
7SUMMITS

Get started building your very own
iPhone and iPad apps

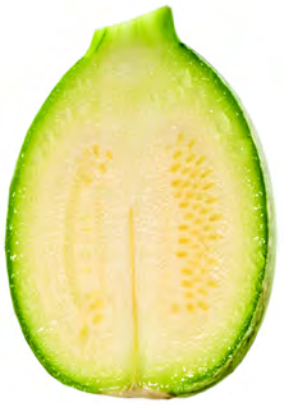
iPhone and iPad Apps for Absolute Beginners

Dr. Rory Lewis

Foreword by Ben Easton

Apress®

iPhone and iPad Apps for Absolute Beginners



Dr. Rory Lewis

Apress®

iPhone and iPad Apps for Absolute Beginners

ISBN-13 (pbk): 978-1-4302-2700-7

ISBN-13 (electronic): 978-1-4302-2701-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

President and Publisher: Paul Manning
Lead Editor: Ben Renow-Clark
Technical Reviewer: Kristian Besley
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editors: Fran Parnell and Debra Kelly
Copy Editor: Jim Compton
Compositor: MacPS, LLC
Indexer: Potomac Indexing, LLC
Artist: April Milne
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/info/bulksales`.

The source code for this book is available to readers at `www.apress.com`.

To Adrian and Eunice Lewis: Love you, Granny, and I miss you so much, Pops.

—Dr. Rory Lewis

Contents at a Glance

Contents

Foreword: About the Author

"Rory and I met in L.A. in 1983. He reminds me of one of my favorite film characters, Buckaroo Banzai—always going in six directions at once. If you stop him and ask what he's doing, he'll answer comprehensively and with amazing detail. Disciplined, colorful, and friendly, he has the uncanny ability to explain the highly abstract in simple, organic terms. He always accomplishes what he sets out to do, and he'll help you do the same.

Why you'll relate to Dr. Lewis

While attending Syracuse University as a computer-engineering student, Rory scrambled to pass his classes and make money to support his wife and two young daughters. In 1990, he landed a choice on-campus job as a proctor in the computer labs in the LC Smith College of Engineering. Even though he was struggling with subjects in the Electrical Engineering program, he was always there at the Help Desk. It was a daunting experience for Rory because his job was only to help his fellow students with computer lab *equipment* questions, but he invariably found his classmates asking deeper and harder questions: *"Dude, did you understand the calculus assignment? Can you help me?!"*

These students assumed that, because Rory was the proctor, he knew the answers. Afraid and full of self-doubt, he sought a way to help them without revealing his inadequacies. Rory learned to start with: *"Let's go back to the basics. Remember that last week the professor presented us with an equation…?"* By going back to the fundamentals, restating and rebranding them, Rory began to develop a technique that would, more often than not, lead to working solutions. By the time his senior year rolled around, there was often a line of students waiting at the Help Desk on the nights Rory worked.

Fast-Forward 17 Years

Picture a long-haired, wacky professor walking through the campus of the University of Colorado at Colorado Springs, dressed in a stunning contrast of old-school and drop-out. As he walks into the Engineering Building, he is greeted by students and faculty who smile and say hearty hellos, all the while probably shaking their heads at his tweed jacket, Grateful Dead t-shirt, khaki pants, and flip flops. As he walks down the hall of the Computer Science Department, there's a line of students standing outside his office. Reminiscent of the line of students that waited for him at the Help Desk in those early years as a proctor in the computer lab, they turn and greet him, *"Good morning, Dr. Lewis!"* Many of these students at UC-Colorado Springs are not even in his class, but they know that Dr. Lewis will see them and help them anyway.

Past—Present—Future

Dr. Lewis holds three academic degrees. He earned a Bachelor of Science in Computer Engineering from Syracuse University. Syracuse's LC Smith College of Engineering is one of the country's top schools. It is there that Intel, AMD, and Microsoft send their top employees to study for their PhDs.

Upon completing his BS (with emphasis on the mathematics of electronic circuitry in microprocessors), he went across the quad to the Syracuse University School of Law. During his first summer at law school, Fulbright & Jaworski, the nation's most prolific law firm, recruited Rory to work in its Austin office, where some of the attorneys specialize in high-tech intellectual-property patent litigation. As part of his clerking experience, Lewis worked on the infamous *AMD v. Intel* case; he helped assess the algorithms of the mathematics of microprocessor electrical circuitry for the senior partners.

During his second summer in law school, Skjerven, Morrill, MacPherson, Franklin, & Friel—the other firm sharing the work on the *AMD v. Intel* case—recruited Rory to work with them at their Silicon Valley branches (San Jose and San Francisco). After immersing himself in law for several years and receiving his JD at Syracuse, Lewis realized his passion was for the *mathematics* of computers, not the legal ramifications of hardware and software. He preferred a learning and creative environment rather than the fighting and arguing intrinsic in law.

After three years away from academia, Rory Lewis moved south to pursue his PhD in Computer Science at the University of North Carolina at Charlotte. There, he studied under Dr. Zbigniew W. Ras, known worldwide for his innovations in data mining algorithms and methods, distributed data mining, ontologies, and multimedia databases. While studying for his PhD, Lewis taught computer science courses to computer engineering undergraduates, as well as e-commerce and programming courses to MBA students.

Upon receiving his PhD in Computer Science, Rory accepted a tenure-track position in Computer Science at the University of Colorado at Colorado Springs, where his research is in the computational mathematics of neurosciences. Most recently, he co-wrote a grant proposal on the mathematical analysis of the genesis of epilepsy with respect to the hypothalamus. However, with the advent of Apple's revolutionary iPhone and its uniquely flexible platform—*and market*—for mini-applications, games, and personal computing tools, he grew excited and began experimenting and programming for his own pleasure. Once his own fluency was established, Lewis figured he could teach a class on iPhone apps that would include *non*-engineers. With his insider knowledge as an iPhone beta tester, he began to integrate the parameters of the proposed iPad platform into his lesson plans—even before the official release in April 2010.

The class was a resounding success and the feedback was overwhelmingly positive, from students and colleagues alike. When approached about the prospect of converting his course into a book to be published by Apress, Dr. Lewis jumped at the opportunity. He happily accepted an offer to convert his course outlines, class notes, and videos into the book you are now holding in your hands.

Why Write This Book?

The reasons Dr. Lewis wrote this book are the same reasons he originally decided to create a class for both engineering and non-engineering majors: the challenge and the fun! According to Lewis, the iPhone and iPad are "*...some of the coolest, most powerful, and most technologically advanced tools ever made—period!*"

He is fascinated by the fact that, just under the appealing touch screen of high-resolution images and fun little icons, the iPhone and iPad are programmed in *Objective-C,* an incredibly difficult and advanced language. More and more, Lewis was approached by students and colleagues who wanted to program apps for the iPhone and would ask his opinion on their ideas. It seemed that,

with every new update of the iPhone, not to mention the advent of the expanded interface of the iPad, the floodgates of interest in programming apps were thrown wider and wider. Wonderful and innovative ideas just needed the proper channel to flow into the appropriate format and then out to the world.

Generally speaking, however, the people who write books about Objective-C write for people who know Java, C#, or C++ at an advanced level. So, because there seemed to be no help for the average person who, nevertheless, has a great idea for an iPhone/iPad app, Dr. Lewis decided to launch such a class. He realized it would be wise to use his own notes for the first half of the course, and then to explore the best existing resources he could find.

As he forged ahead with this plan, Lewis was most impressed with *Beginning iPhone 3 Development: Exploring the iPhone SDK*. This best-selling instructional book from Apress was written by Dave Mark and Jeff Lamarche. Lewis concluded that their book would provide an excellent, high-level target for his lessons...a "stepping stones" approach to comprehensive and fluent programming for all Apple's multi-touch devices.

After Dr. Lewis's course had been successfully presented, and during a subsequent conversation with a representative from Apress, Lewis happened to mention that he'd only started using that book about half-way through the semester, as he had to bring his non-engineering students up to speed first. The editor suggested converting his notes and outlines into a primer—an introductory book tuned to the less-technical programming crowd. At that point, it was only a matter of time and details—like organizing and revising Dr. Lewis's popular instructional videos to make them available to other non-engineers excited to program their own iPhone and/or iPad apps.

So, that's the story of how a wacky professor came to write this book. We hope you are inspired to take this home and begin. *Arm yourself with this knowledge and begin now to change your life!*

Ben Easton
Author, Teacher, Editor

About the Contributing Authors

Ben Easton is a graduate of Washington & Lee University and has a B.A. in Philosophy. His eclectic background includes music, banking, sailing, hang gliding, and retail. Most of his work has involved education in one form or another. Ben taught school for 17 years, mostly middle-school mathematics. More recently, his experience as a software trainer and implementer reawakened his long-time affinity for technical subjects. As a freelance writer, he has written several science fiction stories and screenplays, as well as feature articles for magazines and newsletters. Ben resides in Austin, Texas, and is currently working on his first novel.

Kyle Roucis is a student at the University of Colorado at Colorado Springs pursuing degrees in Computer Science and Game Design and Development. Kyle was Dr. Lewis' teaching assistant for CS 201, which was the class that tested all the apps and tutorial methodologies presented in this book. Kyle graded many students' attempts to write code from the lessons in this book and contributed wonderful suggestions as to how Dr. Lewis should change the way he presented certain topics. Kyle has been developing applications for the iPhone and iPod Touch since the SDK was first released in June of 2007. Most of his work has been iPhone contracting work as well as game and entertainment app development. Kyle lives in Colorado Springs and hopes to create his own game studio with an emphasis on iPhone, iPad and Mac game programming.

About the Technical Reviewer

Kristian Besley is a web developer and programmer. He currently works in education and specializes in games, interactivity, and dynamic content using mainly Flash, PHP, and .NET. He also lectures on interactive media.

Kristian has worked as a freelance producer for numerous clients including the BBC, JISC, Welsh Assembly Government, Pearson Education, and BBC Cymru.

He has written a number of books for Friends of ED, such as the *Foundation Flash* series, *Flash MX Video*, *Flash ActionScript for Flash 8*, and *Learn Programming with Flash MX*. He was also a proud contributor to the amazing *Flash Math Creativity* books and has written for *Computer Arts* magazine.

Kristian currently resides with his family in Swansea, Wales, and is a proud fluent Welsh speaker.

Acknowledgments

When I arrived in America in 1981 at the age of 20, I had no experience, money, or the knowledge to even use an American payphone. Since then it's been a wonderful road leading to this book and my life as an Assistant Professor at two University of Colorado campuses. I am such a lucky man to have met so many wonderful people.

First, to my wife, Kera, who moved mountains to help with graphics, meals, dictations, keeping me working, and sustaining a nominal level of sanity in our house. Thank you, Kera.

To my mother, Adeline, who was always there to encourage me, even in the darkest of times when I almost dropped out of Electrical Engineering. To my sister, Vivi, who keeps me grounded, and my late brother Murray, a constant reminder of how precious life is. To Keith and Nettie Lewis who helped me figure out those American payphones. To Ben Easton, Brian Bucci, and Dennis Donahue, all of whom invited me into their families when I had nobody.

A special thanks to Dr. Zbigniew Ras, my PhD advisor, who became like a father to me, and to Dr. Terry Boult, my mentor and partner in the Bachelor of Innovation program at UCCS.

Last but not least, to Clay Andres at Apress—he walked me through this process and risked his reputation by suggesting to a bunch of really intelligent people that I could author such a book as this.

Many thanks to you all.

Preface

What *This* Book Will Do For *You*

Let me get this straight: you want to learn how to program for the iPhone or the iPad, and you consider yourself to be pretty intelligent—but whenever you read computer code or highly technical instructions, your brain seems to shut down. Do your eyes glaze over when reading gnarly instructions? Does a little voice in your head chide you, *"How about that! Your brain shut down six lines ago, but you're still scanning the page—pretending you're not as dense as you feel. Great!"*

See if you can relate to this…you're having an issue with something pretty technical and you decide to Google it and troubleshoot the problem. You open the top hit—and somebody else has asked the exact same question! You become excited as the page loads, but, alas, it's only a bulletin board (a chat site for all those geeks who yap at one another in unintelligible code). You see your question followed by…but it's too late! Your brain has already shut down, and you feel the tension and frustration as knots in your belly.

Sound familiar?

Yes? Then this book's for you! My guess is that you're probably standing in a bookstore or in the airport, checking out a magazine stand for something that might excite. Because you're reading this in some such upscale place, you can probably afford an iPhone, a Mac, a car, and plane tickets. You're probably intrigued by the burgeoning industry of handhelds and the geometric rate at which memory and microprocessors are evolving…how quickly ideas can be turned into startlingly new computing platforms, into powerful software applications, into helpful tools and clever games…perhaps even into greenbacks! And now you are wondering if you can get in on the action—using your intellect and technical savvy to serve the masses.

How do I know this about you?

Easy! Through years of teaching students to program, I know that if you're still reading this, then you're both intelligent enough and sufficiently driven to step onto the playing field of programming, especially for a device as sweet as the iPhone or as sexy as the iPad. If you identify with and feel connected to the person I've described above, then I know you. We were introduced to one another long ago.

You are an intelligent person who may have mental spasms when reading complex code—even if you have some background in programming. And even if you do have a pretty strong background in various programming languages, you are a person who simply wants an easy, on-point, no-frills strategy to learn how to program the iPhone and iPad. No problem! I can guide you through whatever psychological traffic jams you typically experience and help you navigate around any technical obstacles, real or imagined. I've done this a thousand times with my students, and my methodology will work for you, too.

The Approach I Take

I don't try and explain everything in minute detail. Nor do I expect you to know every line of code in your iPhone/iPad application at this stage. What I will do is show you, step by step, how to accomplish key actions. My approach is simultaneously comprehensive and easy-going, and I take pride in my ability to instruct students and interested learners along a wide spectrum of knowledge and skill sets.

Essentially, I will lead you, at your own pace, to a point where you can code, upload, and perhaps sell your first iPhone/iPad app, simple or complex. *Good news*: the most downloaded apps are *not* complex. The most popular ones are simple, common-sense tools for life…finding your car in a parking lot, or making better grocery lists, or tracking your fitness progress. However, when you complete this book, you may want to graduate to other books in the Apress and Friends of ED series. You have quite a few options here, and down the road I'll advise you regarding the best ways to move forward. Right now, though, you may want to read a little about me so you will feel confident in taking me on as your immediate guide in this exciting app-venture.

May you experience great joy and prosperity as you enter this amazing and magical world.

Peace!

Rory A. Lewis, PhD, JD

Chapter 1

Before We Get Started

This introductory chapter will make sure that you have all the required tools and accessories to proceed fully and confidently. Some of you may already be solid on these points and feel ready to jump right in. If so, you may want to jump ahead to Chapter 2 and start immediately on your first program.

It will behoove you, though, to understand why I teach certain things and skip others. For those of you who have never done it, programming in Objective-C is quite a challenge—even for my engineering students who know Java, C, and C#. Nevertheless, with the appropriate preparation and mindset you will accomplish this.

So I urge you to read on. The time you will invest in this chapter will be well worth it in peace of mind and confidence. Chapter 1 will help structure the way that your brain will file all the rich content that is to come.

Necessities and Accessories

In order to program for the iPhone and/or iPad, and to follow along with the exercises, tutorials, and examples presented in this book, you'll need to pay attention to certain minimal requirements:

- Intel-based Macintosh running Leopard (OS X … 10.5.3 or later)
  - If it was bought after 2006, you're OK.
  - You don't need the latest revved up Mac. If you haven't bought one yet, I suggest you get a basic, no-frills MacBook.
  - If you do own an older Mac, then add some RAM. Make an appointment at the Genius Bar at an Apple Store and ask them to increase the RAM as much as possible.
- Become a registered developer via the iPhone/iPad Software Development Kit (SDK).

- If you are a student, it's likely that your professor has already taken care of this, and you may already be registered under your professor's name.
- If you are not a student, then you will need to follow these steps to sign up.

1. Go to `http://developer.apple.com/programs/iphone/`, which will bring you to a page similar to the one shown in Figure 1–1. Click the Enroll Now button.

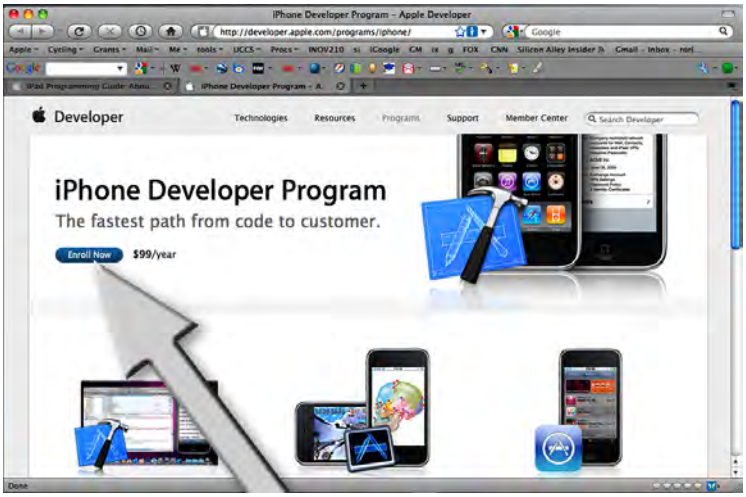


Figure 1–1. *Click the Enroll Now button.*

2. Click the Continue button as illustrated in Figure 1–2.

Figure 1–2. *Click the Continue button.*

3. Most people reading this book will select the "I need to create a new account for …" option (arrow 1 in Figure 1–3). Next, click the Continue button as illustrated by arrow 2 in Figure 1–3. (If you already have an existing account, then you have been through this process before; go ahead with the process beginning with the "I currently have an Apple ID ..." option, and I'll meet you at step 6, where we will log onto the iPhone/iPad development page and download the SDK.)

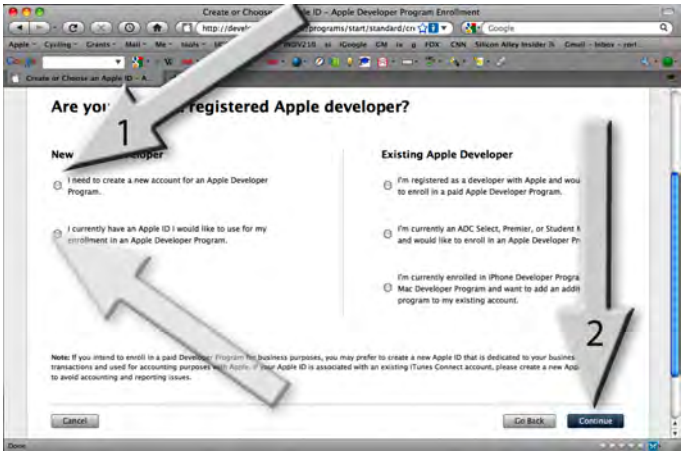


Figure 1–3. *Click the "I need to create an Apple ID …" option to proceed.*

4. You are probably going to be enrolling as an individual, so click the Individual link as illustrated in Figure 1–4. If you are enrolling as a company, click the Company option to the right and follow the appropriate steps; I'll meet you at step 6.

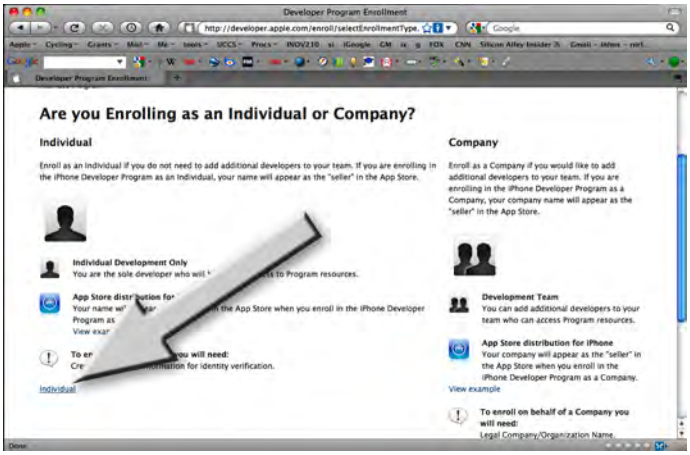


Figure 1–4. *Click the Individual option.*

5. From here you will enter all your information as shown in Figure 1–5 and pay your fee of $99.00 for the Standard Program. This provides all the tools, resources, and technical support you will need. (If you're reading this book, you really do not want to buy the Enterprise program at $299, as it is for commercial in-house applications.) After paying, save your Apple ID and Username; then receive and interact with your confirmation email appropriately.

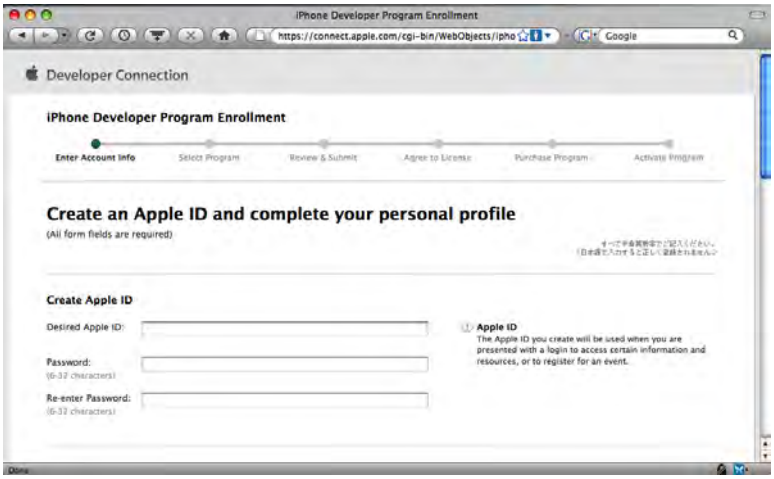


Figure 1–5. *Enter all your information accordingly.*

6. Use your Apple ID to log into the main iPhone/iPad development page. Scroll down to the bottom of the page and download the SDK as illustrated in Figure 1–6. Extract the necessary icons onto your dock. Included with the Apple SDK that you've now downloaded is Apple's integrated development environment (IDE). This is a programming platform that contains a suite of tools, sub-applications, and boilerplate code that all enable us to do our jobs more easily. We will use Xcode, Interface Builder, and the iPhone/iPad Simulator extensively, so I advise you to bring these icons to your dock to save yourself tons of time searching for them.

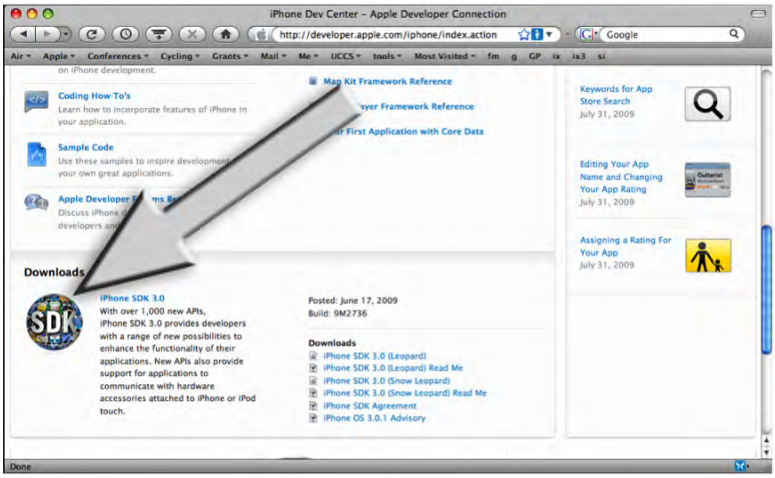


Figure 1–6. *Having logged in as a Registered Apple Developer, you can now scroll down to the bottom of the page and download the SDK.*

7. Bring Xcode to your dock. by choosing Macintosh HD ➤ Developer ➤ Applications ➤ Xcode.app and dragging it onto your dock as illustrated in Figure 1–7. In the same way, bring Interface Builder to your dock by choosing Macintosh HD ➤ Developer ➤ Applications ➤ Interface Builder.app and dragging it. Finally, bring the iPhone/iPad Simulator to your dock by choosing Macintosh HD ➤ Developer ➤ Platforms ➤ iPhone/iPad Simulator Platform and dragging it.

Figure 1–7. *Xcode, Interface Builder, and the iPhone/iPad Simulator—locked and loaded, ready to roll!*

NOTE: Whenever I say "iPhone" or "iPad," I am referring to any iPhone or iPad OS device. This includes the iPod touch.

What I Won't Teach You

With your Xcode, Interface Builder, and iPhone/iPad Simulator tools installed and ready to access easily, you're ready to roll. But wait! You need to know where we're going.

First, though, let me say something about where we won't be going—what I will *not* be covering. I will not attempt to teach you how every line of code works. Instead, I will take a subsystem approach, indicating which pieces or sections of code will serve you in which situations.

While this book is designed to impart to you, the reader and programmer, a comprehensive understanding and ability, we will be dealing in molecules rather than atoms or subatomic particles. The emphasis will be on how to recognize general attributes, behaviors, and relationships of code so that you need not get bogged down in the symbol-by-symbol minutiae. I will get you to a place where you can choose those areas in which you may want to specialize.

Computer Science: A Broad and Diverse Landscape

Consider this analogy: suppose that the iPhone/iPad is a car. Most of us drive cars in the same way that we use computers. Just as I would not attempt to teach you how every part of the car works if I were giving you driving lessons, I would not—and will not—approach iPhone and iPad programming with fundamental computer engineering as the first step.

Even great mechanics who work on cars every day rarely know the fundamental physics and electronics behind the modern internal combustion engine, not to mention all the auxiliary systems; they can drive a car, diagnose what's wrong with it when it needs servicing, and use their tools and machines (including computers) to repair and tune it optimally. Similarly, clever programmers who create the apps for the iPhone and iPad rarely know the fundamental coding and circuit board designs at the root of the Apple platforms. But they can use these devices, they can envision a new niche in the broad spectrum of applications needs, and they can use their tools and applications—residing on their desktops and laptops—to design, code, and deliver them to the market.

