in stored data mode.

Listing 19-10: tableview:didSelectRowAtIndexPath:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

  [tableView deselectRowAtIndexPath:indexPath
                                                animated:YES];
  if (indexPath.section < 1) {
    [kAppDelegate.lastView replaceObjectAtIndex:0
           withObject:[NSNumber
           numberWithInteger:indexPath.section]];
     [kAppDelegate.lastView replaceObjectAtIndex:1
           withObject:[NSNumber
           numberWithInteger:indexPath.row]];
  }


  DetailViewController *targetController = nil;
  int menuOffset =
             [self menuOffsetForRowAtIndexPath:indexPath];
if ((kAppDelegate.useStoredData) &&
                  (menuOffset == 1 || menuOffset == 3)) {
  [self displayOfflineAlert:[[menuList
           objectAtIndex:menuOffset]
           objectForKey:kSelectKey]];
    menuOffset = 2;
}
  switch (menuOffset) {
…
```

Here's what happens when a user tries to select Map or Weather: First, you see whether you are in stored data mode.

```
if ((kAppDelegate.useStoredData) &&
                   (menuOffset == 1 || menuOffset == 3)) {
```

If not, you continue processing normally. If the app is in stored data mode and the selection was Map or Weather, you display an alert and replace the selection with Currency by changing the `menuOffset` and then continue processing. I've chosen not to replace the last selection in case the user then changes his or her mind and goes back and turns off stored data mode. If that happens, the last view that was selected in the Master view will again be displayed.

To create and display the alert, add the code in Listing 19-11 to `RootViewController.m` and add the declaration to `RootViewController.h`.

Listing 19-11: Displaying an Alert

```
- (void) displayOfflineAlert:(NSString *) selection {

  UIAlertView *alert = [[UIAlertView alloc]
           initWithTitle:selection
           message:@"is not available offline"
           delegate:self cancelButtonTitle:@"Thanks"
           otherButtonTitles:nil];
  [alert show];
  [alert release];
}
```

This displays a standard System Alert view. You customize the alert with the name of the selection by passing in the cell text — the same text you display in `tableView:cellForRowAtIndexPath` — which you get from the menu list by using the menu offset into the menu list and using `kSelectKey`:

```
[self displayOfflineAlert:[[menuList
      objectAtIndex:menuOffset] objectForKey:kSelectKey]];
```

Airport and City in Stored Data Mode

Of course, even though the ability to switch to stored data mode may be silly in some situations (Weather), for others it might just make perfect sense to use previously stored (or cached) data. Now, you already implemented storing the data in the `Airport` model object when you implemented it with the `saveAirportData:withDataURL:` method back in Chapter 18. In the following sections, you get to finish the implementation so you can actually use that data when in stored data mode. (You'll add similar functionality to the `City` model object as well.)

Managing real time and cached data

For `Airport`, the place to both make the decision about using stored data and to access the data itself is in the `returnTransportation:` method of `Airport` that you implemented back in Chapter 18. Make the changes in bold in Listing 19-12 in `Airport.m`.

Listing 19-12: Airport Model Method Used by Destination

```
- (NSURL *)returnTransportation:(int)
                                    transportationType  {

  NSURL *url ;
  BOOL realtime = !kAppDelegate.useStoredData;
  if (realtime) {
    switch (transportationType) {
      case 0: {
        url = [NSURL URLWithString:
          @"http://nealgoldstein.com/ToFromiPad100.html"];
         [self saveAirportData:
                   @"ToFromiPad100.html" withDataURL:url];
        break;
      }
      case 1: {
        url = [NSURL URLWithString:
          @"http://nealgoldstein.com/ToFromiPad101.html"];
         [self saveAirportData:
                   @"ToFromiPad101.html" withDataURL:url];
        break;
      }
      case 2: {
        url = [NSURL URLWithString:
          @"http://nealgoldstein.com/ToFromiPad102.html"];
         [self saveAirportData:
                   @"ToFromiPad102,html" withDataURL:url];
        break;
      }
    }
  }
  else  {
    switch (transportationType) {
      case 0: {
        url = [self getAirportData:@"ToFromiPad100.html"];
        break;
      }
      case 1: {
        url =
              [self getAirportData:@"ToFromiPad101.html"];
        break;
      }
      case 2: {
        url = [self getAirportData:@"ToFromiPad102.html"];
```

```
          break;
        }
      }
    }
    return url;
  }
```

Here you access the `storedData` instance variable to determine whether to go out onto the Internet and use the data there (and subsequently cache it) or use the data you have cached.

```
BOOL realtime = !kAppDelegate.useStoredData;
  if (realtime) {
```

You already added `saveAirportData:withDataURL:` back in Chapter18, but to refresh your memory, this method constructs the `NSURL` object that the Web view uses to load the data. (The `NSURL` object is simply an object that includes the utilities necessary for downloading files or other resources from Web and FTP servers or accessing local files.)

The `saveAirportData:withDataURL:` method (see the following code) downloads and saves the file containing the latest data for whatever transportation (`Taxi`, for example) the user selected. It's what will be displayed in the current view, and it'll be used later if the user specifies stored data mode.