To continue with this analogy, programming the iPhone or iPad is like playing with the engine of your car—customizing it to do the things you want it to do. Apple has designed a computing engine every bit as fantastic as a V8 motor. Apple has also provided a pretty cool chassis in which we can modify and rebuild our computing engine. There are restrictions on how we can "pimp" our iPhone/iPad cars, and, for those of you who have never pimped a car, I will demonstrate how to maximize creative possibilities while honoring these restrictions.

I'm going to show you, without too much detail, how to swap oil filters, tires, seats, and windows to convert it into an off-road car, a hot rod, a racing car, or a car that can get us through the jungle. When you've mastered this book, you will know how to focus on and modify the engine, the transmission, the steering, the power train, the fuel efficiency, or the stereo system of the car.

Why Purgatory Exists In Objective-C

My Assumption: you've never worked on a car, and you've never gotten grease on your hands, and you want to pimp one of the world's most powerful automobiles—with a complex V8 engine. I'm going to show you exactly how to do this, and we're going to have fun doing it!

First, you need to know a little about how we even came to have the souped-up car with the V8—that is, the iPad. In 1971, Steve Jobs and Steve Wozniak met, and five years later they formed Apple, producing one of the first commercially successful personal computers. In 1979, Jobs visited Xerox PARC (Palo Alto Research Center), and secured the Xerox Alto's features into their new project called the *Lisa.* Although the Alto was not a commercial product, it was the first personal computer to use the desktop metaphor and graphical user interface (GUI). The Lisa was the first Apple product with a mouse and a GUI.

In early 1985, Jobs lost a power struggle with the Board of Directors at Apple, resigned from the company, and founded NeXT, which eventually bought out Apple in 1997. During his time at NeXT, Steve Jobs changed some critical features of the code on the Macintosh (Mac) to talk in a new language, a very intense but beautiful language called Objective-C. The power of this language was in its ability to efficiently use objects. Rather than reprogramming code that was used in one portion of the application, Objective-C *reused* these objects. Jobs' brain was on overdrive at the time, and this incredible code took this new language of Objective-C to new heights. His inspiration was fused into the guts of the Mac by creating a metalanguage we call Cocoa. A metalanguage is a language used to analyze or define another language. As I've indicated, Objective-C is a very challenging beast, and you can think of Cocoa as the linguistic taming of the beast, or at least the caging of the beast.

As an "absolute beginner" to the world of programming, you cannot be expected to be concerned with the subtleties of coding language distinctions. I am simply giving you an overview here so that you will have a rough historical context in which to place your own experience. The main point I'm making here is that Objective-C and Cocoa are very powerful tools, and both are relevant to the programming of the iPhone/iPad.

Houston, We Have a Problem

This is the essence of the challenge that intrigued me, and led to the design of my original course. How can one teach non-engineering students, perhaps like you, something that even the best engineering students struggle with? At the university level,

we typically have students first take introductory programming classes, and then proceed to introductory object-oriented programming, such as C# or C++.

That being said, we are going to dive *head on* into Objective-C! At times, I'm going to put blindfolds onto you; at other times, I'm going to cushion the blows. There will be times when you may need to reread pages or rewind video examples a few times— so that you can wrap your head around a difficult concept.

How We'll Visit Purgatory Every Now and Again

There are specific places in my courses where I know that half the class will immediately get it, a quarter will have to sweat over it before they get it, and the remaining quarter will struggle and give up. This third group will typically transfer out of engineering and take an easier curriculum. I know where these places are, and I'm not going to tell you. I'll repeat that. I will not tell you.

Don't worry, I won't allow you to disturb a hornet's nest (of Objective-C issues) and get stung to death. Nor will I mark off those concepts that you may find difficult. I'm not going to explain this now. Just accept it! If you just relax and follow my lead, you'll get through this book with flying colors.

When you do find yourself in one of those tough spots, persevere. You can always reread the section or rewind the video examples. In this iPhone and iPad programming adventure, it won't serve you to skip it. There are only about three of these critical areas in the book, and I've made them as easy as I can. There are also blogs and discussion boards you can access in order to discuss problems and share your thoughts with others.

Looking Forward … *Beginning iPhone 3 Development: Exploring the iPhone SDK*

Down the line, some of you may want to continue your iPhone and iPad programming adventure by reading Dave Mark and Jeff Lamarche's book, *Beginning iPhone 3 Development* (Apress, 2009). Remember the analogy of becoming a mechanic for an automobile with a V8 engine mounted on a basic chassis? Their book presumes that the readers know what a carburetor is, know what a piston is, and that they can mount racing tires and super fly rims on their friends' pimped-up wheels.

In other words, they assume that you understand the fundamentals of object-oriented programming: that you know what objects, loops, and variables are, and that you are familiar with the Objective-C programming language.

On the other hand, I assume that you don't know, for example, what a "class" is, or what a "member" or "void" is. I imagine that you have no idea how memory management works on an iPhone/iPad and, furthermore, that you never had an interest – *until now* – in understanding an array, or an SDK.

What You Will Learn

When students start a challenging class, I have found that it works wonders to have them create something real cool, and with relative ease. At each stage of this process, I will typically present an example that you can read, see, and digest right away. Later on, we will return to analyze some of the early steps and go into more detail. I will explain how we accomplished some task or action the first time *without even knowing it*. Then, by comparing the first time through with subsequent modifications, you will learn how to tweak the program a little here, a little there. This way, you'll stay on track—motivated and inspired to absorb the next new batch of tricks, lessons, and methods.

Creating Cool and Wacky Apps: Why I Teach This Way

You've heard the bit about how we best remember things: doing is better than seeing, which is better than hearing, and so on. Well, I know that students love humor—and guess what! We remember funny stories and lessons much better than we remember dull and boring ones. I have found that, without exception, when students work on code that is fun and wacky, they tend to spend much more time solving it.

The more we apply ourselves mentally toward the solution of a problem, the more neural connections are made in our brains. The more neurons we connect, the more we remember and—most importantly—the less apt we are to waste time on ineffective methods.

The more time we spend on a particular topic, the more chance there is that you will experience gut feelings about whether a particular methodology for solving a project is on track or not. So, as we proceed, be aware that I am employing humor to burn computer science and Objective-C concepts and methods into your brain without your exerting any conscious effort.

It is common for my students to contact me after receiving a difficult homework assignment. First, they'll send me a tweet asking if they can Skype me. One particular night, I was playing chess with a colleague when I received a tweet asking if I were available. "Of course," I responded. I warned my colleague, also a professor at the University of North Carolina, that students whom he knew were about to appear on Skype. When they buzzed in, sure enough: four of my electrical engineering students, wide-eyed and smiling. "*Hey, Dr. Lewis, we finally got it, but Dude! The last method you assigned---.*"

When we finished our conversation, and I turned off my Mac, it was 12:30 am. My colleague asked, "Rory, I never called a professor this late in the evening—much less *after midnight*! Shouldn't they ask these questions during office hours?!" He was probably right, but after thinking about it for a minute I replied, "I'm just happy that they're working on my wacky assignment!" As we set up the next chess game, he murmured something about how I might be comfortable in the insanity ward.

The point is that I want you to read this entire book. I want you to work all the examples and to feel elation as you complete each assignment! I have done everything I can to

make this book enjoyable. If you choose to engage with the ideas contained herein, this book will change your life!

By the way, successfully navigating these lessons will make you a certified geek. Everybody around you will sense your growing ability and will witness your transformation; as a result, they will seek you out to request that you write apps for them.

Evangelizing to Your Grandmother … What You Coded Is Crucial!

It's important that you not let complex code turn you inside-out. Just two minutes ago, a student walked into my office—so confused that he couldn't even tell me what it was he didn't know. He said something like, "My second order array worked fine in-line, but not as a class or a method." I said, "No, that's too complex! Here's an easier way of saying it …"

I described how he had a long line of "stuff" going in one end and being spat out the other – and it worked really well. But, when he put it in a *method*, he couldn't see the start of the long line of stuff; when he put it in a *class*, he couldn't see *any* of the stuff!"

"Wow! I know what I did wrong, Dr. Lewis. Thank you!" Now, as I type this, he's explaining it to his two buddies who came in yesterday and tried to ask the same question. Don't worry, the confusion that drove these questions – such as the distinctions between "classes" and "methods," and other coding entities, will be covered later in this book. All in good time!

If you can keep your feet on the ground and transform complex things into simpler ideas, then you can remember them—and master them. Grasp this concept, and you will be able to convert your far out ideas into code—and who knows where that will take you! This is why I am so determined to impart to you the ability to convert things your grandmother wants to be able to do into iPhone and iPad programming language.

How Does This All Work?

Before we start our first program in Chapter 2, it's critical that you are able to step back and know where we've been, where we are now, and where we will go next. Looking at Figure 1–8, you can see a gray strip containing two icons that represent Mac OSX and the SDK, which includes Interface Builder and Xcode. These will be explained in detail later; 90 percent of this book deals with the items in this strip.

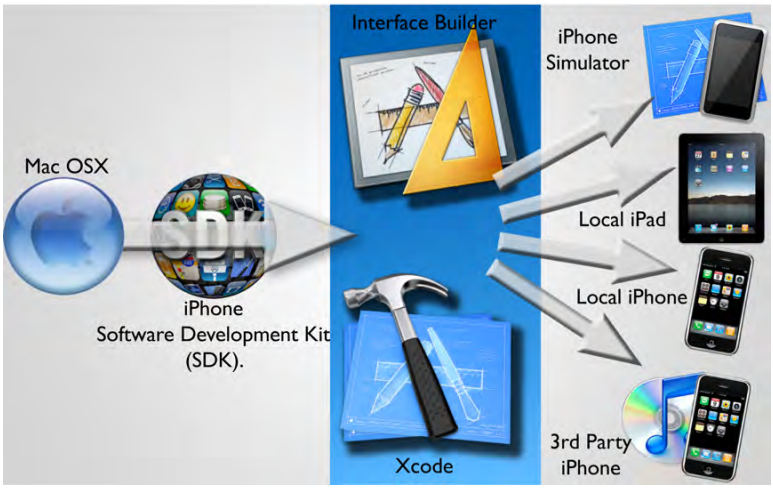


Figure 1–8. *The iPhone and iPad app programming landscape. Mac OS X 10.5.3 or later is housing the iPhone and iPad SDK. I will teach you how to use the SDK's Xcode and the Interface Builder to create apps. Once you create an app, there are four ways to run it: using iPhone/iPad Simulator, with your iPhone attached to your Mac, with your iPad attached to your Mac, or downloaded via iTunes to a third-party iPhone/iPad.*

The blue band in the middle is where we are presently. To the left of this area is where you've been. You have a Mac (purchased after 2006 and running Mac OS X 10.5.3 or higher), and we've just walked through the process of downloading the iPhone and iPad SDK (Figures 1–1 thru 1–6). We have also extracted Interface Builder, Xcode, and the iPhone/iPad Simulator and positioned them onto your dock (Figure 1–7). That's where we are now.

In Chapter 2, we will start using Xcode and Interface Builder to turn you into a bona-fide geek! We're going to run all the programs we make by compiling them to one of several possible locations, the icons for which are to the right of the central blue area. The primary location will be the iPhone/iPad Simulator. The secondary locations will be your local iPhone and/or your local iPad. Lastly, we could use iTunes to upload your iPhone and/or iPad App to the App Store where people can purchase it or download it for free. This is where we are going.

The two central objects in Figure 1–8, as you now know, are where we will spend the vast majority of our time within this book. We'll be using Xcode to type in code, just like the serious geeks do. I'll show you how to operate all its features such as file management, compilation, debugging, and error reporting. Interface Builder is the cool way Apple allows us to drag and drop objects onto our iPhone/iPad apps. If you want a button, for instance, you simply drag and drop it where you want it to be located on the virtual iPhone or iPad.

Essentially, we'll use Xcode to manage, write, run, and debug your app—to create the content and functionality. We'll use Interface Builder to drag and drop items onto your interface until it looks like the colorful and cool application you envisioned—to give it the style, look, and feel that suits your artistic tastes.

After we integrate all the interface goodies with the code we wrote in Xcode, we might get advanced and tweak the parameters dealing with memory management and efficiency. But that's jumping too far ahead in our story.

Our Road Map: Using Xcode and Interface Builder

Very often authors of programming books do the same old thing. They first present a very simple, ubiquitous "Hello World" application and then throttle the user with intense code that loses a great many readers and students straight away. Utilizing Objective-C (being run in Cocoa) along with the iPhone and iPad SDK, I've had to really rethink this introductory process. I have identified three challenges here.

- Teaching you "Hello World" and then going into advanced technologies and APIs would be counter-productive.
- It makes no sense to randomly choose one of the many ways to say Hello to the world from your iPhone or iPad. They are all going to be necessary to have in your toolkit at a later date.
- Trying to write a simple "Hello World" application in Objective-C is more involved than the beginner is ready for, unless we break up the process into stages or layers.

My solution to overcoming these issues is simple. I'll show you how to say Hello to the world from your iPhone/iPad in not one, not two, but quite a few different ways. Each time, we'll go a little bit deeper, and we'll have a blast as we do so.

Each time you travel down the road into the land of Xcode, you are immediately asked what type of vehicle you'd like to drive. A Jeep? A race car? A convertible? By focusing on basics, I am going to show you how to "drive" *in Xcode*. The objective here will be to gain competence and confidence in whatever style of vehicle we must access. So, let's take a look at exactly what these different vehicles have to offer. Here I would like you to follow along with me.

Getting Ready For Your First iPhone/iPad Project

Assuming that you have already downloaded the SDK and installed Interface Builder, Xcode, and the iPhone/iPad Simulator, open up your Mac and click the Xcode icon on your dock. Your screen should look similar to Figure 1–9. Up pops the Welcome to Xcode window; it includes all your iPhone and iPad resources.

Figure 1–9. *After clicking the Xcode icon, you will see the "Welcome to Xcode" screen. Keep the "Show at Launch" option checked.*

Look at the bottom-left corner of the pop-up window and make sure you keep the Show at Launch option checked. There are many valuable resources here that you will find handy. I suggest that, after you have completed Chapter 4, you take a little time and explore these resources—give them a test drive, so to speak. This practice will open all kinds of creative doors for you.

Without actually starting a new project, let's walk up to the showroom floor and check out some of the models we might be driving. To open a new project in Xcode, enter Command + Shift + N simultaneously. This three-key shortcut, depicted in Figure 1–10 as (⌘⇧N), will open a window that showcases the different types of vehicles that you can drive in the land of Xcode.

Figure 1–10 displays the six vehicle models: Navigation-based Application, Open GL ES Application, Tab Bar Application, Utility Application, View-based Application, and Window-based Application.

Early on, most of our travel in Xcode will be by one of the latter two styles shown. Switching back to computer terms, View-based Application and Window-based Application are the structures we will utilize in the basic development cycle for the iPhone/iPad. It is here that we will access cool gadgets and components.

Don't worry, I haven't forgotten our goal of creating a simple "Hello World" application. We will say Hello to the world while using a number of the six options, and you will become familiar with each.

Figure 1–10. *When choosing a template for a new project, you will have a choice of six "vehicle" styles.*

The Accompanying Screencasts

All figures shown in this book have been captured from my screen as I write the code—in a screencast. For example, the helloWorld_001 example in Chapter 2 is located at `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/002_helloWorld_002.htm`.

It is not necessary to view the aforementioned screencast, since I have included all the instructions in Chapter 2. However, I've heard students say that it's fun to retrace what they heard in the lesson. These video examples tend to be rather condensed. If you would like to follow along with the screencasts, please note these recommendations:

- Stop the video when I get ahead of you. Rewind it and get back on track with me.
- After you can complete the project in full, save the screencast to another folder. Then, go through it again with fewer stops until you can master it … and compile it.
- For the competitive among you, perhaps a goal is to execute the code in time with me as I go. Generally though, I want you to feel good and comfortable with programming at a high level. It would behoove you to practice this for all the examples in the book.

The Accompanying PDFs

I also provide a PDF version of the keynote slides that I give to my students at the University of Colorado. These PDFs—which are not required, but merely supplemental—show all the slides from this chapter. There are also links for those of you who want to probe deeper into subject matter that is not covered in this book.

> **NOTE:** You can access videos and supplementary materials at the `www.apress.com` web site.

Pretending Not to Know: The Art of De-Obfuscation

Before we begin in earnest, I want to reiterate that I am going to show you how to program while knowing only the essentials. As we move forward, I will explain concepts a little deeper. However, I will only do this once we've gotten your head wrapped around the easy concepts. This is a new way of teaching, and I have had great success with it.

You may think I've completely lost my mind, but I ask you to follow my instructions anyway. If you have a question that I don't appear to address, trust me that it's not important at the time. We will cover it down the road!

How We'll Travel Through Each Step

This book is completely inclusive. Even though I provide video tutorials for the exercises in this book, you don't need any of it. You can read this book alone, without any Internet connection, and everything you need can be found within these pages.

So, now that you've finished checking your system parameters, signed up as an official Apple developer, downloaded the SDK, extracted the essential tools, and configured your dock, it is time to advance to Chapter 2 and create some code.

Turn the page!

Chapter 2

Blast-Off!

The first program we shall attempt, as mentioned in Chapter 1, will be a basic and generic “Hello World” application. I need to clarify, though, that as regarding most things in the Objective-C context, it will not necessarily be simple.

Our first adventure with this new set of tools will be saying “Hello” to the world from the View-based Application template in Xcode. Later, we will say “Hello” to the world from the Navigation-based Application template, first in a very basic way and then with some modifications.

Besides the information I present here in this book, including various screenshots, I also offer you screencasts, which are available at my website. In this chapter we will work through three examples, and you can access the screencasts at these links:

http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/001_helloWorld_001.htm
http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/002_helloWorld_002.htm
http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/003_helloWorld_003.htm

1. Before opening Xcode, first close all open programs so that you will be able to optimize your processing capabilities and focus your undivided attention on this new material. Press Command + Tab and then Command + Q to close everything until only the Finder remains on your screen. Find and click the Xcode icon in your Dock to open it. You will be presented with the Xcode “Welcome” screen discussed in Chapter 1. See Figure 2–1.

Figure 2–1. *Click the Xcode icon in your Dock to open it. You will be presented with the "Welcome to Xcode" frame, as discussed in Chapter 1.*

2. Now, open a new project in Xcode. The two ways to accomplish this are by using keyboard shortcuts or by clicking your mouse. I *strongly* suggest that you use keyboard shortcuts. These will save you time and make you feel like a pro. Be aware that the best way to *not* get work as an iPhone and iPad App developer is to use your mouse for functions that can be done via shortcuts. Using your keyboard, press Command + Shift + N at the same time. These three keystrokes appear in Figure 2–2 as ⌘⇧N. (Using your mouse to open a new project, you would choose File ➤ New Project.) Your screen should show the New Project wizard as depicted in Figure 2–2.

Figure 2–2. *Select the View-based Application template for your new project.*

NOTE: My View-based Application template icon was highlighted by default; yours may not be. Regardless, click on it, and save the new project to your desktop as `helloWorld_001`.

3. As soon as you save this project to your desktop, Xcode instantiates the item `helloWorld_001` as indicated by the name on the top of the window (see Figure 2–3). If this looks a bit complicated, stay cool … don't freak out! This is Apple's way of arranging all the goodies that you will eventually use to write complex apps. For now, just follow along and try to set aside all the questions you may be asking. As shown in Figure 2–3, double-click to open up the Classes folder. (Don't ask yourself what "Classes" means yet. Just open the folder. In due time, you'll know plenty about classes.)

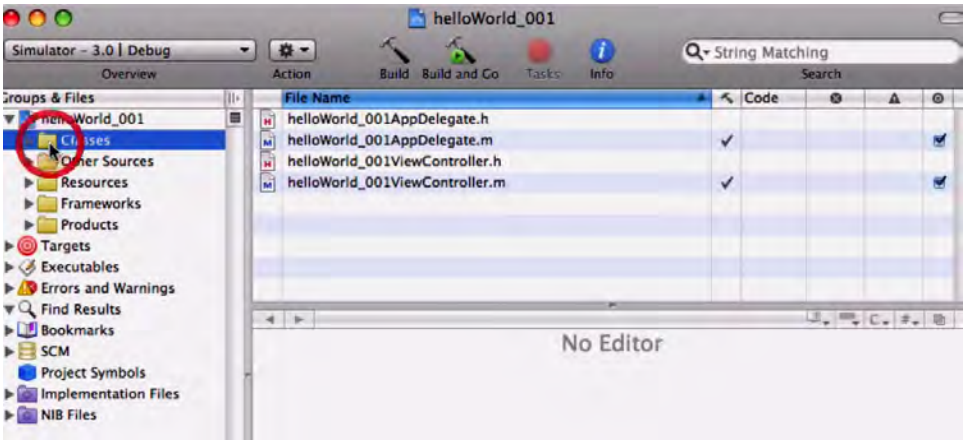


Figure 2–3. *Click the indicated icon to open the Classes folder.*

4. As you can see in Figure 2–4, Xcode has created four files, each starting with a prefix identical to the name of our project, `helloWorld_001`. We are going to select what is called the *interface* file; it has the name of our project followed by `ViewController.h`. Joining these together creates the file `helloWorld_001ViewController.h`. At this point, don't worry about the `.h` extension. Later, I'll explain all the gory details about what we're doing, and it'll make more sense. Next step!

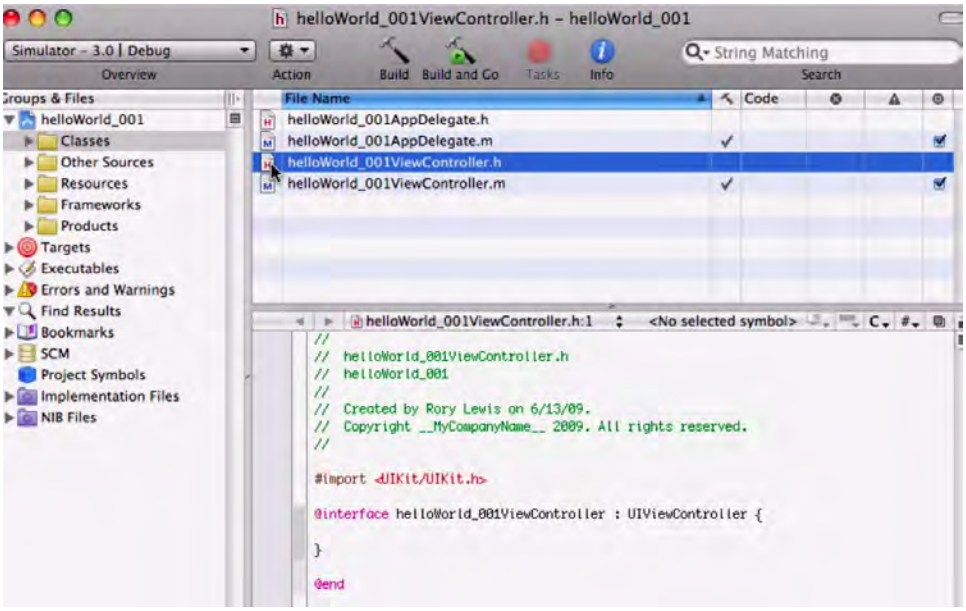


Figure 2–4. *Open the Interface file. This is what your screen looks like before you write your two lines of code.*

5. In Figure 2–4, you probably noticed several lines of code in green; these are internal comments, from one coder or programmer to another. These comments are "invisible" to the computer and because we are presently interested in code that talks to the computer, focus on the line immediately following these comments:

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

This line of code imports something we call the UIKit framework. It provides your application with goodies and shortcuts that make it easier to manage your user interface. This command brings in code that is organized into classes, which you can think of as preprogrammed packets of sub-routines. As a result of the presence of the UIKit, you don't have to write or rewrite a lot of fundamental technical code, but can instead take advantage of certain ready-to-go interface packages to help you design your app. These classes facilitate in the presenting of data to the user and in the responding to user input.

NOTE: When naming things in code, we will represent "User Interface" with the initials UI.

After that line, we see:

```
@interface helloWorld_001ViewController : UIViewController {
```

There are two things to note here. First, the @ symbol tells the innermost part of Xcode, which transforms your code into action, that it's got something essential and important to announce. In fact, we call any statement beginning with @ a "directive." This @interface directive tells Xcode that we have interface stuff to tell it concerning "helloWorld_001," and that the particulars will be enclosed within brackets {}.

But look at your screen, right after the opening bracket, {. It's empty! We're not saying anything yet —right? We know what we want to do, though, and that is to say "Hello" to the world from inside our iPhone/iPad. Now that we've got the compiler's attention with the @interface directive, we want to tell it that we first want our Interface Builder outlet, IBOutlet, to say something … and that something will be written on a UILabel.

NOTE: In our code, we will represent "Interface Builder" with the initials IB.

I want you to do this now by typing the following on the line below the open bracket:

```
IBOutlet UILabel *label;
```

Don't forget the semicolon ";" at the end of the line. This symbol tells the compiler that you're finished talking to it for the moment. Don't worry about the * in front of the word "label." We'll get into this later.