```
- (void)saveAirportData:(NSString *) fileName withDataURL:
                                        (NSURL *) url {

  NSData *dataLoaded = [NSData
                            dataWithContentsOfURL:url];
  if (dataLoaded == NULL)
                        NSLog(@"Data not found %@", url);
  NSArray *paths = NSSearchPathForDirectoriesInDomains
             (NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  NSString *filePath = [documentsDirectory
                 stringByAppendingPathComponent:fileName];
  [dataLoaded writeToFile:filePath atomically:YES];
}
```

If the app is currently in stored data mode, the method returns the stored data as opposed to loading the data for its URL from the Internet. It gets the data by calling the `getAirportData:` method, which reads the data that was stored in `saveAirportData:withDataURL:`.

```
url = [self getAirportData:@"ToFromiPad100.html"];
```

As you might expect, you also need to add the now-all-too-familiar `import` statements to `Airport.m`.

```
#import "Constants.h"
#import "iPadTravel411AppDelegate.h"
```

To get this Access to Cached Data business taken care of, get the `getAirportData:` method working for you by adding the code in Listing 19-13 to `Airport.m`.

Listing 19-13: Accessing Cached Data

```
- (NSURL *)getAirportData:(NSString *) fileName{

  NSArray *paths = NSSearchPathForDirectoriesInDomains
             (NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  NSString *filePath = [documentsDirectory
                  stringByAppendingPathComponent:fileName];
  NSURL *theNSURL= [NSURL fileURLWithPath:filePath];
  if (theNSURL == NULL) NSLog (@"Data not there");
  return  theNSURL;
}
```

`getAirportData:` — just like the `saveAirportData:withDataURL:` method — constructs a NSURL object that the Web view then uses to load the data. No surprises here — you find the path using the same procedure you've already used several times before and then construct the `NSURL` object by using that path.

REMEMBER

In Chapter 18, instead of declaring the `saveAirportData:withDataURL:` method in the interface, I had you place the declaration in a class extension to make it more private. You now have to add `getAirportData:` to the extension as well. In `Airport.m`, add the line in bold to the class extension you added to the beginning of the file:

```
@interface Airport  ()
- (NSURL *)getAirportData:(NSString *) fileName;
- (void)saveAirportData:(NSString *) fileName
                              withDataURL:(NSURL *) url;
@end
```

There ain't no Web cruising in stored data mode

Of course, if the app is in stored data mode, you don't want to allow the user to click on links — that'll only lead to user frustration. You control the ability to access a link in a Web view in the `webView:shouldStartLoadWithRequest:navigationType:` method. Add the code in bold in Listing 19-14 to that method in `AirportController.m`.

Listing 19-14: Inhibiting Link Selection in Stored Data Mode

```
- (BOOL) webView:(UIWebView *)webView
  shouldStartLoadWithRequest:(NSURLRequest *)request
  navigationType:(UIWebViewNavigationType)navigationType {

  if ((navigationType ==
           UIWebViewNavigationTypeLinkClicked) &&
             ([[NSUserDefaults standardUserDefaults]
                  boolForKey:kUseStoredDataPreference])) {

    if ((navigationType ==
           UIWebViewNavigationTypeLinkClicked) &&
             ([[NSUserDefaults standardUserDefaults]
                  boolForKey:kUseStoredDataPreference])) {
            UIAlertView *alert =
              [[UIAlertView alloc] initWithTitle:@""
               message:NSLocalizedString
                    (@"Link not available offline",
                                     @"stored data mode")
               delegate:self
               cancelButtonTitle:NSLocalizedString(
                                     @"Thanks", @"Thanks")
               otherButtonTitles:nil];
      [alert show];
      [alert release];
      return NO;
    }
  }
  else
  if (navigationType ==
           UIWebViewNavigationTypeLinkClicked) {
    if (!backButton) {
      backButton =
      [[UIBarButtonItem alloc] initWithTitle:@"Airport"
           style:UIBarButtonItemStyleBordered target:self
                               action:@selector(goBack:)];
      self.navigationItem.rightBarButtonItem = backButton;
      [backButton release];
      theToolbar.hidden = YES;
    }
  }
  return YES;
}
```

Here are the marching orders for the stuff you're adding in Listing 19-14:

1. **Check to see whether the user has touched an embedded link while the app was in stored data mode.**

```
if ((navigationType ==
        UIWebViewNavigationTypeLinkClicked) &&
   ([[NSUserDefaults standardUserDefaults]
            boolForKey:kUseStoredDataPreference])) {
```

2. **If the app is in stored data mode, alert the user to the fact that the link is unavailable and return `NO` from the method.**

   This informs the Web view not to load the link.

   ```
   UIAlertView *alert = [[UIAlertView alloc]
       initWithTitle:@""
       message:NSLocalizedString(@"Link not available
                           offline", @"stored data mode")
       delegate:self
       cancelButtonTitle:NSLocalizedString
           (@"Thanks", @"Thanks") otherButtonTitles:nil];
   [alert show];
   [alert release];
   return NO;
   ```

   You create an alert here with a message telling the user that the link is not available in stored data mode. The Cancel button's text will be `@"Thanks"`.

There are also a number of other places you have implemented this method (I know, yet another argument for a Web view superclass), which means you should also add the code in Listing 19-14 to

- `CityController.m`
- `CurrencyController.m`
- `WeatherController.m`

You also have to add the following `#import` statements to each controller:

```
#import "Constants.h"
#import "iPadTravel411AppDelegate.h"
```

You actually don't need to disable links in `CurrencyController`, because it doesn't have any links, but it is a good general way to manage links in Web views, and you'll use it when you display your own view with external Web links.

Come to think of it, you don't really need to disable links in `WeatherController` either, because you won't launch the `WeatherController` if you are in stored data mode. This is the alternative approach to the one I showed you in the "Reading Settings in the Application" section, earlier in this chapter. Instead of showing an alert when the user changed the preference to Use Stored Data and displaying Currency as you did, you could leave the view displayed and disable links instead.

Adding Stored Data Mode to City

When you initially created the `City` object's `cityHappenings` method back in Chapter 16, I didn't have you cache the data. To do that, and to use the cached data, add the code in bold in Listing 19-15 to `City.m` and delete the code in BUI.

Listing 19-15: Update cityHappenings

```
- (NSURL *) cityHappenings {

return [NSURL
     URLWithString:@"http://nealgoldstein.com/City.html"];

  NSURL *url = nil;
  BOOL realtime = !kAppDelegate.useStoredData;
  if (realtime) {
    url = [NSURL URLWithString:
                    @"http://nealgoldstein.com/City.html"];
    [self saveCityData:@"City.html" withDataURL:url];
  }
  else {
    url = [self getCityData:@"City.html"];
  }
  return url;
}
```

You also have to add the `saveCityData:withDataURL:` and `getCity-Data:` methods in Listing 19-16 to `City.m`.

Listing 19-16: The City Data Methods

```
- (void) saveCityData:(NSString *) fileName
                              withDataURL:(NSURL *) url {

  NSData *dataLoaded =
                     [NSData  dataWithContentsOfURL:url];
  if (dataLoaded == NULL)
                      NSLog(@"Data not found %@", url);
  NSArray *paths = NSSearchPathForDirectoriesInDomains
           (NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  NSString *filePath = [documentsDirectory
                stringByAppendingPathComponent:fileName];
  [dataLoaded writeToFile:filePath atomically:YES];
}
```

(continued)

Listing 19-16 ***(continued)***

```
-(NSURL *) getCityData:(NSString *) filename {

  NSArray *paths = NSSearchPathForDirectoriesInDomains
             (NSDocumentDirectory, NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  NSString *filePath = [documentsDirectory
                 stringByAppendingPathComponent:fileName];
  NSURL *theNSURL=
               [[NSURL fileURLWithPath:filePath] retain];
  if (theNSURL == NULL) NSLog (@"Data not there");
  return theNSURL;
}
```

Not much that needs explaining here — there's nothing here you haven't already done before.

You also need to add the class extension to `City.m` before the `@implementation` statement and import the necessary interface files:

```
#import "City.h"
#import "Constants.h"
#import "iPadTravel411AppDelegate.h"

@interface City  ()
- (NSURL *) getCityData:(NSString *) filename;
- (void) saveCityData:(NSString *) fileName withDataURL:
                                          (NSURL *) url;
@end