Thus, your code should look like this – with the boldfaced line indicating your insertion. Also, you can refer to Figure 2–5.

```
//
//  helloWorld_001ViewController.h
//  helloWorld_001
//
//  Created by Rory Lewis on 6/13/09.
//  Copyright __MyCompanyName__ 2009. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface helloWorld_001ViewController : UIViewController {
        IBOutlet UILabel *label;
}

@end
```

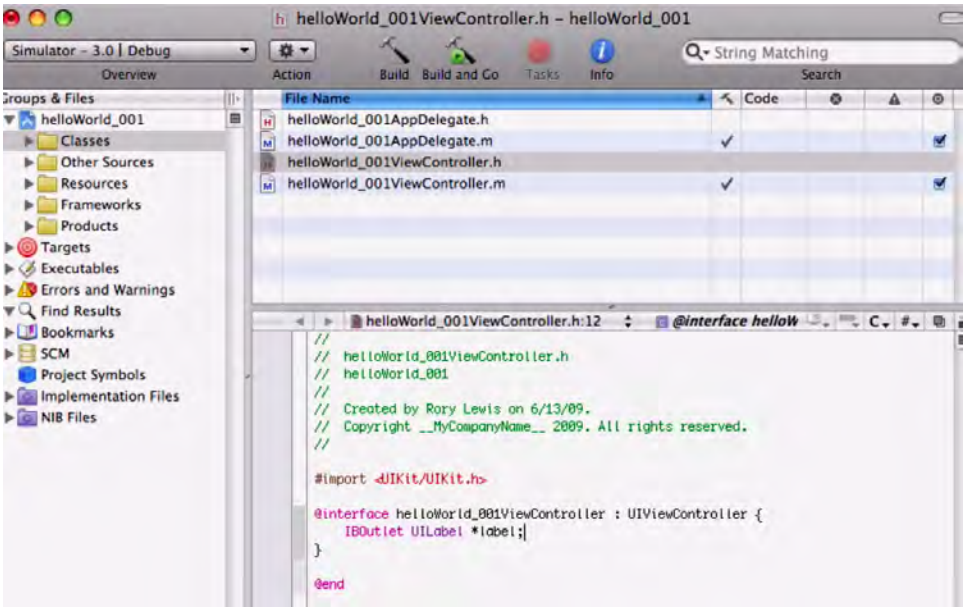


Figure 2–5. *This is a view of your file after you've entered the new code.*

NOTE: If you want to know more of the technical details, that's great! Go to the slide show at `http://www.rorylewis.com`.

6. Now that you have written your first line of code, we need to instruct the compiler about one more thing here. Specifically, we want to have the button on our iPhone or iPad screen use methods that Apple has already coded for us. We want to add an `IBAction`, to which we will refer as "hello." To accomplish this, insert the following line after the closing bracket.

```
- (IBAction)hello:(id)sendr;
```

Again, make sure that you remember to put the semicolon at the end of the line, and don't worry about the hyphen (minus sign). I'll get to that later.

Next, highlight the line exactly as shown in Figure 2–6. Using your keyboard, enter Command + C. This useful shortcut is depicted in Figure 2–6 as ⌘C, and it will copy the highlighted text into the *clipboard*, a short-term memory cache. (You can also use your mouse to copy this line, by choosing Edit ➤ Copy.)

This is what your file should look like now.

```
//
//  helloWorld_001ViewController.h
//  helloWorld_001
//
//  Created by Rory Lewis on 6/13/09.
//  Copyright __MyCompanyName__ 2009. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface helloWorld_001ViewController : UIViewController {
        IBOutlet UILabel *label;
}

- (IBAction)hello:(id)sendr;

@end
```

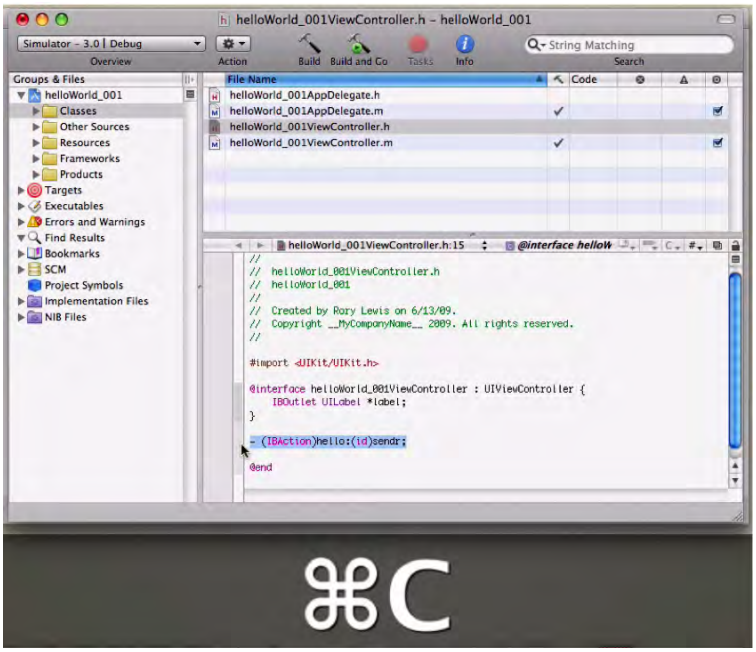


Figure 2–6. *Copy your action code by highlighting the indicated line and using the shortcut.*

7. Now that you are finished with this file, save it by using the shortcut Command + S, highlighted in Figure 2–7 as ⌘S. This is the preferred method of saving—rather than using your mouse.

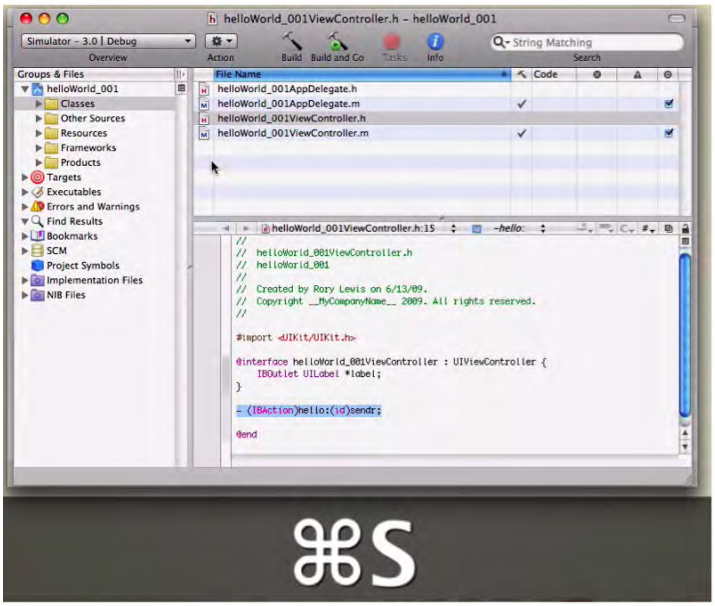


Figure 2–7. *Save your file by using the indicated shortcut.*

8. As you can see in Figure 2–8, we are finished working on the file named `helloWorld_001ViewController.h`. Now, we want to work on a different file, `helloWorld_001ViewController.m`. This file has an identical name – except for the one letter extension!

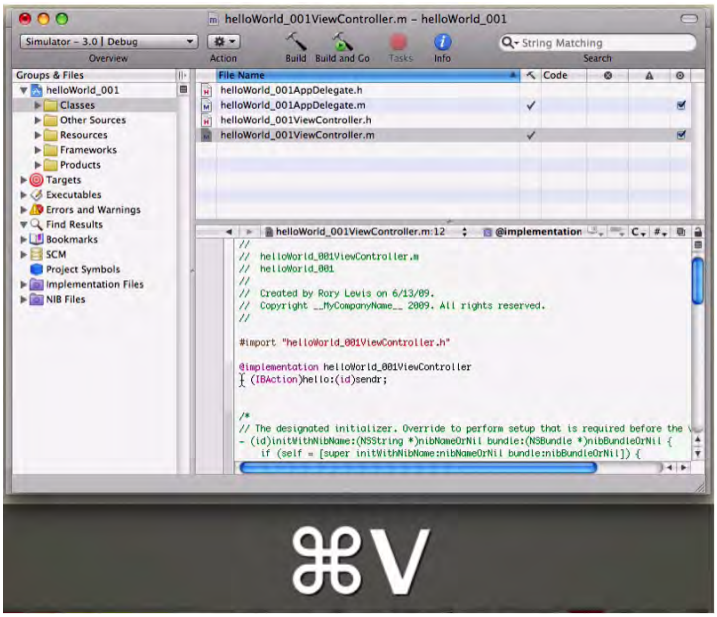


Figure 2–8. *Add to your new View-Based Application file by pasting the saved line of code (from the previous file) into the indicated area.*

Let me talk a little bit about the difference between these two files: one with the .h suffix, the other with the .m suffix.

The `ViewController` manages the interactions your code has with the display, and it manages the user's interactions with your code. It contains a view, but it is not a view itself. The `ViewController` is a *class*, of which you only have a minimal understanding so far. What I want you to get, though, is that every class consists of two parts: the header (`.h`) file and the implementation (`.m`) file.

I want you to read this next part aloud, and I don't care if you're in the bookstore! OK? "We tell the computer in a header file what types of commands we will execute in the implementation file."

Now, let's say it again in context to our code: "We tell the computer in the `helloWorld_001ViewController.h` file what types of commands we will execute in the `helloWorld_001ViewController.m` file."

Well, admit it – that wasn't so bad!

Because you have now handled the first half of this arrangement, let's move to the implementation file and execute the commands that we have announced. To do this, we open `helloWorld_001ViewController.m`, and then we scroll down past the green lines – i.e., all of the comments.

I want you to use the shortcut Command + V to paste the copied line from the last step in our dealings with the header file:

```
- (IBAction)hello:(id)sendr;
```

Refer to Figure 2–6, if necessary. This shortcut is shown in Figure 2–7 as ⌘V. (Of course, you can use your mouse to paste it by choosing Edit ➤ Paste, but you really should get into the habit of using keystroke shortcuts!)

```
//
//  helloWorld_001ViewController.m
//  helloWorld_001
//
//  Created by Rory Lewis on 6/13/09.
//  Copyright __MyCompanyName__ 2009. All rights reserved.
//

#import "helloWorld_001ViewController.h"

@implementation helloWorld_001ViewController
- (IBAction)hello:(id)sendr;
```

9. As you can see in Figure 2–9, I want you to delete the semicolon at the end of the line that you just pasted, which consisted of us declaring an *action*.

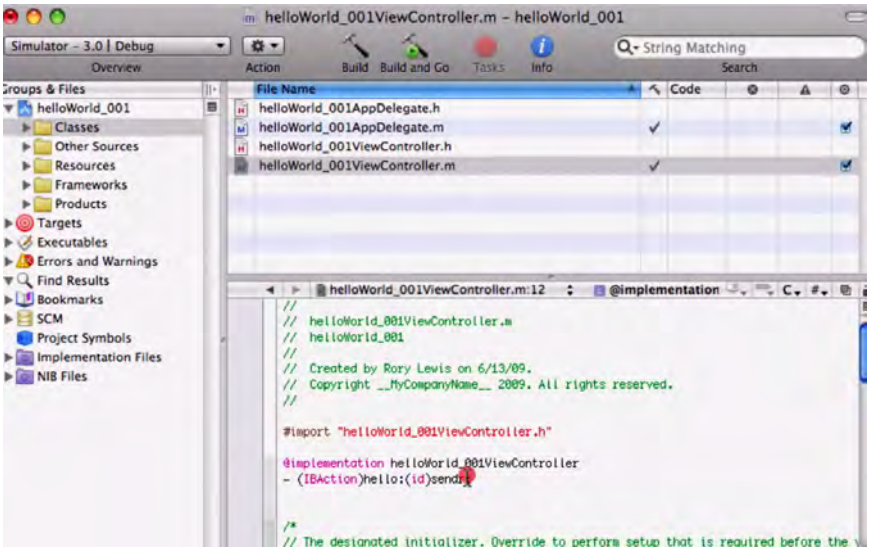


Figure 2–9. *Delete the semicolon as seen in my red dot from the screen flow.*

Now that we're in the implementation file, we don't want to declare an action, we want to *implement*. Here is what the code looks like before you insert the actions that do this:

```
//
//  helloWorld_001ViewController.m
//  helloWorld_001
//
//  Created by Rory Lewis on 6/13/09.
//  Copyright __MyCompanyName__ 2009. All rights reserved.
//

#import "helloWorld_001ViewController.h"

@implementation helloWorld_001ViewController
- (IBAction)hello:(id)sendr
```

10. Having deleted the semicolon after the "sendr" term, you are now ready to insert a command to accomplish our objective. We want to link the label you created (in step 6) to the means by which we may provide text to the screen. Enter this code immediately after "sendr":

```
{label.text = @"Hello World!";}
```

Hit Return after the opening bracket, and then hit Return again after the semicolon. Your file should now look like this:

```
//
//  helloWorld_001ViewController.m
//  helloWorld_001
//
//  Created by Rory Lewis on 6/13/09.
//  Copyright __MyCompanyName__ 2009. All rights reserved.
//

#import "helloWorld_001ViewController.h"

@implementation helloWorld_001ViewController
- (IBAction)hello:(id)sendr{
    label.text = @"Hello World!";
}
```

Refer to Figure 2–10 to see the results.

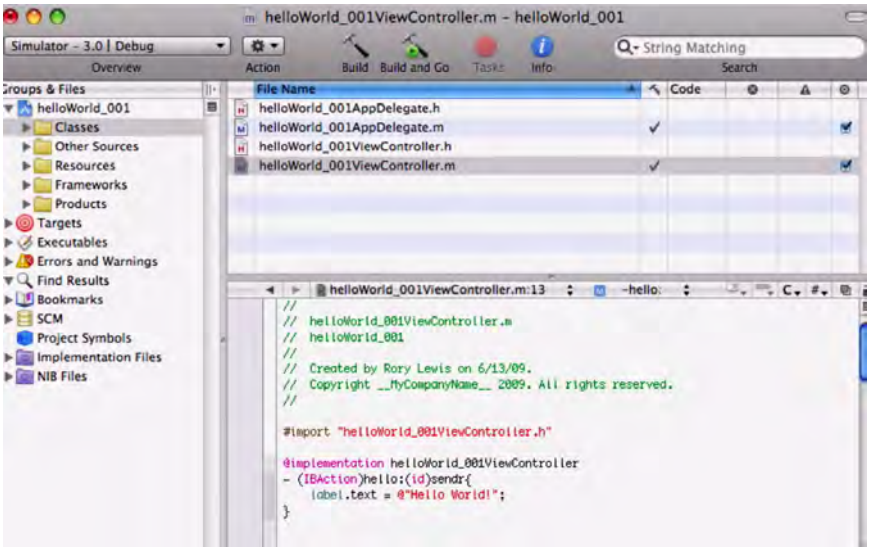


Figure 2–10. *Replace the semicolon with an "open" bracket, input your code, and then on the following line enter a "close" bracket.*

11. Now, using the shortcut Command + S, save the implementation file. See Figure 2–11.

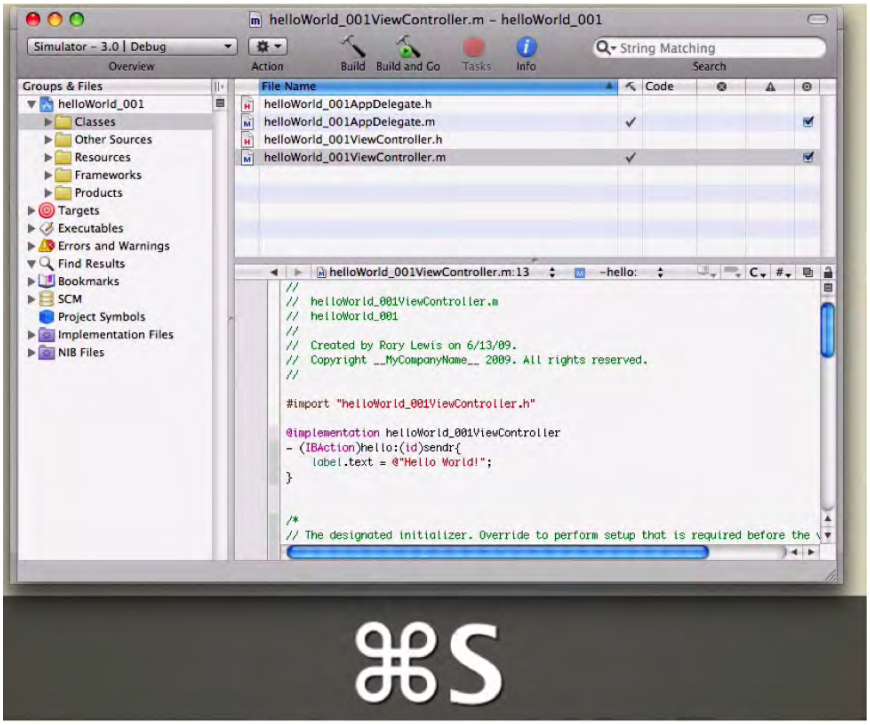


Figure 2–11. *Save your implementation file.*

12. Now, it's time to open Interface Builder. Rather than opening the application and then searching for the file we need, we will just double-click on that file to automatically launch Interface Builder. The file that we need is in the Resources folder. Once you have opened that folder, you will see the `ViewController.xib` file.

    We pronounce this as the *View Controller Nib File.* I say this because it's important to know a little jargon. When a programmer says, "Open your view controller nib file," that means you open the Resources folder and then open the `ViewController.xib` file.

    The name we gave to the project will precede the text in the `ViewController.xib` file. We named our project `helloWorld_001`, so Interface Builder will insert that before `ViewController.xib`, giving the full name of `helloWorld_001ViewController.xib`, as depicted in Figure 2–12. Now, click on `helloWorld_001ViewController.xib` to open IB.

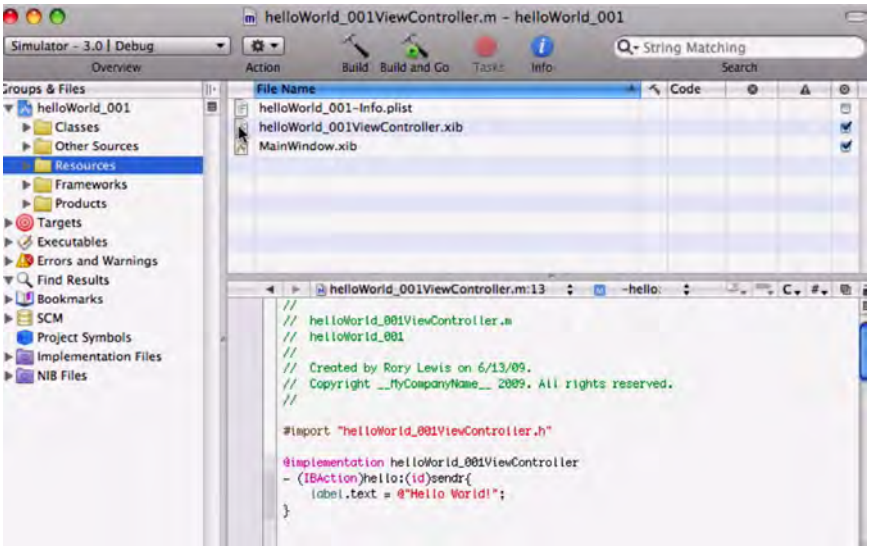


Figure 2–12. *Click on "helloWorld_001ViewController.xib" to open Interface Builder.*

13. Once Interface Builder opens, it will look something like Figure 2–13. Before I ask you to arrange your windows to resemble the placement that I have in mine, we need to first make sure that your *Library* is open. Use the shortcut ⌘⇧L (or go to Tools, then Library) to accomplish this. See if the "Library" window, probably located on the left side, is indeed on your screen.

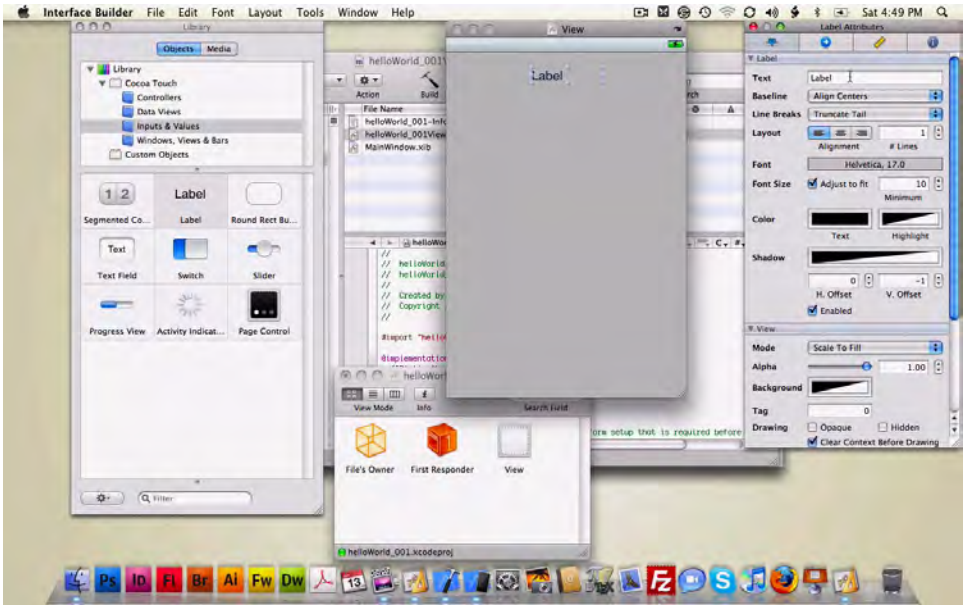


Figure 2–13. *Once Interface Builder opens, make sure that your Library window is open.*

We want to drag a label onto what the general public calls your View window. I will use the same name in this book, although it's not really accurate. If you are interested in the technically correct way of saying this, in geekspeak, you might say: let's drag a label onto our "Hello World 001 View Controller nib" file. In Figure 2–13, this file is represented by a frame (or window) that has the label "View" in its header**.**

If you like, click on the markers or boundaries of the label and make it larger by stretching it out. Then, in the Label Attributes window (to the right), delete the default text "Label" from the top field. I'm having you do this so that you can then enter "Hello World!"

Also note that you can center the text by selecting the Center button in the Layout row (four rows below the Text row).

14. Now, scan the other options and parameters that are available in this window. There are many opportunities here to tweak the look of your apps. The more attention you pay to details, the better the UI, the more sophisticated and clever you appear, and the more likely your creations are to be on the list of "Most Downloaded Apps!" Now, drag a button onto your View window. You may want to expand the frame by clicking, holding, and dragging a lower corner, as shown in Figure 2–14. We want to create more space here to accommodate more text. We want the user to press this button, which will produce a label that says "Hello World!" This means that we need to communicate to the user to press the button. So we need enough room to enter the text: "Press Me."

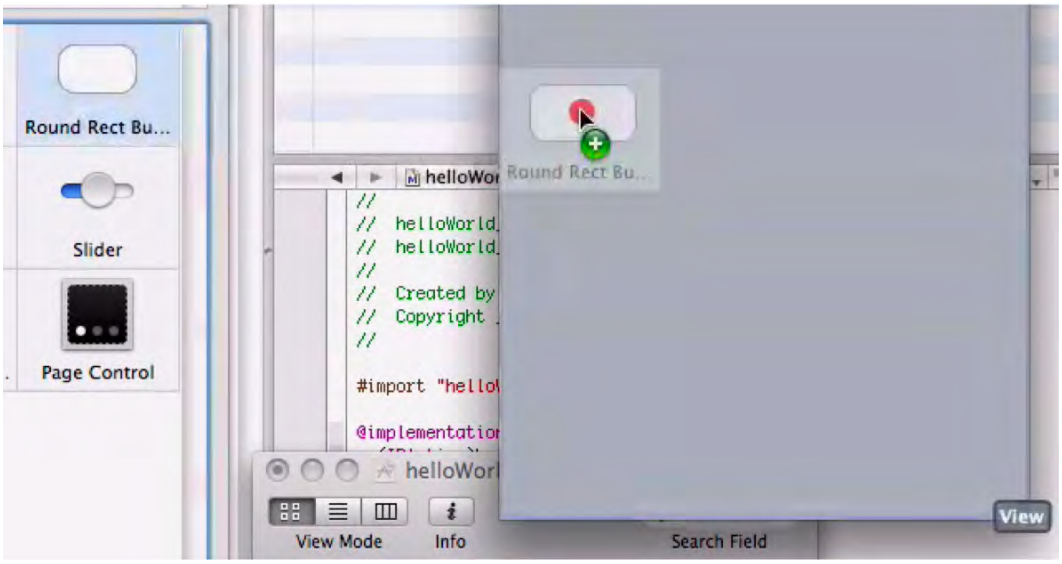


Figure 2–14. *Drag a button onto your View window.*

15. As depicted in Figure 2–15, you now want to enter the words "Press Me" into the Title field attribute of your Button Attributes window. We also want to align both the button and the label with each other and center them in context to the screen**.** We do this by selecting the Ruler tab, which controls size and alignment; it is located at the upper-right side of this window. Once it is selected, go down to the Alignment row and select the Align Vertical Centers icon (second from the right), and then go down one row and click the Align Horizontal Center in Container tab at the very right. This centers both the label and the button with each other, as well as with the interface.

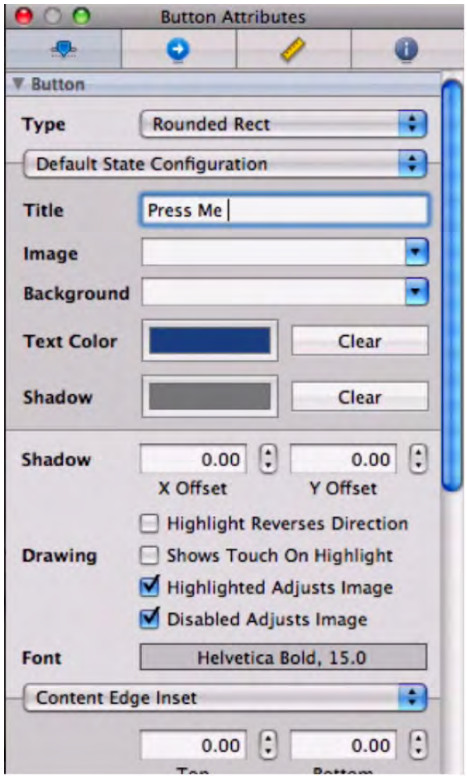


Figure 2–15. *Enter the text "Press Me" in the title field of your Button Attributes frame.*

16. Go to the `helloWorld_001ViewController.xib` window as depicted in Figure 2–16, which contains three icons: File's Owner, First Responder, and View. Go to the File's Owner icon and connect it to the label by holding down the Control key (represented by ^) and clicking the File's Owner icon. Then drag your mouse cursor to the text label in the View window, as depicted in Figure 2–16. Notice that a "fishing line" is being drawn from the File's Owner icon to your Text Label. This indicates that the connection that you desire to create is working. (If there were no line, it would mean that you had not established the connection.) As you near the vicinity of the Text Label, you will notice a black option box extend from the Text Label, containing the word "Label." Recall that, back in step 5, we referred to code that appeared as `*label`. Well, now you are seeing the mechanism to which it pointed. This is exactly where we want to connect our File's Owner—the Label. So drag that fishing line right onto the word "Label" in the black drop-down box, as depicted in Figure 2–16.

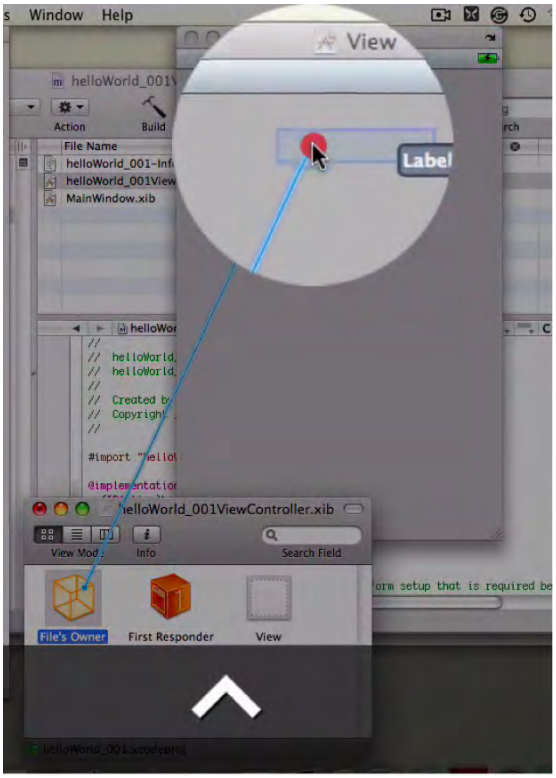


Figure 2–16. *Drag a connection from the "File's Owner" icon.*

17. Take a deep breath and think about what you just did. Let's see if we can connect some of the pieces of this puzzle. I want you to pause for a moment, and go look at four figures—carefully, and then come back to this passage.

First, go to Figure 2–5, and then go to Figure 2–10. Be thinking of the word "label." Then, jump forward slightly to Figures 2–17 and 2–18 … still thinking of "label." Okay? See you soon.

Figure 2–17. *Connect the File's Owner to your Text Label.*

Okay … finished? What did you notice?

I'm hoping that you saw signposts on the trail that are leading us to our destination, which you just glimpsed in Figure 2–18.

Do you remember how, in Step 10, you entered `{label.text = @"Hello World!";}`, and then, in Step 16, you connected the File's Owner icon to the Text Label? Do you recall how you changed that Text Label to say "Hello World!"? You're probably saying, "Okay, I kinda get that, Dr. Lewis, but you forgot about Figure 2–5!"

You began this chain of events way back in Step 5, which gave rise to Figure 2–5. You attached a deep connection between the Outlet and the Label when you entered the line `IBOutletUILabel *label;` … and the rest of the pieces fell into place from there. I know this probably didn't completely connect the dots in your head, but I'm quite satisfied if it at least connected some of them.

So, we're standing in the Forest of Objective-C, and you've seen traces of some interesting connectivity as you reviewed your path. Now that you've been debriefed, we have one more juicy detail we need to clarify and then take care of. Here's a hint … what about the button that says *Press Me*?

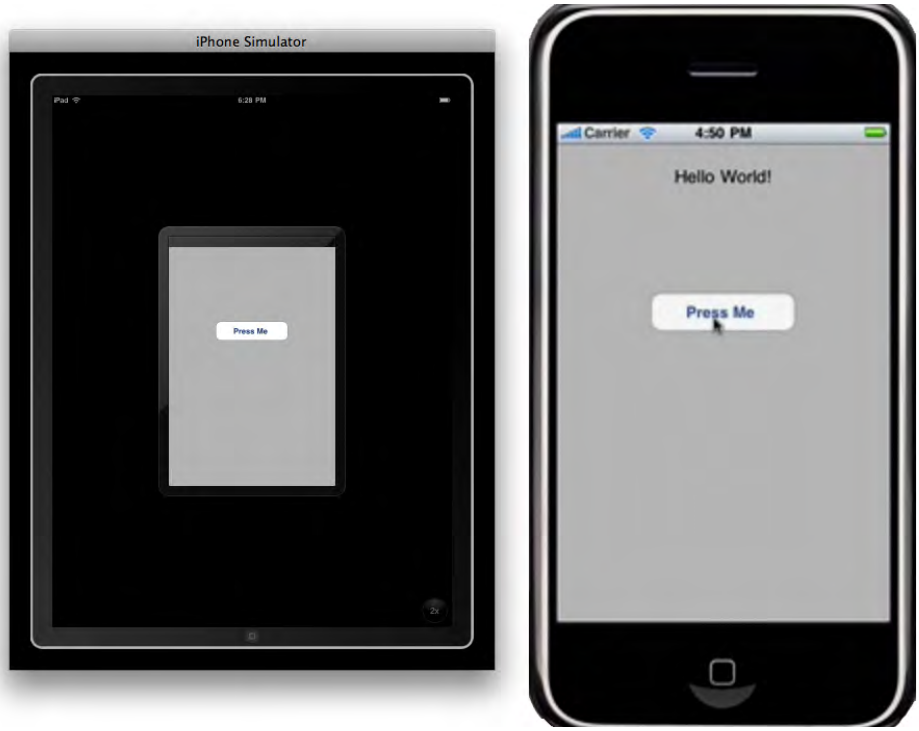


Figure 2–18. *(left) The "Press Me" button presented on the iPhone view of the iPad Simulator; (right) the result of pressing the button is displayed on the regular view - iPhone Simulator.*

18. Just as you made a connection in step 16, from the File's Owner icon to your Text Label, you now need to connect your button that has the text "Press Me" to the File's Owner. Go to the Button icon and connect it to the File's Owner by holding down Control on your keypad (represented as ^ in the screenshot) and clicking on the Button icon. Now, drag your mouse cursor to the File's Owner, as depicted in Figure 2–19. Again, you will see your "fishing line" being drawn from the Button icon to your File's Owner. This indicates that the connection is working.

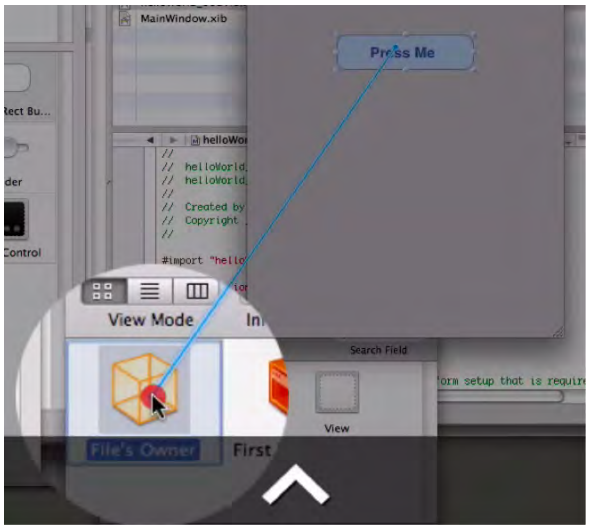


Figure 2–19. *In Interface Builder, note the connection between the File's Owner icon and the View window.*

19. As you near the vicinity of the File's Owner icon, you will notice a black option box appear with the word "hello." Do you remember, back in Step 6, when you entered `- (IBAction)hello:(id)sendr`?

Well … *guess who's comin' to dinner!* Mr. Hello is saying, "Don't you think you should pull that fishing line toward ***me***? Remember – it is I who will tell that label guy out there to say *Hello World!*"

Therefore, you should keep on dragging that line until it connects to the "hello" label in the black drop-down box, as shown in Figure 2–20.

Save your work: ⌘S.

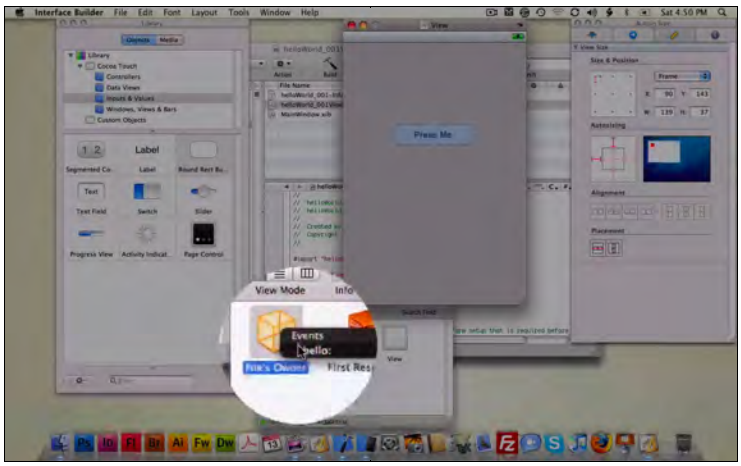


Figure 2–20. *In Interface Builder, open the "Events" drop-down list from the File's Owner icon.*

20. The final step is compiling the code. You will do this by entering ⌘↩ (Command + Run), as depicted in Figure 2–21. Your computer converts all your code to machine language, and then to ones and zeros, and then back to a language that makes sense on your iPhone and iPad Simulator … and, of course, your iPhone and iPad.

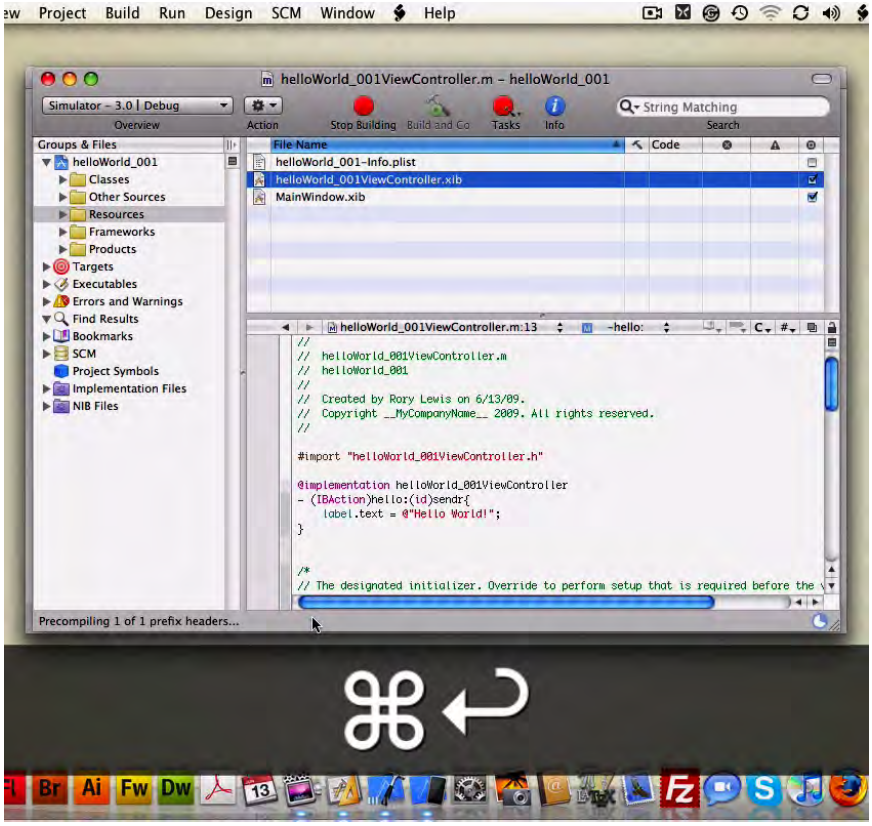


Figure 2–21. *Return to Xcode, and use the indicated shortcut to compile your code.*

After compiling is complete, test your app. Push the “Press Me” button, and you will see your result. Figure 2–22 shows the iPad Simulator in full (2x) view mode, before and after pushing the button.

You have just completed your first iPhone and iPad application. ***Congratulations!***

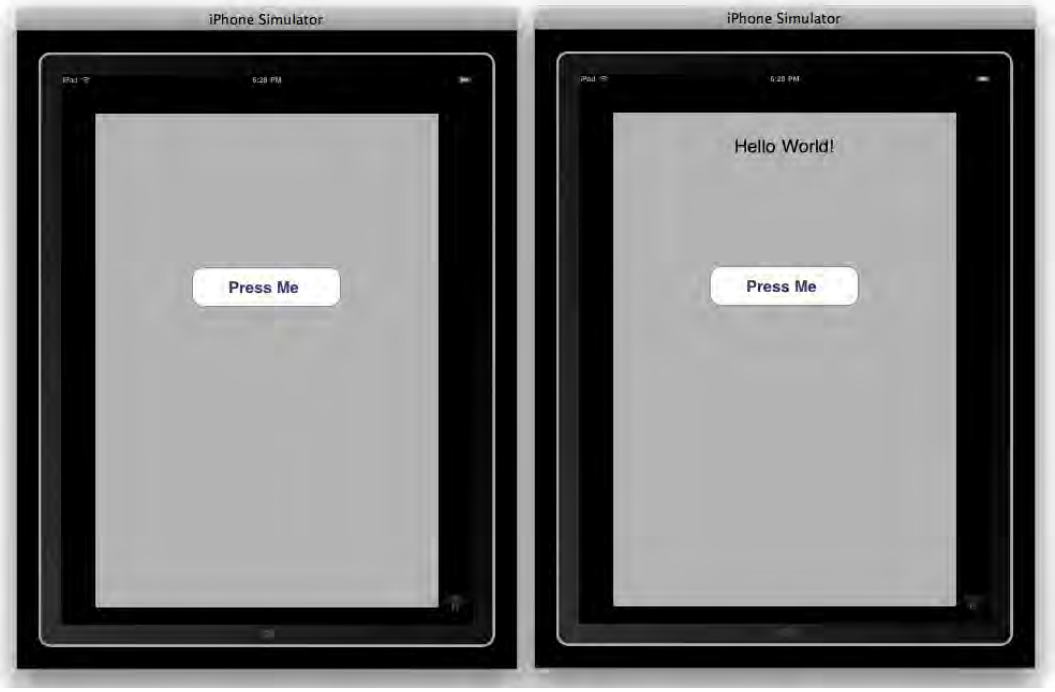


Figure 2–22. (*left) Beautiful button in the iPad full screen mode; (right) Click it … Hello World!*

helloWorld_002 – a Navigation-based Application

In your first program, "`helloWorld_001`," you said "Hello" to the world from the View-based Application template in Xcode. Now, it's time for your second example, `helloWorld_002`, which will be created via the Navigation-based Application template in Xcode.

Before we get started with the next method, you need to save `helloWorld_001` in a folder of your choice that is *not* on the desktop. Create a folder in your Documents folder called *My Programs*, and then save the file named `helloWorld_001` there by dragging it to that folder. Now, with a fresh, clean empty desktop, close all programs. Press Command + Tab and then Command + Q to close everything until only the Finder is left on your screen.

Now, just as you did in the first example, launch Xcode and open a new project by using your keyboard shortcut: ⌘⇧N. Your screen should show the New Project wizard as depicted in Figure 2–23. You may, however, find that your View-based Application template was highlighted by default, because of the last example. Now, though, click on the Navigation-based Application icon, and save the new file to your desktop as "`helloWorld_002`."

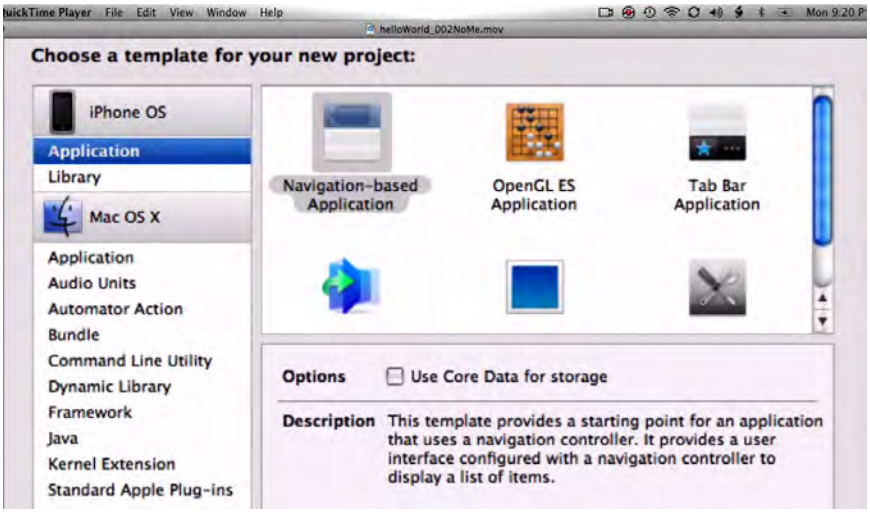


Figure 2–23. *Open Xcode, select the Navigation-based Application template, and then save a new project file to your desktop.*

1. Once you save this project to your desktop, Xcode instantiates a file named `helloWorld_002` with a group called Classes, just as you saw in the first example … see Figure 2–24. In this navigation-based template, we see that it consists of two pairs of subfiles: an AppDelegate header and main, along with a RootViewController header and main. Then, when you click on the `RootViewController.h` (header) file, you will see it has what we call a subclass of UITableViewController, which handles all the cool things necessary for displaying tables in an iPhone/iPad app. Tables are actually used much more than most users probably realize. Because we use tables for so many apps, I thought you should also learn how to say "Hello World!" in this context. In fact, right now, we have everything we need to call up a blank table.

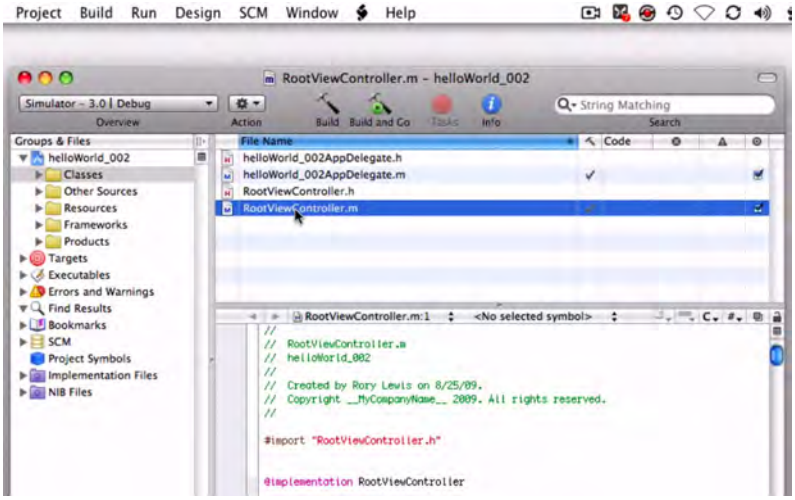


Figure 2–24. *Open the Classes folder and click on the RootViewController.m file to examine the pre-existing boilerplate code.*

Recall that, at the end of the first example (Figure 2–22), we compiled our code by entering ⌘↩. We saw how the computer converted all that code you'd entered into a working application. Well, we're going to do that again, right here, without having entered a single line of code.

Within Xcode, go ahead and enter ⌘↩. Refer to Figure 2–25.

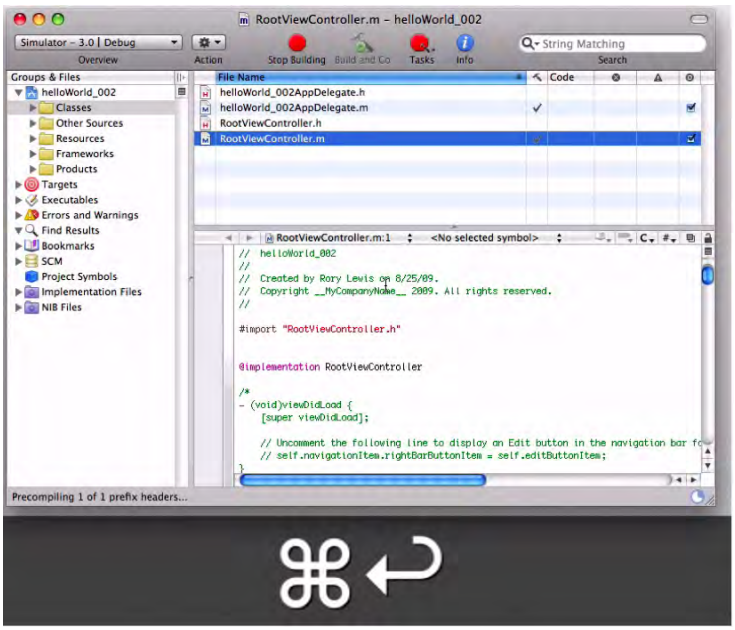


Figure 2–25. *Within Xcode, we use the shortcut to compile and run our program—even though we haven't entered a single line of code!*

As you can see in Figure 2–26, compiling a generic bit of ready-made code may be possible, but it lacks anything specific. The table view is laid out – but, of course, the computer has no idea what we want it to say or display. We must tell it!

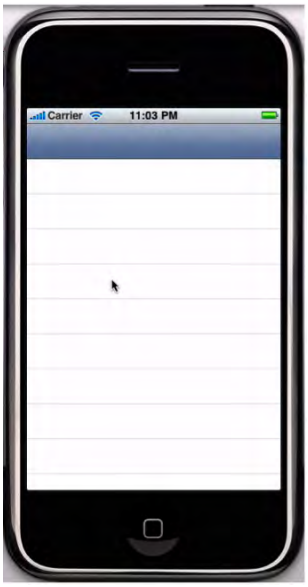


Figure 2–26. *Running the program calls up the iPhone and iPad Simulator, from which we see a perfectly formed Table with absolutely nothing in it!*

2. The iPhone/iPad Simulator has some powerful structures that have been prepared ahead of time, and we are seeing one of the typical projections via the Navigation-based Application template. What do you think is causing this to happen?

   The answer: the `UITableViewController`. This contains code that we will eventually examine, but right now just file it under unsolved mysteries.

   There is something we need to accomplish here, though. We want to use one row of the table to say "Hello World!" Let's do it.

3. First, quit the iPhone/iPad Simulator by entering ⌘Q. This will take you back into Xcode to the RootViewController implementation file, which is where you will enter your code (see Figure 2–27).

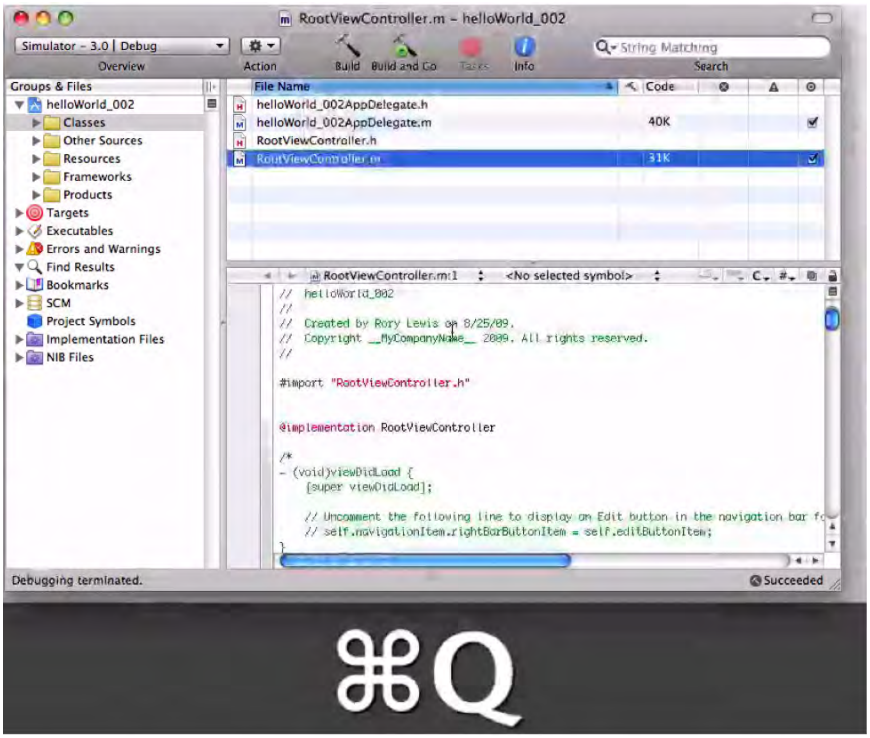


Figure 2–27. *By using the indicated shortcut, we exit the Simulator and go back into Xcode, so that we may begin entering our new code.*

When you first open up `RootViewController.m`, there are a lot of green statements. Some of these comments refer to *canned functions*, which the guys at Apple have kindly programmed, knowing that, at times, these will come in handy for us programmers. We won't be using any yet, but later on we will.

One thing we will do is go to the section that has to do with the number of rows in our table. You will make a simple change there to achieve our objective in this exercise. Refer to Figure 2–28.