@implementation City
```

Finally

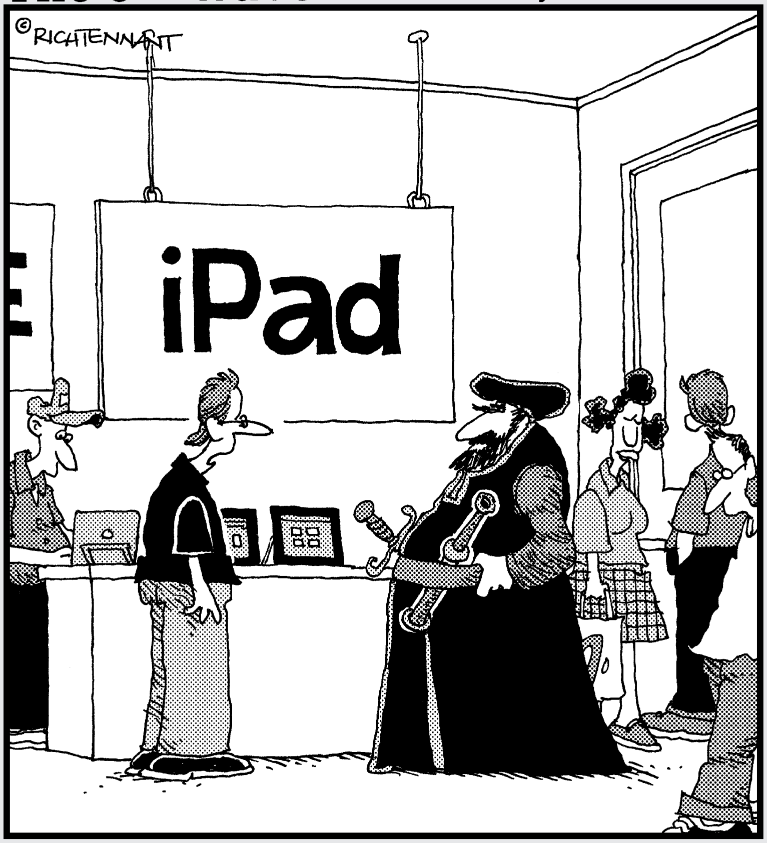
I know I've used *finally* in this book before, but this is the final *finally.*

You've developed a strong understanding of how to create really good iPad apps — now go out and do it!

And don't forget to keep in touch through my Web site `www.nealgoldstein.com` and show me what you have done.

Part VI
The Part of Tens

The 5th Wave By Rich Tennant



"In fact it does come with a compass."

In this part . . .

You've reached the last part, the part you've come to expect in every *For Dummies* book that neatly encapsulates just about all the interesting aspects of this book's topic. Like the compilers of other important lists — David Letterman's Top Ten, the FBI's Ten Most Wanted, the Seven Steps to Heaven, the 12 Gates to the City, the 12 Steps to Recovery, The 13 Question Method, and the Billboard Hot 100 — we take seriously this ritual of putting together the *For Dummies* Part of Tens.

- In Chapter 20, we offer ten important iPad app design tips that can help you create a more successful app. Included are tips on when your app should save data, how your app should handle starting and stopping, how to support all display orientations, and even how to submit a potent app icon.
- Chapter 21 presents ten techniques for attaining iPad developer enlightenment (or just ways to be happy). Included are tips on following memory management rules, planning ahead to extend your code, and creating code that's easy to understand.

Chapter 20

Ten Tips on iPad App Design

In This Chapter

- ▶ Figuring out what makes a great iPad application *icon*
- ▶ Discovering the features of the iPad that can inspire you
- ▶ Understanding Apple's expectations for iPad applications
- ▶ Making a plan for developing iPad software

When John Reed wrote *Ten Days that Shook the World* in 1919, he was writing about a different kind of revolution than the one Steve Jobs referred to in his announcement of the iPad. For this revolution, I put together ten tips that will shake up your thoughts about application design, especially if you're familiar with iPhone app design. There are important differences to know about, and these tips will help you make your iPad app more successful.

You can find more details about each and every one of these tips in the *iPad Human Interface Guidelines* inside the iOS Reference Library. Chapter 4 shows you how to register as a developer and gain access to this library and other resources in the Apple iPhone Dev Center.

Making an App Icon for the Masses

With over 300,000 apps in the App Store, it's a challenge to come up with an icon for your app that would make it stand out in the App Store and look unique and inviting to touch on an iPad display.

Don't even think about using an Apple image, such as an iPad (or iPhone or iPod) in your icon, or you'll most likely receive a polite, but firm, e-mail rejecting the application.

Chapter 9 shows you how to add your icon to your app, and Chapter 6 spells out the details of what form your icon should take — including the fact that you need to use the same (or very similar) graphic image for the small app icon on the iPad display (at 72 x 72 pixels) you'll use for the larger App Store icon (at 512 x 512 pixels). You also need to supply an approximately 48-x-48-pixel version of this icon for display in Spotlight search results and in the Settings application (if you provide settings).

As with iPhone application icons, iOS on the iPad automatically adds rounded corners, a reflective shine, and a drop shadow, so you shouldn't add those effects to your icon. Create an icon with 90-degree corners and without any shine or gloss (or alpha transparency) and save it in the `.png` format to submit to Apple.

Make sure you fill the entire 72-x-72-pixel area — if your image boundaries are smaller, or you use transparency to create see-through areas within it, your icon will appear to float on a black background with rounded corners. Although this may seem fine at first, remember that users can display custom pictures on their Home screens, and an icon with a visible black background looks bad.

Launching Your App Into View

I go into considerable detail in Chapter 8 about how an app starts up and displays a view. Although the chapter is long and takes a while to read, iOS on the iPad performs these functions instantaneously — so fast that the view appears instantly.

You should take advantage of the speed of the app launch to display an image that represents the heart of your app's functionality — such as one that resembles the most common view of the app's user interface — in the iPad's current orientation. (See "Supporting All Display Orientations," later in this chapter.) You may think you want to use an About window (with your brand image) or a splash screen, but that slows app startup, and your users will see it every time they start your app. Better to use a simple, stripped-down screen shot of your app's initial user interface or a similar image with only the constant, unvarying elements of the user interface. Avoid all text because you don't want to go through the nightmare of providing different images for different countries.

Your goal during startup should be to present your app's user interface as quickly as possible. Don't load large data structures that your app won't use right away. If your app requires time to load data from the network (or perform other tasks that take a noticeable amount of 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 app is loading the necessary data or otherwise doing something important.

Stopping Your App on a Dime

When the user presses the Home button to quit, or presses it twice quickly to switch to another app, your app comes to an immediate stop; but it shouldn't crumble as if it hit a brick wall. You need to provide a good stopping experience — or more to the point, a good restarting experience for the user who quits and then returns to your app.

If you save data in your app frequently, your app will stop more gracefully if you don't require the user to tap a Save button. (See "Saving Grace with Your App's Data," later in this chapter.) Your app needs to save user data while it's running, and as often as reasonable, because a stop or terminate notification — like Immigration or the Spanish Inquisition — can arrive at any time. And be sure to save the current state when stopping, at the finest level of detail possible, because users expect to return to their earlier context when they switch back to or restart your app. For example, if you use a Split view in your app, store the current selection in the master pane and be sure to display that selection again when the user returns to your app.

For example, in Chapters 10 and 11 you add code to the DeepThoughts app that saves the user's preference settings as they're set (the text for the flowing words and the speed for the animation). When the user restarts DeepThoughts, the app uses the user's settings for the falling words and speed.

Saving Grace with Your App's Data

Don't force your users to tap a Save button. iPad apps should take responsibility for saving the user's input not only periodically, but also when a user opens a different document or quits the app.

This design goal addresses the very essence of the iPad experience: that users should feel comfortable consuming information and have complete confidence that their work is always preserved (unless they explicitly delete the work or cancel). If your app lets users create and edit documents, design it so that users don't have to explicitly perform saves. If your app doesn't create content but lets users switch between viewing information and editing it (such as Contacts), your app can offer an Edit button that turns into a Save button when users tap it, and the app can include a Cancel button when that happens. By doing both, your app reminds the users that they're in edit mode and that they can either save the edits or cancel.

If your app uses popovers, you should always save information that users enter in a popover (unless they explicitly cancel) because users may dismiss the popover inadvertently.

Supporting All Display Orientations

As you probably know (or read in Chapter 2), when you rotate the iPad from a vertical view (portrait) to a horizontal view (landscape), the accelerometer detects the movement and changes the display accordingly. Motion detection happens so quickly that you can control a game with these movements.

This is important: iPad users expect apps to run well in the device orientation they're currently using. As much as possible, your app should enable users to interact with it from any side by providing a great experience in all orientations.

For example, the iPad app's launch image should be ready to launch in any of the four orientations — so you need to provide four unique launch images. Each launch image should be in the `.png` format and should match the size of the iPad display in either portrait orientation (1,024 x 768 pixels) or landscape orientation (768 x 1,024 pixels).

Flattening Information Levels

If you've developed for the iPhone, you want to rethink your app design goals for the iPad. One technique in particular is the one-level-per-screen structure of iPhone apps, which forces your information into a hierarchy of screens resembling an upside-down tree (with the first screen acting as the root). As you tap an option on a screen, you go deeper into the upside-down tree into more detailed or more specific screens.

Although this structure makes sense for the iPhone's smaller display that can hold only one screen at a time, for your iPad app you need to flatten this structure — spread the information out horizontally rather than in a vertically oriented tree structure — so that it doesn't force iPad users to visit many different screens of information to find what they want. They have one large display, so use it. Focus the app's Main view on the primary content and provide additional information or tools in an auxiliary view, such as a popover. (See "Popping Up All Over," later in this chapter.)