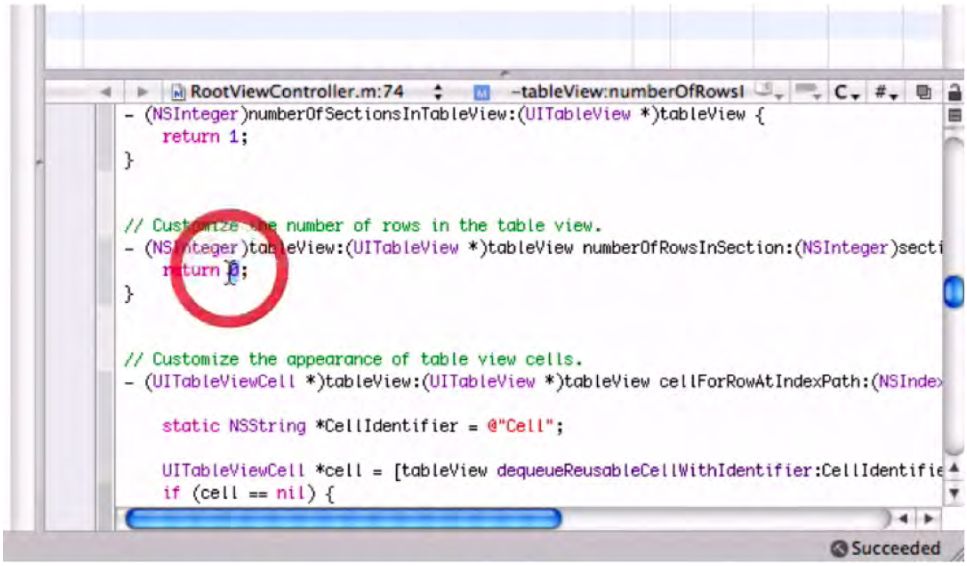


Figure 2–28. *The red circle highlights the place where you change the default value 0 to 1.*

4. Scroll down in the `RootViewController.m` file until we get to the code that determines how many rows we will have in our app's table. In this simple app, we only need to have one line of output, so you will have to ask it to return not zero, which is the default number of rows, but one. The red circle in Figure 2–28 indicates the place to make this change.

   So, delete the 0, and replace it with 1.

   Change this:

```
// Customize the number of rows in the table view.
    -   (NSInteger)tableView☹UITableView
        *)tableViewnumberOfRowsInSection:(NSInteger)section {
return 0;
}
```

to this:

```
    -   // Customize the number of rows in the table view-
        (NSInteger)tableView:(UITableView
        *)tableViewnumberOfRowsInSection:(NSInteger)section {
return 1;
}
```

5. Now, I want you to add a little code that will change the properties of that one row we want to manipulate. We want to say "Hello World!" in that one row. Below the line on which you entered `return 1` in the previous step, we see a green commented section: "`Customize the appearance of table view cells.`"

Scroll down to where it says, “Configure the cell,” as shown in Figure 2–29. Highlight that line and delete it.

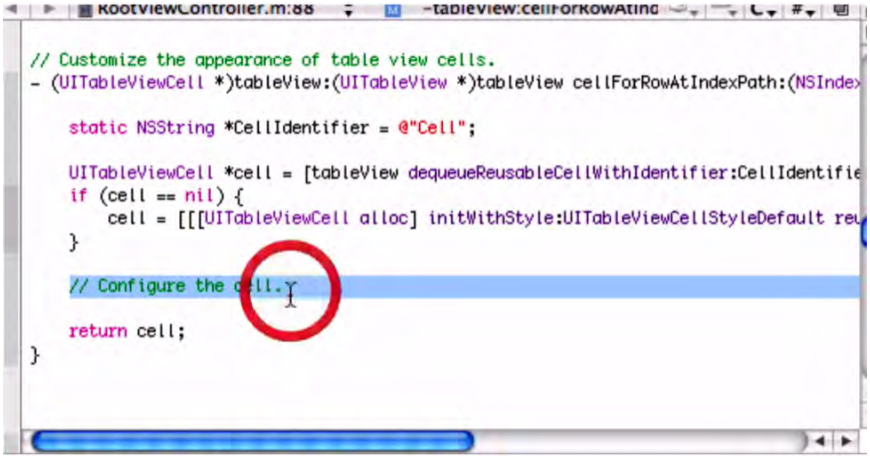


Figure 2–29. *Delete this line of default code so that you can insert the desired command.*

6. Now, in that same place, I want you to type:

```
[cell.textLabelsetText”@"Hello World!"];
```

This command says we need a cell that has the Apple code required to insert text into a label. Furthermore, we want the words “Hello World!” to populate the cell. Here is the full code:

```
-   // Customize the appearance of table view cells- (UITableViewCell
    *)tableView:(UITableView *)tableViewcellForRowAtIndexPath:(NSIndexPath
    *)indexPath {

staticNSString *CellIdentifier =”@"Ce”l";

UITableViewCell *cell = [tableViewdequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
cell = [[[UITableViewCellalloc]
initWithStyle:UITableViewCellStyleDefaultreuseIdentifier:CellIdentifier] autorelease];
    }

      [cell.textLabelsetText”@"Hello World!"];
return cell;
```

You will be prompted to save your work upon entering the shortcut for compiling and running your new code (⌘↩). Refer to Figure 2–30. Hit Return.

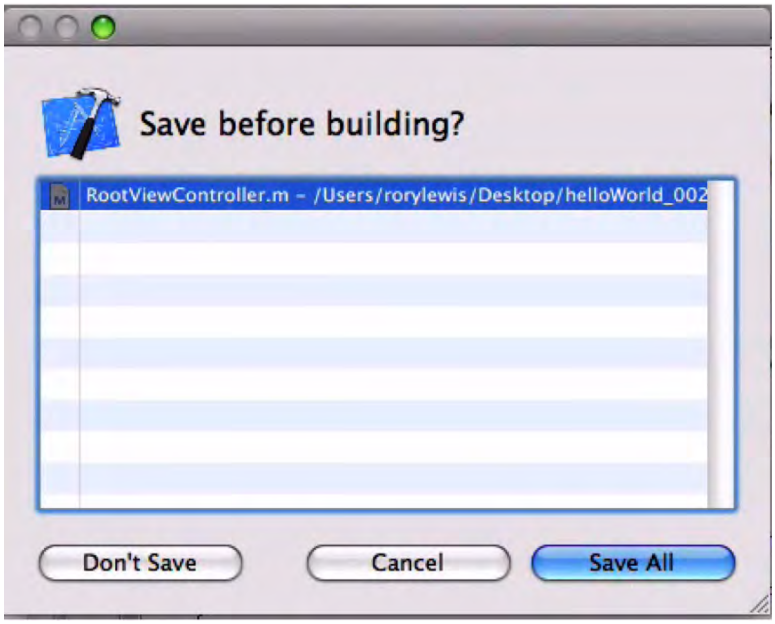


Figure 2–30. *Upon entering the shortcut to* ***compile*** *the new code, you will be prompted to save your work.*

7. Once this crucial step is complete, we will see a table that contains one row with the desired text: "Hello World!" See Figures 2–31 to 2–34.

Great job! You have now written your *second* iPhone and iPad application.

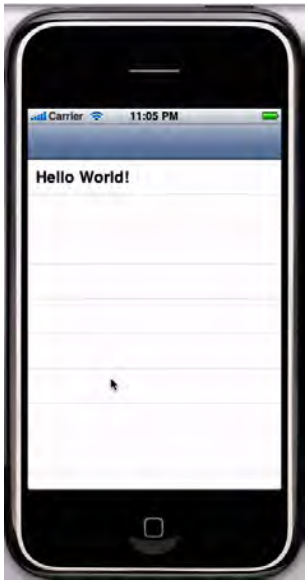


Figure 2–31. *Voilà … success!*

Figure 2–32. *Our result is a simple and handsome table view with "Hello World!" on the top row, seen in iPad's embedded iPhone View.*

Figure 2–33. *Click on "Hello World!" and the top row becomes active.*

Figure 2–34. *Here we see the top row highlighted and active, as seen in iPad's normal full screen mode.*

helloWorld_003 – Modifying a Navigation-based App

In our final example for this chapter, you will repeat the steps you took for the previous app, helloWorld_002, and make some simple modifications. Select the Navigation-based Application template once again, and name your project helloWorld_003.

1. Scroll down in the `RootViewController.m` file, and, just as you did for helloWorld_002, enter this line of code:

```
[cell.textLabelsetText:@"Hello World!"];
```

At this point, the complete RootView Controller file looks like this:

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier] autorelease];
    }
        [cell.textLabel setText:@"Hello World!"];
    return cell;
}
```

2. Now, I want you to center the text. To accomplish this, you can either type the following code directly, or you can copy and paste the line you just inserted *above* itself … and then edit it to appear as follows:

```
[[cell textLabel] setTextAlignment:UITextAlignmentCenter];
```

   The file should now appear as:

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier] autorelease];
    }

        [[cell textLabel] setTextAlignment:UITextAlignmentCenter];
        [cell.textLabel setText:@"Hello World!"];
    return cell;
}
```

   In essence, this command is calling out, "Hey, cell with text label, `[cell textLabel]`, listen to the directions being issued by your superior, `setTextAlignment`, and obey the User Interface text alignment command we've just given it."

   To our output row of text it means, "Center it up, buddy!"

3. But hold on, we're not done yet. For this little exercise, I also want you to make the text editable, to a degree. We'll put up an Edit button that, when pressed, will give the user the ability to delete our "Hello World!" text.

   How do we do this? Back in step 4 of example 2, I mentioned that there are many functions that we programmers are, at times, not using. Well, this is one time that we will dip into that area to make a change.

I introduced the idea of comments earlier, telling you that these are in green and that they include or introduce information or functions that are invisible to the processor. Earlier, the comments we considered were denoted by the double slash symbol, //. There is another style of comments that is bound by /* at the beginning of the content and */ at the end of the content.

Go toward the top and you will see the code saying "Uncomment the following line to display an Edit button in the navigation bar …" It looks like this:

```
/*
- (void)viewDidLoad {
    [super viewDidLoad];

    // Uncomment the following line to display an Edit button in the navigation bar for
this view controller.
    // self.navigationItem.rightBarButtonItem = self.editButtonItem;
}
*/
```

4. You need to delete the two comment markers: /* and */, and then delete the line comments starting with the // so the section now appears like this:

```
- (void)viewDidLoad {
    [super viewDidLoad];

    }
```

Excellent! Now, let's go through the remaining steps similar to the way you proceeded in helloWorld_002, starting with saving your work on the RootView Controller file. Using your shortcuts, save and compile the code.

Congratulations! You have just finished your third iPhone/iPad app!

Figure 2–35 reveals the crisp results of your hard work.

Figure 2–35. *With these simple modifications, you get a nice clean table view with centered text on the top row, along with an Edit button.*

As simple as this is, consider the mechanisms that have been pre-coded to allow this functionality, which you have unleashed by making a few basic changes to the file. Clicking on the Edit button triggers the display of the red and white button; it also causes the text on the original button to read “Done.” See figure 2–36.

Figure 2–36. *Clicking the Edit button activates the "Delete" command, represented by the red and white button. The button now reads Done.*

Chapter 3

What's Next?

Now that you've gotten your feet wet, programming your first three iPhone and iPad apps, I want you to ask yourself: *Where do I go from here?* The answer to that question is what this chapter is all about.

I intentionally held off on this orientation until you had a little programming under your belt. With the taste of the code fresh in your mind, you are now poised to appreciate the sequence and the variety of challenges ahead. Read on!

Figure 3–1. *The author with iPhone and iPad Apps for Absolute Beginners displayed on both devices.*

To help you get better oriented, I have divided this overview chapter into three sections. In Section One (§I), I explain why I have chosen the road map set forth in Chapters 4 through 9, and I give you a brief description of each chapter.

In Section Two (§II), I discuss the relationship between the iPhone and the iPad (Figure 3–1), and how this may affect your approach to the upcoming material and related exercises, perhaps giving you a new perspective on your own creative priorities and objectives.

In Section Three (§III), we venture under the hood of the hardware, and we consider some deeper structures implied by the code that runs these devices. It isn't imperative that you read either §II or §III immediately, but I suggest you at least scan §III—even if you don't think you're absorbing it all. Just let your eyes flow over the text. You'll probably find that you benefit from a quick glance back here from time to time.

§I: THE ROAD AHEAD

I based the iPhone/iPad class that I run at the University of Colorado on the six most common components of all iPhone/iPad apps: navigation, actions (with single and multiple outlets), switching views, touches and gestures, tables, and maps. Here is the schedule of how these elements are addressed.

Chapter 4: An Introduction to the Code

Buttons & Graphics

Chapter 5: Buttons & Labels with Multiple Graphics

IBOutlets & UILabels

Chapter 6: Switch View with Multiple Graphics

Three Different Approaches

Chapter 7: Dragging, Rotating & Scaling

Touches, Gestures, and Events

Chapter 8: Table Views, Navigation, and Arrays

Designing a Series of Linked Tables

Chapter 9: MapKit

Annotations, Map Views & Map Controls

Introducing Chapter 4—*An Introduction to the Code*

In Chapter 4, I will have you do what I have my regular students do: help me learn your name. You are going to manipulate a picture of yourself to work with and learn about buttons and graphics. If you don't have a picture of yourself, you are welcome to download the picture I use in the example (Figure 3–2). By the time you are done, we will have moved from the "Hello World!" stage of coding to the placing of buttons that control graphics. In this chapter, we will use buttons and graphics that interact with one view only.

Our work with buttons and labels will stir up some juicy questions, and I will do my best to address the more pressing ones. For instance, the geeky guys who developed Cocoa Touch, the underlying code that controls how humans interact with the iPhone and the iPad, used a concept they called the Model-View-Controller (MVC). An essential part of this model is the code that makes up the Graphic User Interface (GUI, pronounced "gooey"). The GUI is *the way* that users relate to, talk to, and communicate with the computer. We delve into the MVC components in the third section of the present chapter, §III: A Look Under the Hood.

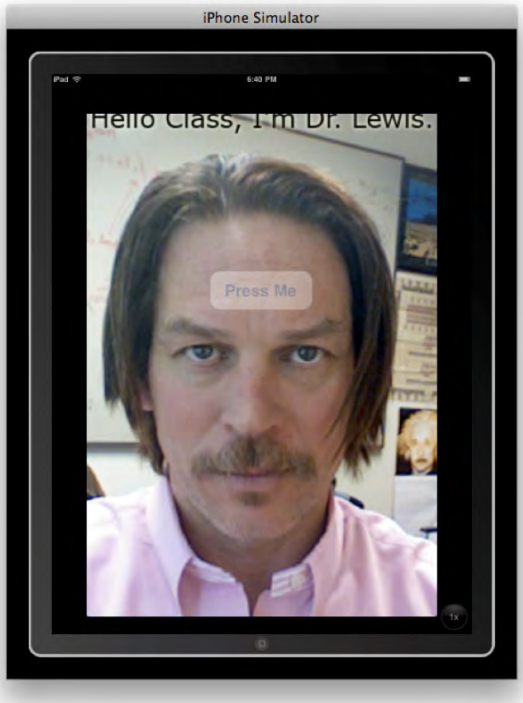


Figure 3–2. *Hello, I'm Dr. Lewis.*

Introducing Chapter 5—*Buttons & Labels with Multiple Graphics*

Buttons and graphics that interact with one view are fun, but how do we have a button interact with more than one view? It's really not about making a graphic program interact with multiple views; it's about configuring your program, whatever it is (a map, a game, code for calculations), to efficiently and effortlessly jump between various views. This concept is important for all your future programming, and your experience in this chapter will serve you well for the remainder of the book.

Chapter 5 is not only a step up in terms of the complexity of coding, but we will also transition to more technical language. Don't worry, though—you'll be OK! Everything is laid out with discrete steps and humorous analogies. You will see that we perform a little visual sleight of hand by taking a picture of a scene and then superimposing another picture onto it (see Figure 3–3). This technique is used all the time in games, and some of the related ideas will be touched on in our discussion of INDIO at the end of *this* chapter.

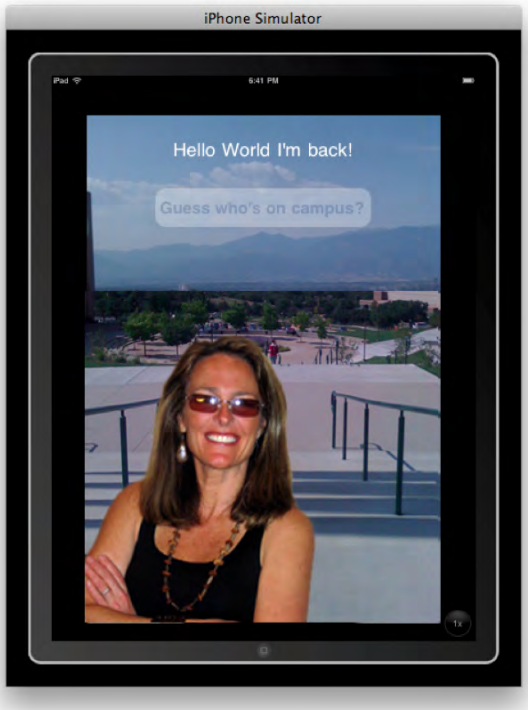


Figure 3–3. *The iPad image of the overlay of views: "Hello World, I'm back!"*

Introducing Chapter 6—*Switch View with Multiple Graphics*

This is the longest chapter in the book because we work through three examples that do the same thing, but in three different ways. Chapter 6 is all about tabs and switching views with tabs—something that is standard in nearly every iPhone and iPad app I've ever seen (see Figure 3–4). In the first example, we code the graphics the long way. In the second example, we see how the no-brainer way works so easily—yet how it also fails to educate us or let us experience the code underlying tabs and switching views. Thus, we utilize the third example to compose a hybrid of these two approaches; you gain the experience and stamina of one method and the time-saving shortcuts of the other.

Your success in Chapter 6 will advance you from the novice stage to that of a legitimate programmer-in-progress. By tackling some difficult coding issues, you will have earned respect in the iPhone/iPad programming community. The good news is that you will be ready and able for the next step, for you will have followed the path I have prepared on the basis of my previous teaching experience.

A big part of my philosophy is that we prepare ourselves for future challenges by mentally stretching and visualizing. This requires or involves organizing our abstractions into pieces we can manipulate and repurpose, in whatever new context comes along. This is a flexibility dance! So, begin now to breathe into these future exercises and challenges, and to appreciate that they are not just about displaying random images at the push of a button.

It's vital that you see that each picture stands for a different part of your code, which gives the user of your app access to a new view, a new set of options, or a new level of the program. We use pictures at this part of the learning process because it's easier than code. It also makes troubleshooting far easier; if the image doesn't do what it's supposed to do, you know that your "switching" code is the problem.

Be prepared to be catapulted into the programming stratosphere!

Figure 3–4. *Seamlessly switching in the iPad's full view. Not a single line of code!*

Introducing Chapter 7—*Dragging, Rotating, and Scaling*

Leveraging the power of Multi-Touch on the iPhone and iPad can be pretty awesome, and that is what we begin to learn in this chapter. Creating an image that will rotate, scale, and move with an intuitive flick of your fingers will confirm just how powerful these devices are. (see Figure 3–5.)

Think about it. Before the iPad or iPhone, we used a mouse and the keyboard. Now it's all about touching. There is an old acronym that some of you probably know: WYSIWYG. It stands for "what you see is what you get," when a word processing document is printed the way you expect. Now, in the Apple world of Multi-Touch, we see a new paradigm forming— one that reflects that touching the screen in certain ways accomplishes some natural and logical task in an easy and intuitive manner. The computer will say to you, as the genie said to Aladdin, "Your touch is my command!"

In Chapter 7, I will introduce you to the code that interacts with the different kinds of touches and gestures users will employ when they run your app. If you are like my past students, you will delight in learning the grammar that directs the images to move according to our touches, swipes, pinches, drags, etc.

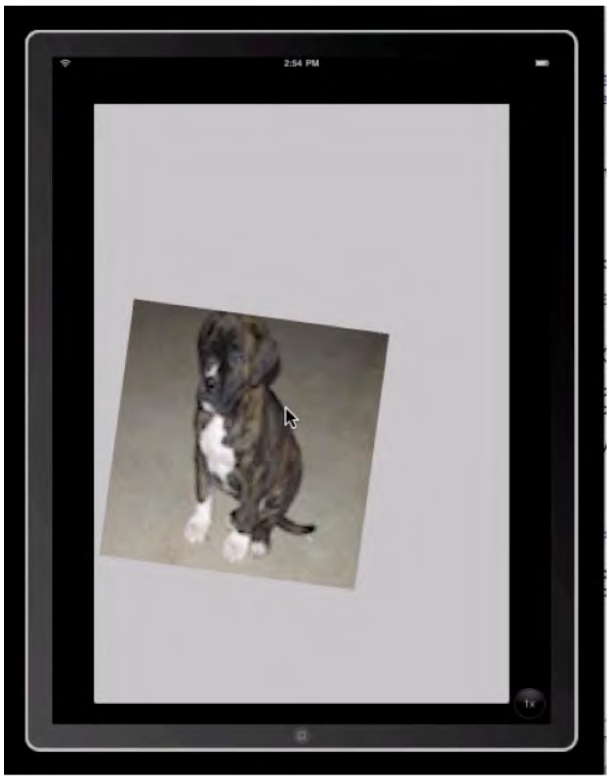


Figure 3–5. *The examples work on both the iPhone and iPad.*

Introducing Chapter 8—*Table Views, Navigation, and Arrays*

Table Views and navigation are essential elements in the vast majority of apps. Ever wonder how the App Store makes those really cool lists of apps? Ever need to keep a list of items on hand, but you aren't sure which application to use? Worry no more! You will be creating a table view application capable of switching among multiple views (see Figure 3–6). Can't find what you want on the App Store? Make it for yourself—*and others!*

While this all sounds warm and fuzzy, there is a problem. To get beyond the essentials of table views and navigation, you will need to get your head wrapped around a notorious programming beast: the array.

Due to the degree of difficulty with this programming concept, it is generally conceded that arrays should never be taught to beginners. I was not going to include this chapter in the book, but because so many of my students wanted to utilize tables in their final projects, I had to rethink my strategy. I decided to teach my students, and you readers of this book, how to work with and control arrays with very little understanding as to how they actually work.

Therefore, I want you to consider Chapter 8 as an *optional* chapter. When we get there, I will walk you through the process of deciding whether you even want to learn arrays, and then I actually urge you to go directly to Chapter 9. Let's cross that bridge when we come to it.

Figure 3–6. *Table views created in Chapter 8.*

Introducing Chapter 9—*MapKit*

Chapter 9 is my favorite, and I predict you will find it the most enjoyable as well. By the time you get there, you'll know the basics well enough to start enjoying MapKit—the code we use to interact with maps on the iPhone and iPad. We'll find our location in the world and realize how tiny but how very smart we are (Figure 3–7). Creating our own annotations, map views, and map controls are just a few of the delights in store!

I will also take you on a tour of some excellent existing apps that use the mapping functions, in the hopes of inspiring you and stirring up ideas for future creations. In that vein, I will share some of my students' projects that vividly demonstrate how quickly and impressively it is possible to apply the knowledge acquired in this course to produce apps with a decent level of sophistication.

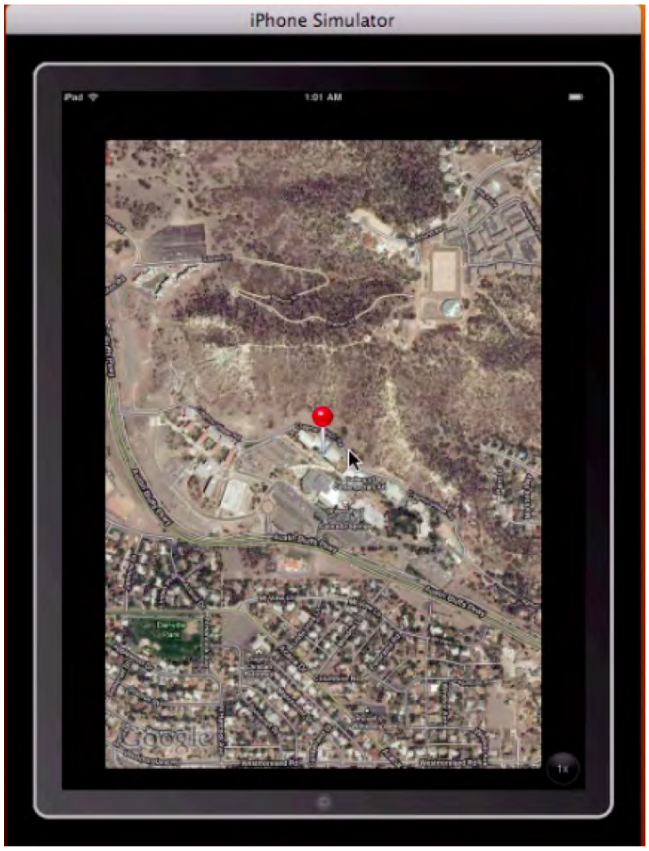


Figure 3–7. *The MapKit app we'll build in Chapter 9.*

§II: THE iPHONE AND THE iPAD

Right off the bat, let's understand what the iPad is all about. In fact, let's think about what all the Apple geeks were doing after the iPad was launched. Within the first 24

hours of the iPad's release, chances are many of the Mac-heads were scratching their heads and complaining. "What's in it for me? There's nothing new here. This really sucks!" This crowd was tuned into the fact that, with their Macs and their iPhones, they had everything they needed already, and many of them quickly concluded that the iPad wouldn't be serving any immediate need.

Well, then, who *is* the target audience for the iPad?

Knowing why the iPad was developed, and *for whom*, will help you understand which kind and style of apps have the greatest chance at succeeding. I spent close to two years following the events that led up to the introduction of the iPad—from initial speculation, to whether it would be called the iTablet or the iSlate, to mid-release reports and news updates—everything.

> **NOTE:** The target market of the iPad is NOT the Mac user. Nor is the target market the average iPhone user.

Believe it or not, the iPad design is aimed at older users who are new to the computer world and who don't want to spend more than $1,000. These are people who have just figured out how to turn the computer on and off and use e-mail! The iPad is also targeted at young students in an attempt to outdo the Kindle. One of the iPad's primary goals is to make the purchase of textbooks obsolete by offering a digital bookstore, in the same manner that iTunes offers music. So there you have it: older and younger generations of users are the target markets, instead of the standard set of geeks that normally goes for high-tech devices.

In this book, I present you with the knowledge and techniques to create applications that work with the iPhone *and* the iPad. The app images in this book include the iPad Simulator, which inherently implies the iPhone Simulator. Be sure to check out the various figures that contain images of results, and study the differences between the iPad and the iPhone. It will be important to know how the iPad *does* differ from the iPhone.

Do Apps Run on Both the iPad *and* the iPhone?

All the apps illustrated in this book run on both the iPad and the iPhone. I know this because I personally tested them—not only on the simulators, but also on my actual devices. Some of the images could have been made with larger (higher resolution) files had they been of apps coded exclusively for the iPad, but I chose to keep everything compatible with both devices.

The best way to tell if a future app you create works for both the iPhone and iPad is to simply plug in your iPad to your Mac; iTunes will display the apps that work on the iPad (Figure 3–8).

Figure 3–8. *This shows the author's iPhone apps that iTunes says are compatible with iPad.*

> **NOTE:** Some of the apps featured on the iPad—in the normal view—look a little fuzzy; this is because the images are being stretched beyond their optimal 100% size. For those of you interested in programming apps *exclusively* for the iPad, you will need to make sure your image resolutions stay at or below 100%.

More Screen Space

The physical dimensions of the iPad are a little more than twice the height of the iPhone, and slightly more than triple the width, but the depth is increased by only 1.1 millimeters! The result is a roughly six-fold increase in the surface area, at practically the same depth, so this is truly a much bigger device that still possesses an extremely slender and light body. The physical dimensions are not, however, the same as the display dimensions.

> **NOTE:** The *iPhone* display contains 320 x 480 pixels; the *iPad* display contains 768 x 1024 pixels. This gives the *iPad* a relative increase in usable screen space over the *iPhone* of just over five times.

This extra space has very important implications. Most obviously, quite a bit more information can be presented to the user at any given time. This means the

organizational mindset that programmers are using on their existing iPhone apps may not always translate directly to the new, relatively expansive platform. Also, it turns out that migration from iPhone to iPad involves classes—preprogrammed sub-routines—that are not shared between the devices, and accomplishing this may require some specialization inside applications. Handling the user interface changes and application programming interface (API) differences will be addressed a little later. Right now let's consider the implications of the increased real estate the iPad boasts.

The iPad has the ability to run iPhone applications without changing the code one iota. Moreover, the iPad can run these applications at double their original size using a magnification function. This makes the migration process nearly painless for current iPhone developers. However, we are dealing with much more here than an oversized iPhone! There are numerous legitimate improvements over the iPhone that went into this newer device—not just incremental change, but quantum change. Because of this, we want to leverage these improvements and knock the socks off our future users.

More screen space means more information and more eye candy. A user can now see multiple pieces of data at once without having to navigate through the application. More room for videos, pictures, text, or game graphics means more flexibility and longevity for your app. The iPad structure supports all of the same features of the iPhone, including UIKit, Core Graphics, Media Kit, and OpenGL ES. Simply put, these are tool kits from which a bunch of different applications can be built. The increased processing power and larger screen size, however, mean that the iPad—and therefore your apps—can do significantly more with the same set of tools.

This increase in screen area, with improved high-resolution multi-touch sensors, also means more space for the user to interact with your application—with finer articulation. Complex gestures can now be used to navigate and manipulate an application in ways that are impossible on the relatively small iPhone screen. Keep this in mind as your application begins to take shape.

> **NOTE:** Users can now use their entire hand, or even both hands, to interact with elements in an application.

What gestures make sense here? What kinds of input could trigger *this* or *that*? In addition to the increased ability to support input from key gestures that stand for shortcut commands, the large screen size makes toolbars and accessory views possible. Providing the user with a toolbar is no longer taboo, and this may actually be encouraged to help your users leverage the increased interface of the iPad.

When you take all of these changes collectively, it behooves us to carefully consider our applications' designs. The creativity you and your fellow programmers wield must not only take advantage of the ability to display greater quantities of information in an easy-to-read, intuitive format, it must border on a total reorganization and transformation of the user experience.

Master-Detail

One of the central features of the iPad, not found on the iPhone, is the *master–detail* interface. It's been around for a while on Mac operating systems, and now it has blossomed in a new form. Master-detail is an interface layer that sits on top of a list or a table and acts as an intermediate stage in place of a "Go back" or "Go to" button. On the iPhone, either the master or the detail can be seen, one at a time, but never both at the same time. This dual-purpose layer can serve the user in the e-mail realm, for instance, with the master as inbox and the detail as individual letter, or in countless other contexts as a table of contents and selected data. As you can see in this image from Apple (Figure 3–9), the iPad presents the master-detail in either orientation.

Figure 3–9. *Note how the master-detail sits on top of an individual e-mail in the portrait orientation, but side-by-side in the landscape orientation.*

User Interface on iPad

Some applications will scale up from the iPhone without problems. However, these applications are likely to be in the minority, as the input capabilities and display space of the iPad completely change an application's fundamental character. To accommodate this, we need to bring some specialized user interface design needs into the picture in order to present your application on both platforms.

> **NOTE:** Be aware that although the iPad can enlarge images on an iPhone, my testing of iPad apps (as of April 3, 2010) showed that large images would *not* convert seamlessly to an iPhone application.

The new interface elements introduced with the iPad allow for highly stylized and powerful data flow and interaction within an application. The smart folks at Apple have done most of the work for you, providing incredibly rich and easy-to-use interface elements that let you focus on the functionality of your application.

Checking the Platform

That's all fine and good, but obviously there are some things that the iPad can do that the iPhone cannot—and vice versa. We already know there are additional classes the iPad uses that the iPhone will not recognize. So what must we do to make our application behave correctly depending on which device it is currently running? This is, unfortunately, not as straightforward as one would hope, but it can be done.

For this reason, we are not going to worry about checking the current platform. Instead, we will just change the nib files to do what we need them to do on the iPad. Don't worry—nibs and Interface Builder work exactly the same way for the iPad as for iPhone.

However, if you absolutely must know, there are a variety of techniques used to check the current platform of an application. These all involve checking for the existence of platform specific classes, functions, or features. For example, you could check the width of the screen to determine the platform. An iPhone app will return a screen width of 320 pixels, while an app written for the iPad will return a screen width of 768 pixels. Clearly, if the returned screen width is less than 768, we know we are working on an iPhone or, perhaps, an iPod Touch. Of course, there are other ways of checking the current platform. You can also check a function called `UI_INTERFACE_IDIOM()`.This function can return either `UIInterfaceIdiomPhone` or `UIInterfaceIdiomPad`. These examples are pretty self-explanatory, I hope, and can be used to accurately determine the device. Of the many different techniques to determine the current platform, you will learn to use the one that is the most convenient in the given context.

> **NOTE:** As stated previously, each device can do certain things the other can't. The iPad can't make phone calls, take photographs, or record video. Case by case thought should be applied to apps regarding location-aware services and accelerometer functions—for although both devices are portable and shakable, size may matter! ☺

Compatibility

The release of iPhone OS 3.2 has changed a lot of basic functionality we are all used to on our iPhones and iPod Touches. However, not everyone will be using 3.2, so we need to make sure our applications are compatible with previous versions, specifically 3.1.2 and 3.1.3. Fortunately, doing so is quite simple. In your new iPhone OS 3.2 project, you need only set the "Base SDK" in your project settings to 3.1.2 or 3.1.3 and run your application.

NOTE: As we all know, the downside of rapid technological advance is that we must always be prepared for a new upgrade to come along. Thus, regardless of present compatibility, an upgrade may be needed six months hence. The instructions here should still work by setting the SDK version to the latest one.

If you want to run your application in the iPad Simulator to see what it looks like, click on the pop-up button in the upper left corner of Xcode. From there, choose iPhone Simulator 3.2 (or higher). This will automatically move your application into the iPad Simulator. While your application is running in the iPad Simulator, you can zoom in on the action by pressing the 2x button in the lower right-hand corner of the Simulator. Additionally, you can zoom even further by opening the Window menu, hovering your mouse over the Scale item, and then choosing your desired zoom scale.

§III: A LOOK UNDER THE HOOD

Thus far, we've gone through three examples where we said "Hello world!" from inside the iPhone/iPad. The first example was from the View-Based Application template; the next two (related) examples were built from the Navigation-Based Application template. Most importantly, we've had an opportunity to tinker around with Xcode, Interface Builder, and the iPhone and iPad Simulators.

Indeed, the most important aspect of Chapter 2 was familiarizing you with the creative process in the context of programming apps: go in with an idea and come out with a tangible, working product. Several times I asked you to ignore heavy-duty code that I judged would be distracting or daunting. You may have also noticed that when you did try to understand some of this thicker code, it made sense in a weird, wonderful, chaotic way. Well, as we progress forward, we are going to make the "chaos" of the unknown less unsettling.

Before dealing with this issue, let me also put you at ease that when it comes to Objective-C, our programming language, I have yet to meet a single advanced programmer who actually knows every symbol and command. Just as in other industries, people tend to get very knowledgeable in their specific domains and specialize (e.g., integrating Google Maps to a game or an app).

An analogy I like goes like this: Car mechanics used to be able to strip an engine down completely and then build it back up—presumably better than it was. Nowadays, car mechanics are very specialized, with only a handful knowing how to completely strip down and rebuild a specific modern-day car. We get an expert in Ford hybrid engines, or an expert in the Toyota Prius electrical circuitry, or a specialist in the drum brakes that stop big rigs and so on. There is nothing wrong with this!

This is similar to how you are proceeding. You have just gotten your hands greasy and dirty by *successfully* programming three apps. And now, if all goes according to plan, you are going to walk toward the future brimming with confidence. I know from experience that confidence in my students can be derailed if they are intimidated or

blown away by too much complexity or technicality. I have found that students can handle bumps in the road if they know where they are going, and if they know that the rough stretches won't get too scary or dangerous.

You've Said "Hello!" … but now, INDIO!

We can divide most iPhone and iPad apps into four different functions: **I**nteraction, **N**avigation, **D**ata, and **I/O** (Input/Output). We have seen enough apps to know that we can interact with them; we can navigate from one screen to another; we can manipulate and utilize data; and, we can provide input (type, paste, speak) and receive output (images, sounds, text, *fun!*).

Before we zoom in again to approach a program from any one of these specific areas, we need to first have a better grasp of how these different aspects of iPhone/iPad programming work, look, and behave. We also need to learn about their limitations and the pros and cons in terms of the projected or desired user experience. And, because of the differences discussed above, whether the app is for the iPhone or the iPad.

As you gain a working knowledge of where any limitations and barriers exist, your journey through these four domains, all parts of a vast "forest," will be more powerful and productive. A very important part of my job is to show you how to conduct yourself safely through the Forest of INDIO. Some sections of the forest are more daunting than others, but the good news is that you will be getting a nice, high-level view, as from a helicopter! After our aerial tour, we will parachute down to the forest floor, open Xcode, and continue to explore the paths, the watering holes, and the short cuts—to mark off the unnecessary sections and to be on the lookout for wild animals.

Model-View-Controller

As mentioned previously, the programmers who developed Cocoa Touch used a concept known as the Model-View-Controller (MVC) as the foundation for iPhone and iPad app code. Here is the basic idea.

Model: This holds the data and classes that make your application run. It is the part of the program where you might find sections of code I told you to ignore. This code can also hold objects that represent items you may have in your app (e.g., pinballs, cartoon figures, names in databases, appointments in your calendar).

View: This is the combination of all the goodies users see when they use your app. This is where your users interact with buttons, sliders, controls, and other experiences they can sense and appreciate. Here you may have a main view that is made up of a number of other views.

Controller: The controller links the *model* and the *view* together while always keeping track of what the user is doing. Think of this as the structural plan—the backbone—of the app. This is how we coordinate what buttons the user presses and, if necessary, how to change one view for another, all in response to the user's input, reactions, data, etc.

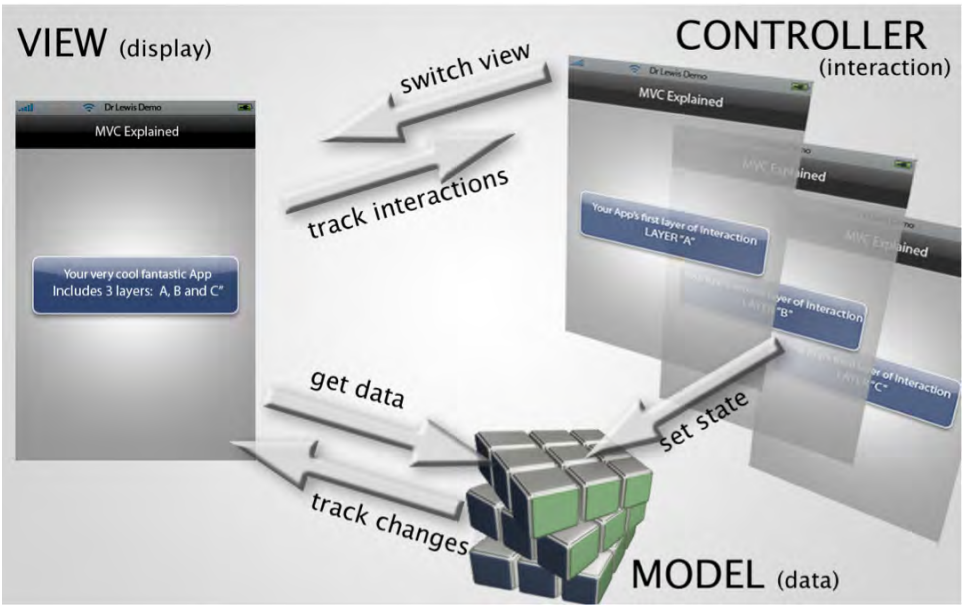


Figure 3–10. *The model, the view, and the controller (MVC)*

Consider the following example that illustrates how you can use the MVC concept to divide the functionality of your iPhone/iPad app into three distinct categories. Figure 3–10 shows a representation of your app; I've called it "MVC Explained." You can see that the VIEW displays a representation—a label—of "Your very cool fantastic App Includes 3 layers: A, B and C."

In the CONTROLLER section of the app, we see the three individual layers separated out, Layer "A," Layer "B," and Layer "C." Depending on which control mechanism the user clicks in the VIEW domain, the display the user sees, the CONTROLLER returns the appropriate response—the next view from the three prepared layers.

Your app will probably utilize data of some type, and this information will be stored in the MODEL section of your program. The data could be phone numbers, players' scores, GPS locations on a map, and so on.

As the user interacts with the VIEW section of the app, it may have to retrieve data from your database. Let's say your data contains the place your user parked her car. When the user hits a particular button in your program, it may retrieve the GPS data from the MODEL. If it's a moving target, it may also track changes in the user's position in relation to a car in the parking lot. Lastly, the CONTROLLER may change the state (or mode) of your data. Maybe one state shows telephone numbers, while another shows GPS positions or the top ten scores in a game. The CONTROLLER is also where animation takes place. What happens in the animation can affect and perhaps change the state in your Model. This could be done by using various tools, such as UIKit objects, to control and animate each layer, state, etc.

If this sounds complicated, bear in mind that you've already done much of this without even knowing it! In Example 1, you had the user press a button and up popped a label saying "Hello World!" This shows how you have already built an interaction with a ViewController. We will be delving further into these possibilities, of course. In Chapter 4, we will venture deeper into the **I**nteraction quadrant of the Forest of INDIO, and allow the user to add and delete table view items.

When we do this, I will do my best to keep you focused on the big picture when it comes to interactions … via **N**avigation. Our goal will be to have the user move from less-specific information to more specific information with each new view.

Chapter 4

An Introduction to the Code

Before examining specific pieces of code and the way we will create the apps set forth in this chapter, we need to spend a moment assessing our approach. We will be going through the material in this chapter a little differently than in the previous chapters.

I have included short bursts of information about each step, but without the details of the previous instructions. You will see many similarities between the functions and methods we are going to use in this next program and ones we utilized previously. If you need to go back and look at the detailed instructions, that is perfectly OK. It is for the sake of streamlining and saving space that I am giving fewer detailed instructions on the oft-repeated steps.

Another difference is that, after you have completed the upcoming program, we will do a code review. This section will appear after each major example throughout the remainder of the book, and we will call it "Digging the Code." In these reviews, we will go over some of the code we have written, and I will reference familiar code and explain processes in more detail. Here, I will begin to introduce you to more technical terms that you will use in future chapters and in correspondence with other programmers.

Consider this analogy: Thus far, I have taught you how to get into a car, turn the ignition, press on the accelerator, and steer as you move forward. In this chapter, I will guide you with similar directions, but in a different model of automobile on the way to a new destination—with fewer detailed instructions. After we get to our destination, I will open the hood and show you how, when you pressed the accelerator, it pumped gasoline into the engine, or how, when you pressed the brakes, it activated the disc brakes. By Chapter 8, we will go into the amount of gasoline being squirted into the pistons by the carburetors, and we will discuss how the brake pads heat up when going at a particular speed, and so on. Guess what—you'll be able to handle it!

So let's get on with the next application. Be ready to go back with me to review lines of your input as we focus on certain portions of the code, and as we look at how it all works together.

004_helloWorld: Buttons with Graphics

As indicated in Chapter 3, we will now embark on a journey in which we will explore the four categories of iPhone and iPad apps, abbreviated as INDIO: Interaction, Navigation, Data, and I/O. Our fourth example is very similar to the elementary "Hello World!" programs we have been creating so far, except that it shows how to do something that is not as intuitive as it may seem at first glance—how to handle buttons and graphics.

One of the ways I make sure I learn the names of all my students is to have them include photographs of themselves in one of the exercises. The students are told they must paste the pictures as the background to a simple "Hello World!" app. Invariably, they all think this is far too easy for them. It turns out that not only is there a struggle with sizing and placing the images on their apps, but then they also hide their code with the picture once they have it on there.

In this project, I have created a picture just as I ask my students to do, and I have placed it as a type of wallpaper. So when they place their faces in the appropriate spot, their photographs say "Hello" to me and tell me who they are. This exercise goes step by step through this process so that you can do the same thing with your photo. It will get a little tricky at times, so follow along carefully. Try not to fall into the trap of thinking that you can skip steps because you've already practiced on a significant portion of the initial code.

Take a deep breath, relax, and go through each step. You will learn some valuable tools in this exercise. Everything you need is right here in this book: screenshots and step-by-step instructions. I want to remind you that all the screenshots are directly from my screencast, available at:

`http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/002_helloWorld_002.htm`

1. As always, let's start with a clean, uncluttered desktop. Close all programs using the Command + Tab and Command + Q shortcuts until only the Finder remains open.

   You will see that I have created a picture of myself to use as an example in this app. So, whenever you see my photo in the book, you will want to see a photo of yourself there. Use any program you want to create an image that is 320 pixels in width by 480 pixels in height. If it is not *exactly* in conformity with the size as depicted in Figure 4–1, your iPhone may give you all kinds of zany effects, including not displaying the image at all. For good practice, just accept that you need to keep full images sized to 320 × 480. Later, when we have bars on the top, it will all change. For now, let's get 320 × 480 into your brain.

Figure 4–1. *Create an image of yourself in a photo-editor that can save it as a .png file.*

As you can see in the screenshot, I have used Photoshop. You can copy all the parameters exactly from Figure 4–1 to help make sure that your photo will fit correctly into your app. Save the picture to your desktop as a `.png` file named `helloworld.png`. We do this because iPhones talk to `.png` files—not to `.jpg` or anything else. Don't question it. Just accept it. You need to use a program that can create, edit, and save `.png` files.

If you do not have Photoshop, or a similarly powerful editing application, and you plan to program iPhones/iPads with graphics, I strongly suggest that you invest in a copy. Once your image is created and saved to your desktop, quit Photoshop (or equivalent). At this point, you should have a beautiful, empty desktop, except for the image.

2. Let's start by opening a View-based Application in Xcode. Open Xcode, and then enter ⌘⇧N to open a new program file. In Figure 4–2, you can see that my View-based Application icon was highlighted by default. If yours is not, then click on it.

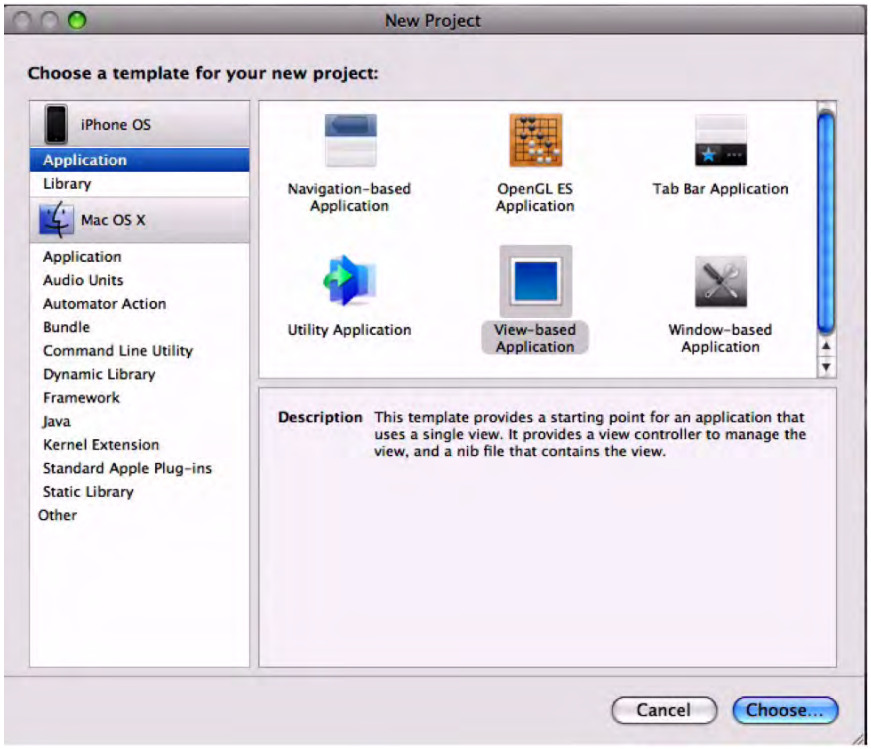


Figure 4–2. *After entering ⌘⇧N, create a View-based Application.*

3. Save your View-based Application to the desktop as `helloWorld_004`. See Figure 4–3.

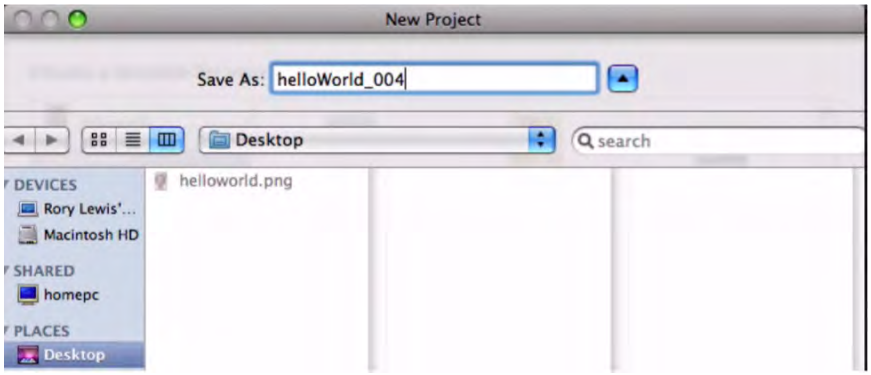


Figure 4–3. *Save your View-based Application to your desktop as helloWorld_004.*

4. As shown in Figure 4–4, once you save your project to your desktop, Xcode instantiates a project named `helloWorld_004`, as indicated by the name on the top of the window. (If you're not already familiar with the term *instantiate*, see "Digging the Code" at the end of this chapter.) By clicking on the root file of your `helloWorld_004`, you can see the `plist` file, the `AppDelegate` files, and the `ViewController` files. These files are going to be your friends, and, as we move through each new example, you we will get to know them better and better.

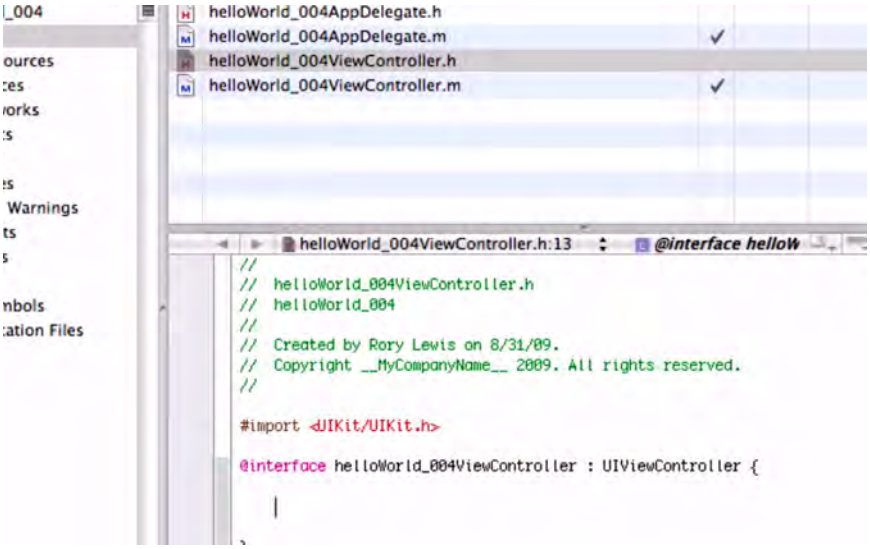


Figure 4–4. *This is the view of your helloWorld_004ViewController file before you've entered your new code.*

Looking at the files, we see that some end with `.h` and others end with `.m`. The `.m` represents *implementation files*, and the `.h` represents *header files*. At this point, you should know that header files contain classes, types, functions, and constant declarations. In contrast, implementation files contain the lines that use already coded material we have declared. In this exercise we'll be using the `.h` files. This is enough for now, so let's move on to the next step.