Your app needs to provide easy access to the functionality users need without requiring them to leave the context of the main task. For example, in Chapter 14 you take a Table view for the iPadTravel411 sample app, which is appropriate for an iPhone app, and implement it as one of the views of a *Split view* (two views on the display at once), which is more appropriate for an iPad app. Use a Split view to persistently display the top level of a hierarchy in the left pane and content that changes in the right pane, as shown in Figure 20-1. This flattens your information hierarchy by at least one level, because two levels are always onscreen at the same time.

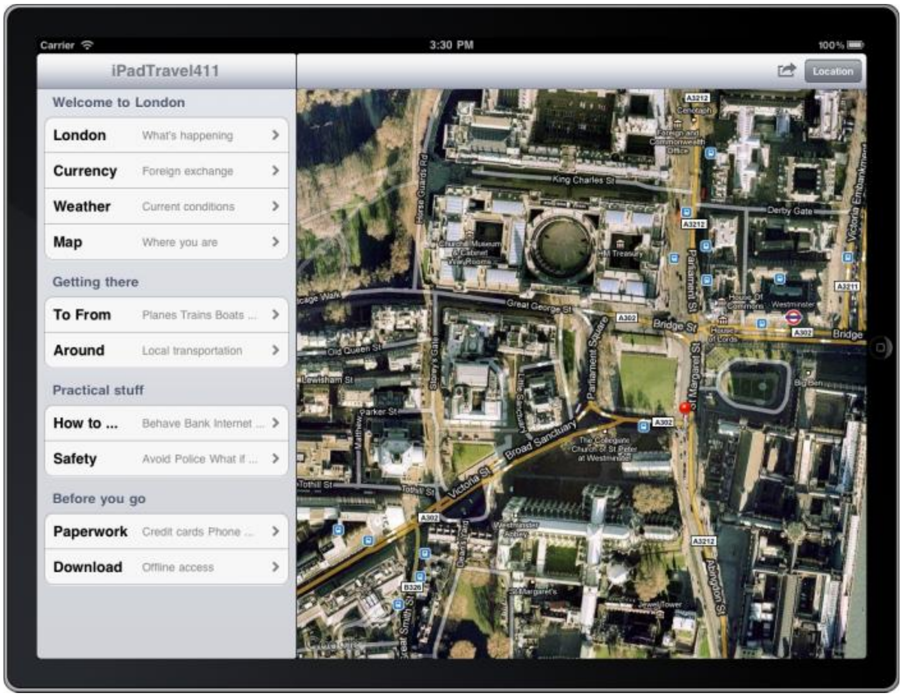
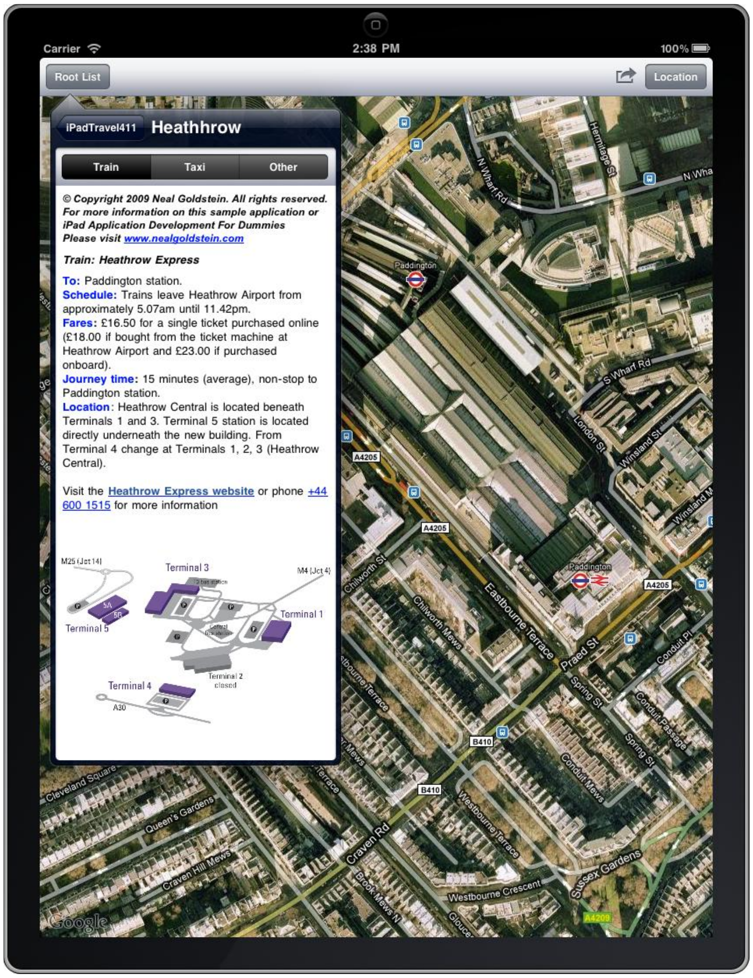

Carrier
3:30 PM
100%
iPadTravel411
Location
Welcome to London
London What's happening
Currency Foreign exchange
Weather Current conditions
Map Where you are
Getting there
To From Planes Trains Boats ...
Around Local transportation
Practical stuff
How to ... Behave Bank Internet ...
Safety Avoid Police What if ...
Before you go
Paperwork Credit cards Phone ...
Download Offline access

Figure 20-1: The split screen design for iPad-Travel411 in landscape orientation.

Popping Up All Over

A popover appears (that is, it pops up on top of the Main view) when a user taps a control or an area of the display. You can use a popover to enable actions or provide tools that affect objects in the Main view. A popover can display these actions and tools temporarily on top of the current view, which means people don't have to move on to another view to perform the actions or use the tools.

You can put almost anything in a popover, from tables, images, maps, text, or Web views to navigation bars, toolbars, and controls. A popover can be useful for displaying the contents of the landscape orientation's left pane when a Split view–based application is in portrait orientation. For example, in Chapter 14 you find out how to display the Master view that you see in landscape orientation in a Popover view in portrait orientation, as shown in Figure 20-2.



© Copyright 2009 Neal Goldstein. All rights reserved. For more information on this sample application or iPad Application Development For Dummies Please visit www.nealgoldstein.com

Train: Heathrow Express

To: Paddington station.
Schedule: Trains leave Heathrow Airport from approximately 5.07am until 11.42pm.
Fares: £16.50 for a single ticket purchased online (£18.00 if bought from the ticket machine at Heathrow Airport and £23.00 if purchased onboard).
Journey time: 15 minutes (average), non-stop to Paddington station.
Location: Heathrow Central is located beneath Terminals 1 and 3. Terminal 5 station is located directly underneath the new building. From Terminal 4 change at Terminals 1, 2, 3 (Heathrow Central).

Visit the Heathrow Express website or phone +44 600 1515 for more information

Figure 20-2: A popover shows the Master view for iPad-Travel411 in portrait orientation.

Minimizing Modality to Maximize Simplicity

Yo! Keep it simple! Keep those Modal views to a minimum. A Modal view is a child window, like a dialog in Mac OS X, that appears on top of the parent window (the main view) and requires the user to interact with it before returning to the parent window. I know, I showed you how to add a Modal view in DeepThoughts in Chapter 11, but that one is for one purpose only — to change the preference settings — and users can choose whether to tap the display (or the Info button) to bring it up.

You don't want to annoy users with modal dialogs that force them to perform a task or supply a response. iPad apps should react to taps in flexible, nonlinear ways. Remember that modality prevents freedom of movement through your app by interrupting the user's workflow and forcing the user to choose a particular path. In general, you should use modality only when it's critical to get the user's attention, or a task must be completed (or explicitly abandoned) to avoid leaving the user's data in an ambiguous state, such as unsaved. Got that?

And if you must use a Modal view, keep the modal tasks fairly short and narrowly focused. You don't want your users to experience a Modal view as a mini application within your application.

Turning the Map into the Territory

As you discover in Chapter 15, working with maps is one of the most enjoyable things you can do on the iPad because Apple makes it so easy — you can display a map that supports the standard panning and zooming gestures by simply creating a view controller and a nib file. You can also center the map on a given coordinate, specify the size of the area you want to display, and annotate the map with custom information.

You can also specify the map type — regular, satellite, or hybrid — by changing a single property. For many apps, one type may work better than another, but consider using hybrid so that your app displays streets and highways superimposed over the satellite image. What you should do is provide a control for the user to make the choice between all three types.

Making Smaller Transitions (Don't Flip the View)

The iPad's display is inherently immersive; many things can be going on inside a single view. It's far better to change or update only the areas of the view that need it, rather than swapping in a whole new full-page view when some embedded information changes (as you would probably do in an iPhone app).

Don't flip the entire view if something needs to change. Do transitions with smaller views and objects. Associate any visual transitions with the content that's changing. Use a Split view so that only one part of the view changes (see "Flattening Information Levels," earlier in this chapter), or use a popover for information that changes, to lessen the need for a full-screen transition (as described in "Popping Up All Over," earlier in this chapter). As a result, your app will appear to be more visually stable, and users will feel confident that they know where they are in a given task.

Chapter 21

Ten Ways to Be a Happy Developer

In This Chapter

- Finding out how not to paint yourself into a corner
- Avoiding "There's no way to get there from here."

There are a lot of things you know you're supposed to do, but you don't do them because you think they'll never catch up with you. (After all, not flossing won't cause you problems until your teeth fall out years from now, right?)

But in iPad (and iPhone) application development, those things catch up with you early and often, so I want to tell you about the things I've learned to pay attention to from the very start in app development, as well as a few tips and tricks that lead to happy and healthy users.

It's Never Too Early to Start Speaking a Foreign Language

With the world growing even flatter, and with the iPad available in many countries, the potential market for your app is considerably larger than just the people who happen to speak English. *Localizing* an application — getting your app to speak the lingo of its user, whether that be Portuguese or Polish — isn't difficult, just tedious. Some of it you can get away with doing late in the project, but when it comes to the strings you use in your application, you'd better build them right — and build them in from the start. The painless way: Use the `NSLocalizedString` macro (refer to Chapter 14) from the very start, and you'll still be a happy camper at the end.

Remember Memory

The iPad operating system (iOS) doesn't store changeable memory (such as object data) on the disk as a way to free up space and then read it back in later when needed. It also doesn't have garbage collection, which means there is a real potential for damage from memory leaks unless you tidy up after your app. Review and follow the memory rules in Chapter 8 — in particular, these:

- Memory management is really creating *pairs* of messages. Balance every `alloc`, `new`, and `retain` with a `release`.
- When you assign an instance variable using an accessor with a property attribute of `retain`, you now own the object. When you're done with it, release it in a `dealloc` method.

Constantly Use Constants

In the iPadTravel411 application, I put all my constants in one file. When I develop my own applications, I do the same. The why of it is simple: As I change things during the development process, having *one* place to find my constants makes life much easier.

Don't Fall Off the Cutting Edge

The iPad is cutting edge enough that there are still plenty of opportunities to expand its capabilities — and many of them are (relatively) easy to implement. You're also working with a very mature framework. So if you think something you want your app to do is going to be *really* difficult, check the framework; somewhere in there you may find an easy way to do what you have in mind. If there isn't a ready-made fix, consider the iPad's limited resources — and at least question whether that nifty task you had in mind is something your app should be doing at all. Then again, if you really *need* to track orbital debris with an iPad app, go for it — someone needs to lead the way. Why shouldn't it be you?

Start by Initializing the Right Way

A lot of my really messy code that I found myself re-doing ended up that way because I didn't think through initialization. (For example, adding on initialization-like methods after objects are already initialized is a little late in the game, and so on.)

Keep the Order Straight

One of the things that can really foul up your day as a developer is the order in which methods in objects are called. If you expect an object to be there (and it isn't) or to have been initialized (and it wasn't), you may be in the wrong method. Type up a copy of Table 21-1 in a file and/or make a photocopy of it and tack it up where you can easily find it. It shows you in crisp, tabular form the order in which objects are called — from soup (view controller) to nuts (delegate).

Table 21-1 The Natural Order of Things

Object	*Method*
View Controller	`awakeFromNib`
Application Delegate	`application:didFinish LaunchingWithOptions::`
View Controller	`viewDidLoad`
View Controller	`viewWillAppear:`
View Controller	`viewWillDisappear:`
Delegate	`applicationWillTerminate:`