5. Just as we did back in the very first exercise, after opening the interface file, let's scroll down past `#import <UIKit/UIKit.h>`, which is the command that imports the UIKit framework. Keep scrolling, past

```
@interface helloWorld_004ViewController : UIViewController {
```

and then hit Return to make space for the code that you will enter in the next step.

6. As before, add onto the line that is below the open bracket (the { character) the following command:

```
IBOutlet UILabel *label;
```

This adds an Outlet to a label, just as we did in previous examples. Do not forget the semicolon (;) at the end of the line, as this tells the compiler you're done talking to it for the moment.

7. We now need to add an *action* to a button. Enter the close bracket (}) and write the following:

```
- (IBAction)hello:(id)sendr;
```

Make sure to put the semicolon at the end of this line. In the following code, you see what your file should look like.

```
//
//  helloWorld_004ViewController.h
//  helloWorld_004
//
//  Created by Rory Lewis on 6/13/09.
//  Copyright __MyCompanyName__ 2009. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface helloWorld_004ViewController : UIViewController {
        IBOutlet UILabel *label;
}

- (IBAction)hello:(id)sendr;

@end
```

After selecting the line

```
- (IBAction)hello:(id)sendr;
```

copy it by highlighting and using the shortcut ⌘C.

8. Save your file by entering ⌘S. Figure 4–5.

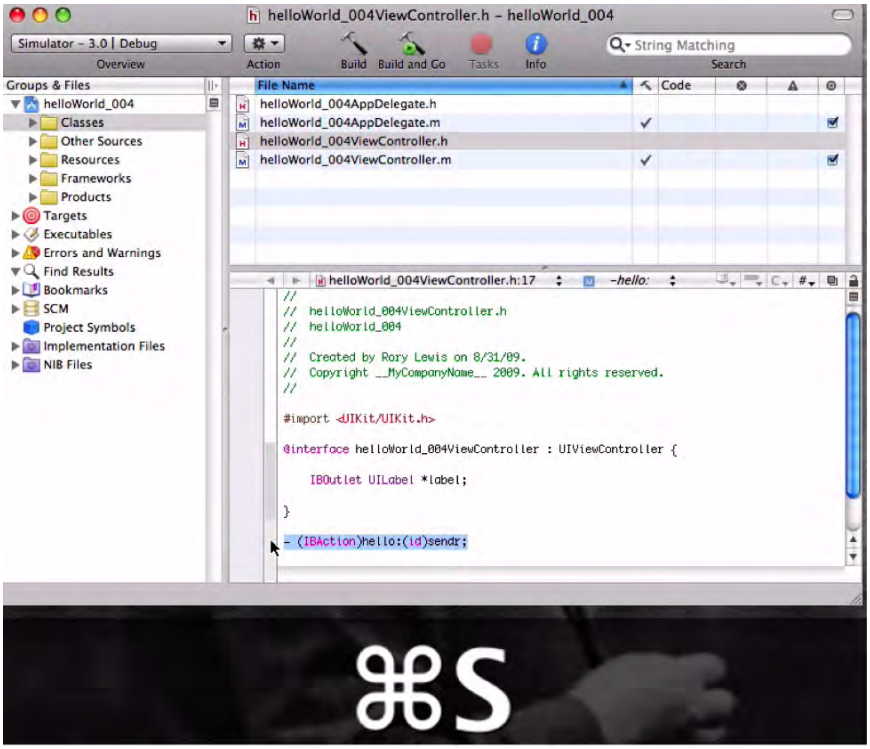


Figure 4–5. *Save your work by entering ⌘S.*

9. Now that we have set up our header file, we want to move to our implementation file: `helloWorld_004ViewController.m`. This file manages how your code interacts with the display. Open this file, scroll down past the green comment lines, and insert the line that you just copied, by clicking in the correct location and using the shortcut ⌘V.

```
#import "helloWorld_004ViewController.h"

@implementation helloWorld_004ViewController
*** THIS IS WHERE YOU PASTE ***
```

10. Delete the semicolon at the end of the line you just pasted and replace it with an open bracket as shown in Figure 4–6.

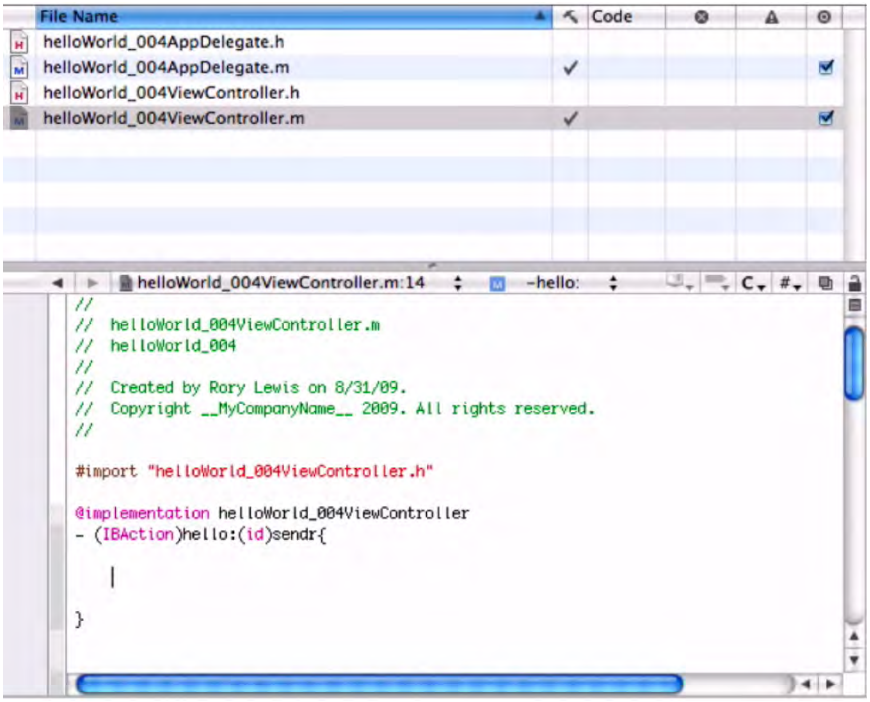


Figure 4–6. *This is the implementation file after you insert the copied line of code and enter an open bracket "{" in place of the deleted semicolon.*

```
#import "helloWorld_004ViewController.h"

@implementation helloWorld_004ViewController
- (IBAction)hello:(id)sendr{
```

11. After you delete the semicolon and replace it with an open bracket, go to the next line and enter this code

```
label.text = @"OK I'm repeating myself!";
```

followed by a close bracket on the line below that. Refer to Figure 4–7.

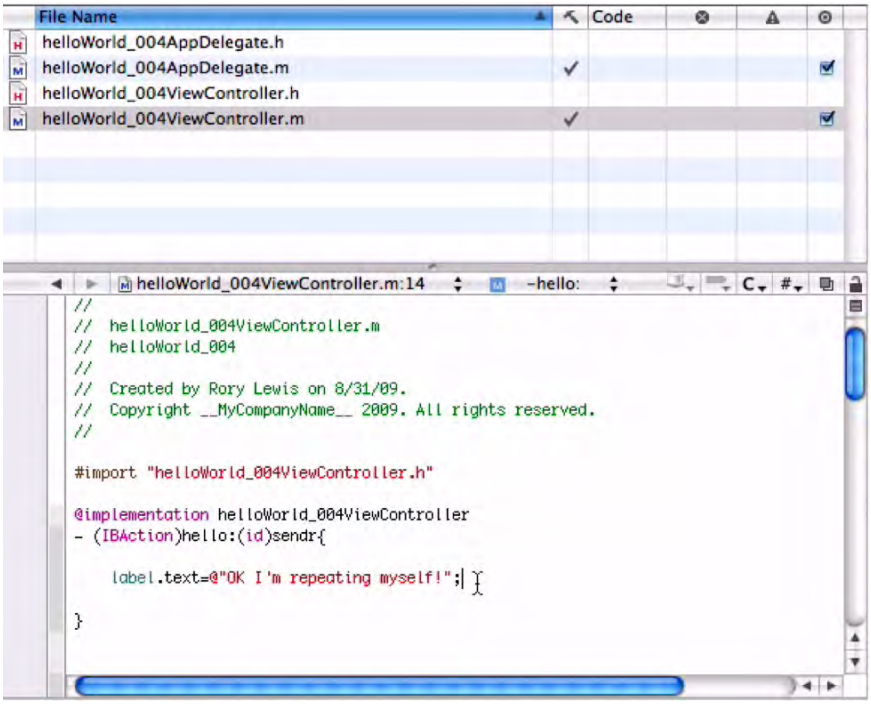


Figure 4–7. *Enter the command that will generate the text "OK I'm repeating myself!" Then, save your file by using the indicated shortcut.*

```
#import "helloWorld_004ViewController.h"

@implementation helloWorld_004ViewController
- (IBAction)hello:(id)sendr{
label.text = @"OK I'm repeating myself!";
}
```

12. Now, save your work on this interface file (⌘S).

13. Click the `Resources` folder. I want you to activate it so that you can save your image file there. See Figure 4–8.

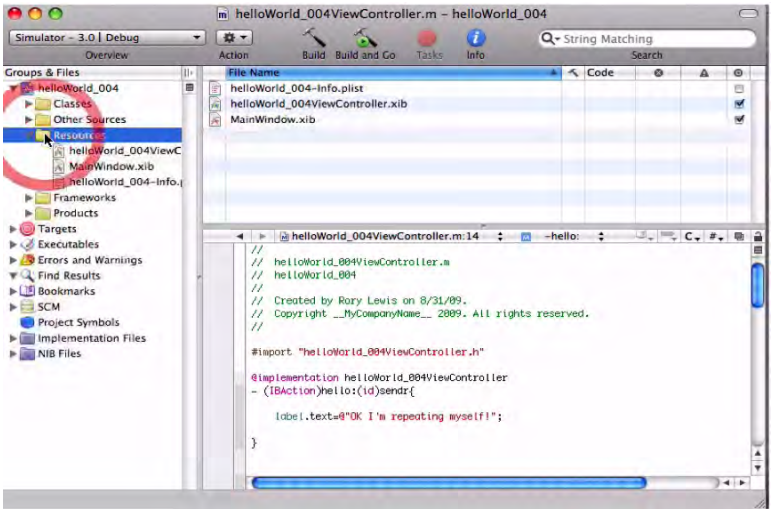


Figure 4–8. *Click on the Resources folder.*

14. Locate your `helloworld.png` image on your desktop and drag it into the Resources folder. I'd like you to get into the habit of saving your images here. It's a good habit that will help you to stay organized. Later on, if you begin working with huge, image-driven files, you may want to create your own `Images` folder. For now, drop your images into the `Resources` folder.

15. Check the box that states: "Copy items into your destination group's folder." Doing so ensures that even if you change the location of your folder or delete the image, this image will still appear in your application file because Xcode has made a copy of it in your `Resources` folder.

    For example, right now you have the image stored on your desktop. If you delete it from there, or send this folder to the Internet or an iPhone, without checking this box, then it will not be a part of your program. Checking this box guarantees that the `helloworld.png` you're putting into the `Resources` folder will be wherever the program is. To complete this step, click the Add button or hit your Return key.

16. Once you have opened the `Resources` folder, double-click on the nib file, as shown in Figure 4–9. As you know already, this is the file that ends with `.xib`—specifically the `helloWorld_004ViewController.xib` file—and opening a nib file automatically opens up Interface Builder, which is how and where we visually control our app. We want to do this because it's time to connect pictures, buttons, and labels to the code you've been writing.

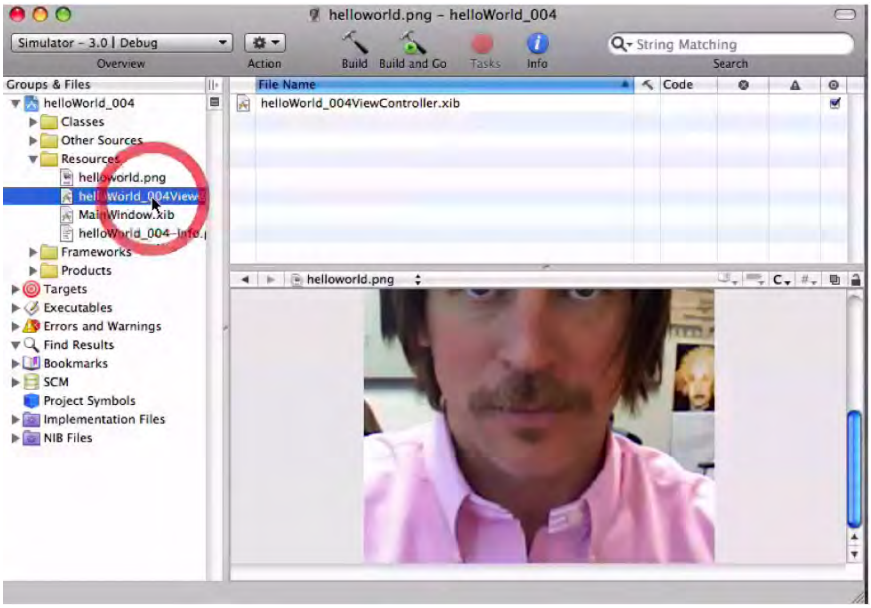


Figure 4–9. *We want to open Interface Builder, so click on your nib file.*

17. We need to have some wallpaper of your image that lies underneath all the buttons and labels we will be including in our app. This means that we need a place where the image can reside. In Interface Builder, you can place images on the Image Views. Scroll down in your Library until you see the Image View icon. Drag one onto your View frame, as shown in Figure 4–10.

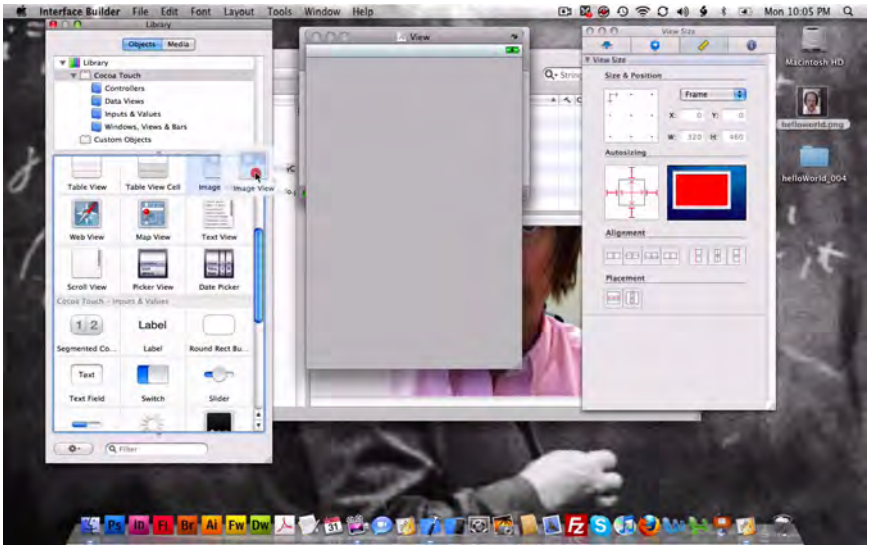


Figure 4–10. *Drag the Image View icon onto your empty View frame.*

18. We want to connect your `helloworld.png` to your image view so that it will appear. The way that you do this is go to the Information tab of the File Attributes window, open the drop-down window, and select your saved image.

    We need to make sure that your image, `helloworld.png`, appears in your View frame. If the image that you prepared earlier doesn't display properly, you need to stop right now and carefully go through all these steps again.

    If there is a problem at this point, potential reasons might be that the image file was not properly saved into your Resources folder, or that it was not saved in the proper format, as a `.png` file.

19. Go back to your Library, scroll down to the Label icon, and drag it onto your image. You can place it anywhere you like. I chose to place it on my head. See Figure 4–11.

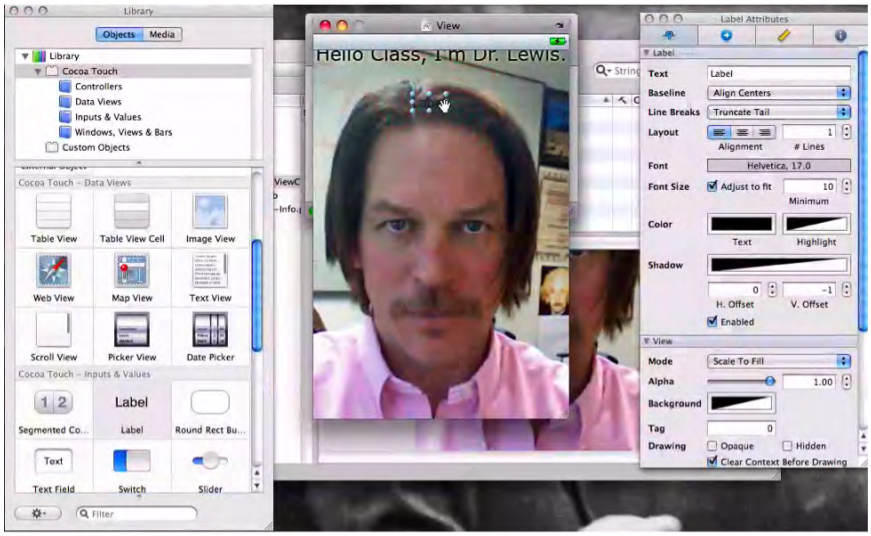


Figure 4–11. *Click in the Label option to drag a label onto the image.*

20. Just as we did in previous exercises, we want to remove the default label. So delete the text that's already there. Also, go to the color box and give it a click; we want to change it to a different color. See Figure 4–12.

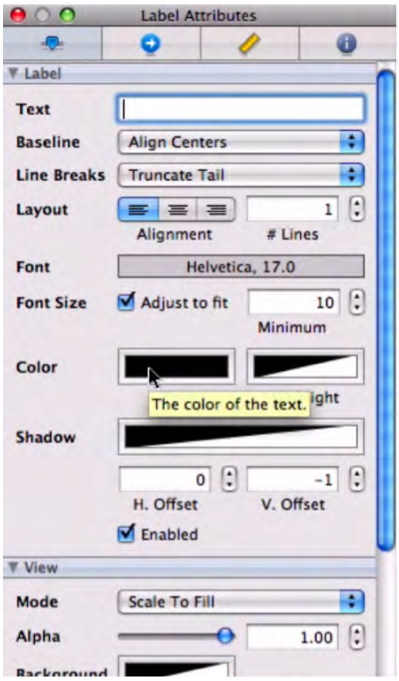


Figure 4–12. *Remove the default text inside the Text field, which is at the top of the Label Attributes frame, and then insert "OK I'm repeating myself!"*

You can change the color of the text inside the label to any color you want. I chose to change mine to white because my hair is brown. If your hair is blonde, for instance, you may want to select a darker color for contrast.

21. We want to expand the label to be of sufficient size to hold our text. Drag one of the dots (size indicators) on the label to get to a size you believe will hold the text, which in my case is "OK I'm repeating myself!"

22. As shown in Figure 4–13, we now need to drag a button onto the view. When a user presses the button, it will display the label.

Figure 4–13. *Drag a button onto the View frame.*

23. As shown in Figure 4–14, we also want to change the button's default text. On the Button Attributes tab, change the text in the Title field to "Press Me."

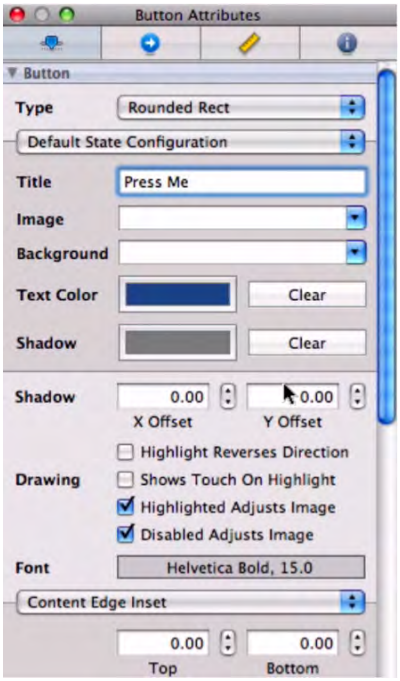


Figure 4–14. *Delete the default text in the Title field, and then enter "Press Me."*

24. On the Attributes frame, you can also adjust the transparency so that we can partially see through the button. Go to the bottom of the window and change the Alpha slider to a level that looks right to you. This effect is illustrated in Figure 4–15.

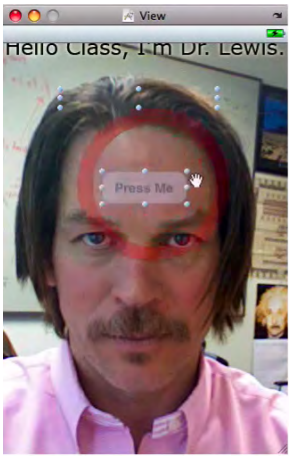


Figure 4–15. *The semi-transparent button has been selected along with the label in order to adjust alignment and placement.*

25. With both the button and label selected, line them up by choosing a Placement option at the bottom of the View Size frame.

26. Next, hold down the ⌘ key while you click and drag the File's Owner icon to the Label. Interface Builder will display the now-familiar "fishing line" to demonstrate the relationship. See Figure 4–16.

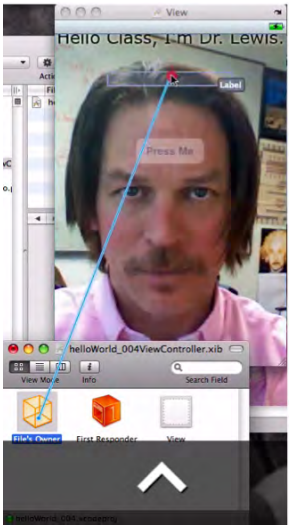


Figure 4–16. *Command(⌘)-drag the File's Owner icon to the Label.*

27. As shown in Figure 4–17, establish the connection between the File's Owner and the Label by choosing from the drop-down menu.

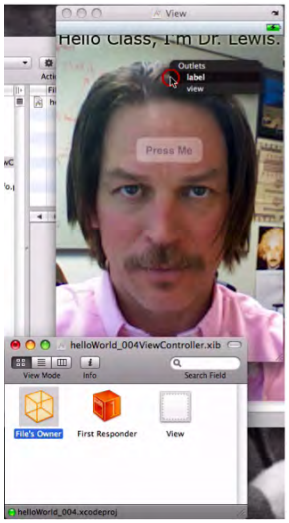


Figure 4–17. *Establish a connection between the File's Owner and the Label by selecting from the Outlets drop-down menu: label.*

28. Now, hold down the ⌘ key while you click and drag from the Button to the File's Owner icon, as shown in Figure 4–18. Once these elements are connected, the pull-down menu appears and you can select the text "OK I'm repeating myself!" Save your project (⌘S), and exit Interface Builder (⌘Q).

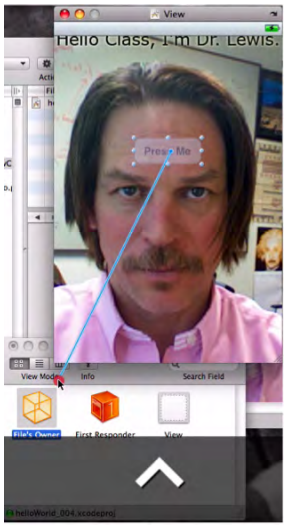


Figure 4–18. *Command(⌘)-drag from the Button to the File's Owner icon. After you are presented with a drop-down menu in which you select your desired message, save (⌘S) the file and quit (⌘Q) Interface Builder.*

29. Finally, enter ⌘↩ so that you can run the code. When your program launches, you will see your picture (or mine) overlaid by a semitransparent button. Follow the directions … push the button and check out the result. Figure 4–19 shows the iPhone Simulator, while Figures 4–20 and 4–21 demonstrate the iPad Simulator.

Congratulations! You have integrated user/programmer interaction with some cool graphics. You've also been able to get through this with fewer instructions from me.

Well Done!