What trips up many developers is that the `awakeFromNib` message for the initial view controller (the one you see when the application starts) is sent *before* the `applicationDidFinishLaunchingWithOptions::` message. If you have a problem with that, do what you need to do in `ViewDidLoad`.

Avoid Mistakes in Error Handling

A lot of opportunities for errors are out there; use common sense in figuring out which ones you should spend work time on.

There are, however, some potential pitfalls you do have to pay attention to, such as these two big ones:

- Your app goes out to load something off the Internet, and (for a variety of reasons) the item isn't there or the app can't get to it.
- An object can't initialize itself (for a similar range of perverse reasons).

When (not *if*) those things happen, your code and your user interface must be able to deal with the error.

Remember the User

I've been singing this song since Chapter 2: Keep your app simple and easy to use. Don't build long pages that take a lot of scrolling to get through, and don't create really deep hierarchies. Focus on what the user wants to accomplish and be mindful of the device limitations, especially battery life. And don't forget international roaming charges.

In other words, try to follow the Apple's iPad Human Interface Guidelines, found with all the other documentation in the iOS Dev Center at `http://developer.apple.com/ios` under the iOS Reference Library section — Required Reading. Don't even *think* about bending those rules until you really, *really* understand them.

Keep in Mind that the Software Isn't Finished Until the Last User Is Dead

If there's one thing I can guarantee about app development, it's that Nobody Gets It Right the First Time. The design for all of my apps evolved over time, as I learned the capabilities and intricacies of the platform and the impact of my design changes. Object-orientation makes extending your application (not to mention fixing bugs) easier, so pay attention to the principles.

Keep It Fun

When I started programming the iPhone and then the iPad, it was the most fun I'd had in years. Keep things in perspective: Except for a few tedious tasks (such as provisioning and getting your application into the Apple Store), lo, I prophesy: Developing iPad apps will be fun for you, too. So don't take it *too* seriously.

TIP

Especially remember the *fun* part at 4 a.m., when you've spent the last five hours looking for a bug. Here's a handy way to do that: Check out what Douglas Adams says about the *Hitchhiker's Guide to the Galaxy* entry for surviving in space. The guide "says that if you hold a lungful of air you can survive in a total vacuum of space for about thirty seconds. However, it does go on to say that what with space being the mind-boggling size it is the chances of getting picked up by another ship within those thirty seconds are two to the power of two hundred and seventy-six thousand, seven hundred and nine to one against." Now, get back to work!

Index

Macintosh/Programming

Turn your incredible ideas into impressive iPad apps with help from this informative guide!

Ready to join the iPad developer ranks? Now you can — even if you've never developed an app for a mobile device. If you know just a bit about object-oriented programming, Neal and Tony will help you do the rest, walking you through the iPad app development process in language you can understand. All you'll need is an Intel-based Mac, your iPad, your imagination, and this book to get started today!

- ***Plan your app* — *understand what makes a great iPad app and how to create a terrific user experience***
- ***Handle the administrative stuff* — *download the SDK, register as a developer, and follow all the rules for submitting your app to the App Store***
- ***Explore app anatomy* — *get acquainted with the frameworks that structure an app and the app lifecycle***
- ***Build on that framework* — *put together a sample app using Interface Builder and get comfortable with the tools***
- ***Get serious* — *learn to build an app with major functionality and take full advantage of the iPad's capabilities***

Open the book and find:

- **What makes a killer iPad app**
- **Secrets for creating a super user experience**
- **How to market and spread the word about your app**
- **Rules you must follow to avoid App Store rejection**
- **Tips for working with the SDK**
- **What design patterns are and how to use them**
- **Advice on testing and debugging your app**
- **How to maximize the iPad's unique features**

Visit the companion Web site at
www.dummies.com/go/ipadappdevfd2e
to download all source code used in the book

Neal Goldstein is a master at making cutting-edge technology practical. He was an early pioneer of object-oriented programming and enjoys rock-star status among mobile developers. **Tony Bove** has written more than two dozen books, including all editions of *iPod touch For Dummies* and *iPod & iTunes For Dummies*.

For Dummies®
A Branded Imprint of
WILEY

$29.99 US / $35.99 CN / £21.99 UK

ISBN 978-0-470-92050-3

52999

9 780470 920503