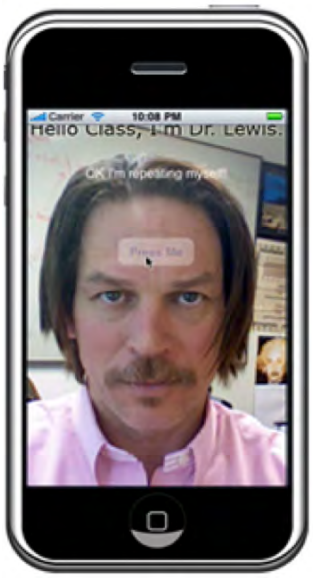


Figure 4–19. *Click the button, and … it works!*

Figure 4–20. *This is the iPad Simulator view in iPhone view mode – before the button gets pressed.*

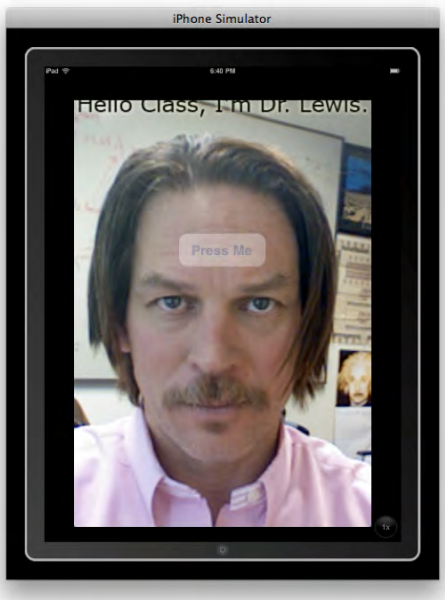


Figure 4–21. *Here is the iPad Simulator in full (2x) view mode – before the button gets pressed. The "1x" button at the lower-right will change the view to the one in Figure 4–20.*

Digging the Code

In this new section of the chapter, we will be examining several concepts that have been mentioned but that are probably still shrouded in some degree of mystery. This is an opportunity to read along *without* definitive understanding. I hereby give you permission to partially "get it." Of course, if you happen to attain full comprehension of the subject matter in all its details, let me be the first to commend you. Meanwhile, see if holding on more loosely doesn't, in fact, give you a firmer grasp on the big picture.

Nibs, Zibs, and Xibs

These "items" are basically all the same thing, but different people refer to them in different ways. At a recent conference in Denver, "360iDev for iPhone Developers," it was interesting to hear how so many presenters referred to `.xib` files as "nibs" or "zibs." Most programmers prefer to say "nib" files. No matter how we refer to them, it's important for us to understand what's going on with these files. What are they? Do we need them? Do you need to know how they work?

Do you recall, from step 16, how we opened Interface Builder? That's right – it happened automatically when you clicked on a nib file. Once it opened, you saw that the file contained all the code associated with our buttons and images, and that, in fact, this information is stored there. That way, when you run the app, all the objects and all the links associated with the objects are integrated properly, and they can then "magically" come together and give the user the experience that you envisioned.

It turns out that nib files, when examined at the level of Cocoa or Objective-C, contain all the information necessary to activate the UI files, transforming these into a graphical iPhone or iPad work of art. It's also possible to join separate nib files together to create more complex interactions, and you'll see this in the next chapter.

All the information that resides in these files is put there so that it can *create an instance of* the buttons, the labels, the pictures, and so forth that you've entered. This collection of commands is plonked down and saved into your nib files to become the UI. The code and the commands taken together become real, and they are sensed by the user – seen or heard, or even felt.

We sometimes use the term *instantiate* in a similar fashion. For example, when you first save a new project, the computer instantiates – makes real and shows you the evidence for – a project entity by virtue of assigning it a body of subfiles, as it were. In helloWorld_004, you saw how the project was instantly given "arms and legs" … two AppDelegate files and two ViewController files.

We say we've "created an instance of" something when we've told the computer how and when to grab some memory and set it aside for some particular process or collection of processes such that, when the parameters are all met, the user has an experience of this data (i.e., whatever was assigned in memory). Sometimes we refer to these collections or files of descriptions and commands as *classes*, *methods*, or *objects*. In this code-digging session, these terms might seem to run together and appear as

synonyms, but this is not the case. As you read on, you will come to understand each term as a distinct coding tool or apparatus, each to be employed in a particular situation, relating to other entities in a grammatically correct way.

When we say that you created an instance of the buttons and labels in your nib file, what we're really saying is that, when you run your code, a specific portion of your computer's memory, known by its address, will take care of things in order to generate the UI – the user experience – you have designed. Each time your application is launched on an iPhone or iPad, the interface is recreated by the orchestrated commands residing in your nib files.

Consider the nib file associated with the action depicted in Figure 4–13. You dragged a button from the Library into the View window, and thus you *created an instance of* this button. If somebody were to ask you what that means, you might look them in the eye, with a piercing and enigmatic look, and say, "By creating an instance of this button, I have instructed the computer to set aside memory in the appropriate `.xib` file, which, upon the launching of my app, will appear and interact with the user, *precisely as I have intended*."

Methods

The next concept I would like to explore a little more deeply is that of *methods*. As I did with nibs, I am only going to give you a high-level look at this time. You've already used methods pretty extensively, so I'm simply going to tell you what you did.

Looking at Figures 4–18 through 4–21, recall how you ⌘-dragged from the button to the File's Owner icon and, when the pull-down menu appeared, you selected the statement "OK I'm repeating myself!" Guess what—that was done with methods!

Looking way back to step 7, I want you to understand that when you copied and pasted the command

```
- (IBAction)hello:(id)sendr;
```

you were instructing the computer to associate an action with a button. The first symbol in this piece of code is a minus sign (–). It means that `IBAction` is something we call an *instance method.* On the other hand, if you had entered a plus sign (+) there, as in + (`IBAction`), we would have called it a *class method*. One symbol announces (to the processor) an instance, while another symbol announces a class. What these two statements have in common, though, is the method: IBAction. Furthermore, just by the name alone, we see that this is an action that will be performed in Interface Builder.

Meanwhile, let's look at what we know about instances. Back in step 9, I remarked that we had finished our work on the *header* file, and that it was time to work on the *implementation* file, `helloWorld_004ViewController.m`. This is the file that manages the code, and that is why you were instructed to open it. I had you insert a line of code in this (`.m`) file by virtue of copying and pasting from the header file: `- (IBAction)hello:(id)sendr;`

```
#import "helloWorld_OO4ViewController.h"

@implementation helloWorld_OO4ViewController
- (IBAction)hello:(id)sendr;
@end
```

You deleted the semicolon at the end of that line and replaced it with an open bracket ({). The important issue here is that the (`.m`) file then contained the commands for the object, and that these go in between the "`@implementation`" section and the final "`@end`" `command`. Our method then appeared like this:

```
#import "helloWorld_OO4ViewController.h"

@implementation helloWorld_OO4ViewController
- (IBAction)hello:(id)sendr{

}

@end
```

But wait – I really haven't explained the term *method* yet. OK, see if this helps. Methods are the packets of code that the computer executes when a particular input is received and must be processed—such as the pressing of a button.

Consider this analogy: a programmer says, "Here comes an app that will assist you in drawing a nice, pretty house." That is a header type of announcement. Then, the programmer enters specific instructions for how the house will be constructed, how it will sit on/against the landscape, what kind of weather is in the background, and so on. "Draw a slightly curving horizon line one third from the bottom of the display, and midway on this place a rectangle that is 4×7, on top of which is a trapezoid with a base length of …," and so on. These specific, how-to instructions belong in the implementation file, for they describe the actual actions – the *method* – of drawing the house.

So, to connect your button to a method named "`hello`," you added this code:

```
- (IBAction)hello:(id)sendr;
```

This *created an instance of* your `hello` method. Then, you created a place in memory to execute the code inside your `hello` method.

To summarize:

1. You first dragged a button from the Library into the document window and created an instance of that button.

2. You then ⌘-dragged from the button to the File's Owner icon and connected the button to the `hello` method listed in the pull-down menu.

3. The moment you entered the code `- (IBAction)hello:(id)sendr;` you created this "`hello`" instance method:

- The (-) means that `IBAction` is an *instance* method.
- `(IBAction)` is the returned *type*.
- `hello` is the method *name*.
- `(id)` is the *argument* type
- `sendr` is the argument *name*

That's all we'll do with methods for now. We'll be repeating these steps of creating and using methods in the chapters ahead, and as you practice, the concept will become clearer and clearer.

Chapter 5

Buttons & Labels with Multiple Graphics

In this chapter, we'll tackle our fifth program together, and it's time to quicken the pace a bit. As in Chapter 4, you'll be able to simply view the screen shots and implement the code if you remember most of the details—steps that have been described repeatedly in the previous examples. There will be fewer figures pertaining to each step, yet more procedures; we will be using the short bursts of information introduced in Chapter 4.

Also, as in Chapter 4, once you have completed the program, we will do a code review in the "Digging the Code" section. Initially, we will cover some of the same aspects and concepts we discussed in this section in Chapter 4, and then we will zoom in on some of the new code. Not only will we go a little deeper, but we will also expand our horizons to consider other computing concepts that link up to this deeper level of analysis.

You will probably also notice a change of style in Chapter 5, for we will be moving away from the "elementary" language used in previous chapters. So, let's pick up the pace—a little faster, a little more advanced, and using more of the technical nomenclature. Again, if you don't grasp every concept and technique fully, that is perfectly okay! Relax and enjoy this next example.

helloWorld_005: a View-Based Application

In this example, we continue to delve deeper into INDIO—Interaction, Navigation, Data, and I/O (input/output). Our code will retrieve two images, by different means, which will interact with the user. This exercise gets us closer to the I/O aspect of INDIO, which we have not quite wrapped our heads around because, simply put, we're not there yet.

Like a digital warrior, you are striding along a path into the forest of Objective-C, to a place where you will need to be accustomed to vaulting over rivers, hunting tigers, building fires in the rain, and so forth. So far, you've learned to start a fire with flint and steel—in dry weather, and you've gotten pretty good at hopping over streams. This

lesson will teach you to vault over wider streams and, once there, to hunt tigers. Soon, you'll be equipped to fight the legendary demons of I/O in the daunting realm of INDIO.

Right this second—feel good about yourself! You are already quite deep into the Forest of Objective-C. If you lose your way, remember that you can refer to the "map" at this link:

`http://www.rorylewis.com`

Preliminaries

As usual, let's begin with a clean desktop and only four icons: your Macintosh HD and three image files (shown as icons in Figure 5–1). As I'm sure you have gathered by now, I think it's essential to have an uncluttered desktop, and I want to encourage you to continually hone your organizational mindset. Using our familiar shortcuts, close all programs.

Figure 5–1. *Create three .png image files: a bottom layer, a top layer, and a desktop icon. Save them all to a beautiful, clean desktop.*

You are welcome to download these images (from `www.apress.com`), which will become key building blocks of this project, but we really want to encourage you to find and prepare images of your own. That way, you'll have more passion about this assignment. You have two basic choices at this point: download our images here, or prepare your own. Assuming that you are willing to go through the effort of creating three distinct photo files of your own choosing, pay attention to the following guidelines.

The size of the first picture will be the iPhone standard of 320 pixels in width by 480 pixels in height. This will be the bottom layer of two images, so we'll call it the *background* layer. Our background, then, is a photograph of the stairs leading out the back of the Engineering building, here at UC-Colorado Springs. We will use this picture

as a backdrop for a picture of my wife, Kera. When the program is run, the background will display and, once a button is clicked, up will pop the photo of Kera at the top of the stairs. How nice—Kera Lewis has decided to return to University!

This is our scenario then: You see a familiar background, and then somebody (or something) unusual or unexpected suddenly appears. This background picture needs to be 320 × 480 pixels, and it needs to be saved as a `.png` file (Figure 5–2).

Figure 5–2. *This is the background image – or bottom layer.*

In order to create the second image, which we'll call the *top* layer, copy the background layer photo. Then crop this copy to create an image with these exact dimensions: 320 × 299 pixels. Yes, I know the height is a strange number—trust me! Now you have a roughly square copy of the bottom two-thirds of your background photo.

Next, paste onto this a partial image—probably a cut-out of some interesting or unusual object. This will yield something like the image in (Figure 5–3): Kera Lewis, in front of the background scene. This modified top layer will, of course, be saved as a `.png` file.

Figure 5–3. *This is the modified top-layer image, which will overlay the background.*

Thus, you will end up with a prepared top layer that consists of the bottom section of the original background photo, with some interesting person or object pasted over it. You can probably guess that we're going to program the computer to start with the background image, and then, with some user input, insert the top layer—with bottom edges matching up flush, of course. This will give the illusion that our interesting guest, or object, suddenly materialized out of nowhere. Our top layer will not affect the space near the upper part of the background; we are reserving this region for the text that we will also direct the computer to insert. We go this route because the iPhone and iPad do not support `.png` transparency.

The third image file is an icon of your choice. As in the previous chapter, you may want to customize your icon. In my case, I took a portion of the photograph of my wife's face and put it into my "icon" file (Figure 5–4).

Figure 5–4. *This is the image for the screen icon … a lovely face!*

Recall that icons for the iPhone/iPad have a recommended size of 320 × 480 pixels. Make sure to be mindful of these dimensions. Once you have all three of these images—the bottom layer, the top layer, and the icon—save them onto your desktop.

Xcode – Beginning a New Project

With the preliminaries out of the way, let's start building the application:

1. Start by opening Xcode and entering ⌘⇧N. In the New Project window, select the View-based Application template, as illustrated in Figure 5–5. You may be thinking that a view-based application template is usually used to help us design an application with a single view, and that we should pick another option—because we've just made two views, the image of the stairs and the modified image of the stairs with my wife in it. This reasoning would appear to be sound because navigation-based applications yield data hierarchically, using multiple screens. That choice would seem to be the right one for this project, except that this is actually not the case here.

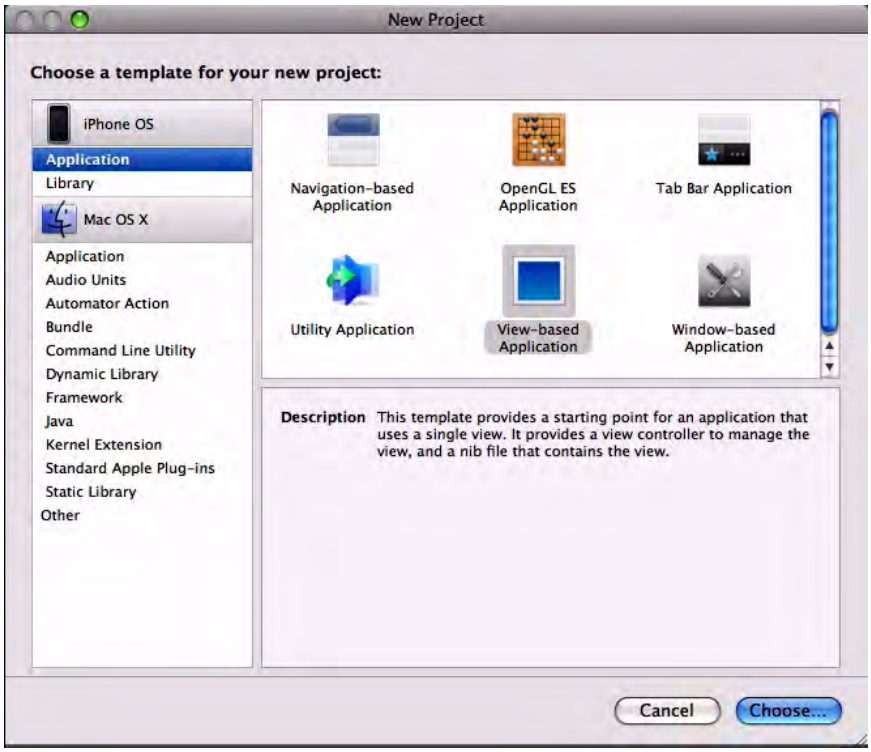


Figure 5–5. *Enter ⌘⇧N and select View-based Application from the New Project window.*

We will be dealing with only one perspective onto which we will superimpose an image, not a view. If we were going to have portions of our code in one navigation pane, and other portions of our code in other navigation panes, then we probably would choose a navigation-based application. In this current project, though, we are going to manipulate one view in which we will superimpose images, rather than navigate from one pane to another. In essence, we'll be playing tricks with a single view.

2. Save your View-based Application to your desktop as "`helloWorld_005.`" See Figure 5–6. This is going to be the last of our "Hello World!" apps. I'd like to suggest that, once you've completed this program, you save all of these in a `Hello World` folder inside your `Code` folder. You will probably find yourself going back to these folders at some point to review the code.

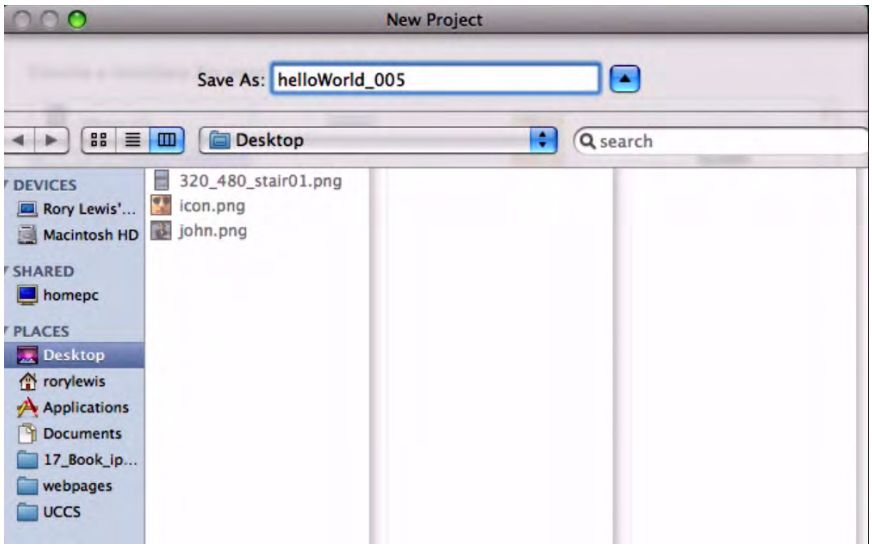


Figure 5–6. *Save your View-based Application to your Desktop as* `helloWorld_005.`

Later in the book, when we go into the details of Objective-C and Cocoa, there is a good chance that you'll scratch your head and say, "Damn—that sounds complicated, but I *know* I did this before. I want to go back and see how I connected these files in those 'Hello World!' exercises I did at the beginning of this book."

As I hope you are now beginning to see, Xcode instantiates a project named `helloWorld_005` (see Figure 5–7). Click on your `helloWorld_005` root file in the Overview—Groups & Files section, and then let's pause for a second.

Do you see the folders named “Classes,” “Other Resources,” “Resources,” and so on? As mentioned earlier, we’re moving on from our elementary language and I will be throwing out some technical jargon that is more specific.

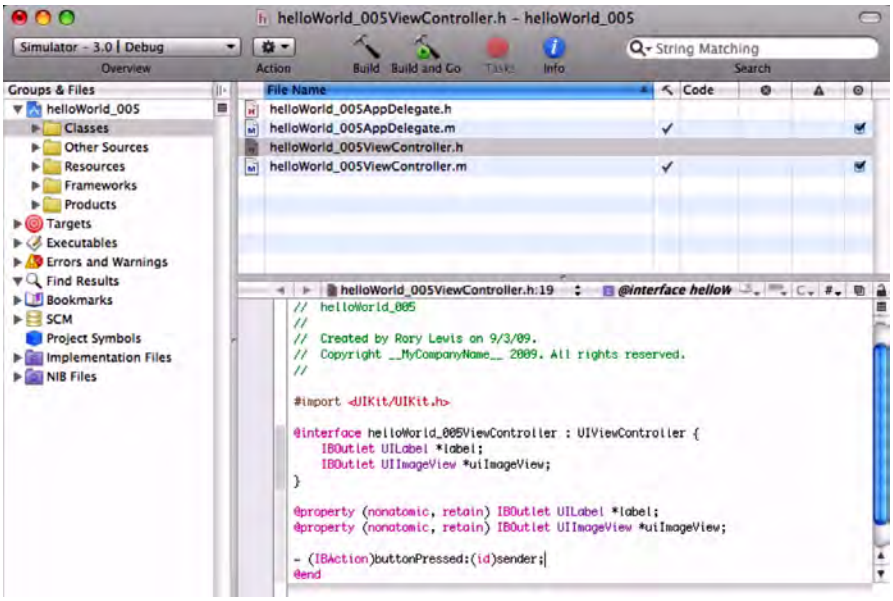


Figure 5–7. *After you save the project, Xcode instantiates a project named* `helloWorld_005`*, as indicated by the name at the top of the window.*

Instead of saying, “Open the Classes folder,” I might say, “Click the ‘disclosure triangle’ next to the Classes folder.” This, as you may have gathered, is the small triangle pointing to the right that, when clicked, rotates to point downward and reveals the files within a folder. Knowing this nomenclature can even come in handy—like at a cocktail party or to win a bet with one of your fellow programmers! So if you’ve never heard of that little arrow being referred to as the ‘disclosure triangle’—hey—you’re slowly morphing into a geek!

Understanding IBOutlets

You now see two `AppDelegate` files and two `ViewController` files. Previous chapters have already discussed the `.m` and `.h` extensions in detail. We’re going to do what we and most Cocoa and Objective-C programmers do—start off by programming the header files. In geekspeak, you’d say, “Open the header.” If anybody were to ask you what you mean, you’d say, “Click on the disclosure triangle in your `Classes` file and open the file with extension `ViewController.h`!”

You've already programmed four previous header files, so you should be accustomed to just flying over this portion of your code. But this time, we're going to put on the brakes and think about what we're doing. For all our previous examples, we've only had to use one `IBOutlet`, a thing that allows us to interact with the user. Let's get more technical and specific, for that statement is too elementary. Let's dig deeper into what an `IBOutlet` is so that, when we get to the "Going Back and Digging That Code" section, you'll be able to really understand it.

Open your `helloWorld_005ViewController.h` file and read the following code. Let's see if we can find our way to a deeper understanding of these elements:

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

@interface testViewController : UIViewController {

}

@end
```

Look at the first line: `#import <UIKit/UIKit.h>`. This is what permits us to use the `IBOutlet` keyword. We use `#import` to import the `UIKit`, which is the user interface (UI) framework inside the huge body of core chunks of code called IPhoneRuntime, which is a stripped-down version of the OS X operating system found on a Mac. Of course, IPhoneRuntime is smaller, so it can fit onto an iPhone or an iPad.

When we import the `UIKit` framework, it delivers to our toolbox the ability to use tons of code Apple has already written for us—called *classes*—one of which is the very cool and popular class that you've already used: `IBOutlet`. The `IBOutlet` keyword is a special directive called an "instance variable" that tells Interface Builder to display items that you want to appear on your user's iPhone or iPad. In turn, Interface Builder uses these "hints" to tell the compiler that you'll be connecting objects to your `.xib` files. Interface Builder doesn't connect these outlets to anything, but it tells the compiler that you will be adding them.

In our exercise, we'll be using two `IBOutlets`—one dealing with our text and the other with our image. To recap: in previous examples, we've only had to use one `IBOutlet`, but in this example we will need two. One `IBOutlet` will be used for our text (something like "Hello World, I'm back!"), and one will be used for the top layer image of Kera (Figure 5–3).

Keeping inventory of what we'll be using: 1) the background image of the stairs, 2) the top-layer image of my wife, and 3) the text of what she will be "saying" upon her return to the campus.

We now know that we need two `IBOutlets`. We can start by focusing inside the brackets that follow `@interface testViewController : UIViewController`. Our code will need to appear as follows.

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

@interface testViewController : UIViewController {
IBOutlet
```

```
IBOutlet
}

@end
```

As you can see, we have no code here yet—just placeholders, but we know what's coming: two `IBOutlets`. We now know for what purpose each will be utilized; one will produce text for what Kera says, and the other will produce a picture that is superimposed on top of the background.

We know that when we shoot text out onto the iPhone or iPad screen we use the `UILabel` class. This class draws multiple lines of static text. Therefore, go ahead and type in `UILabel` next to your first `IBOutlet`, as shown in the following code. Now, consider what we will need for the second `IBOutlet`. We know that we want to impose the top layer image as shown in Figure 5–3.

A good idea here would be to use the `UIImageView` class because it provides us with code written by Apple that can display either single images or a series of animated images. With this said, enter the `UIImageView` class next to your second `IBOutlet`.

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

@interface testViewController : UIViewController {
IBOutlet UILabel
IBOutlet UIImageView
}

@end
```

Pointers

Now that we have the means to push text and an image onto the screen of the iPhone/iPad, we need to specify which text and which image. We sometimes use predefined code, created by the folks at Apple, that does what it does by virtue of referencing or pointing to our resources—that is, our text and images. As you are beginning to see, this is the context in which we will be using pointers.

In previous examples, we told you not to worry about that star thing (*). Well, now it's time to take a look. Let's focus for a moment on how these (*) things—*pointers*—do what they do. At the end of the chapter, in "Digging the Code," we'll get into it even more, but here's an introduction.

We need an indirect way to get our text and picture onto the screen. We say "indirect" because you will not be writing the code to accomplish this—you will use Apple's code to retrieve these. You will call up pre-existing classes, and then these classes will call up your text and your image. That is why we say this is an indirect means of obtaining your stuff.

Consider this little analogy. Suppose you make a citizen's arrest of a burglar who broke into your house. You call the police and when they arrive, you point to the criminal and say, "Here's the thief!" Then, the policeman, not you, takes the criminal away to deal with the accused.

Now, you want to display text on your iPhone/iPad. You call `UILabel`, and when it "arrives," you point to your words and say, "Here's the text." Then, the `UILabel`, not you, deals with the text.

You will do likewise when you want to display an image on your iPhone/iPad. You call `UIImageView`, and when it "arrives," you point to your photograph or picture and say, "Here's the image." Then, the `UIImageView` code, not you, deals with the picture.

Perhaps you're asking yourself what the names of these pointers are, or need to be. The good news is that you can give them whatever names you want. Let's point the `UILabel` to `*label` and the `UIImageView` to a pointer with the name of `*uiImageView`.

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

@interface testViewController : UIViewController {
IBOutlet UILabel *label;
IBOutlet UIImageView *uiImageView;
}

@end
```

Some of the clever people at Apple describe their reasoning for creating and coding IBOutlets as giving a hint to Interface Builder as to what it should "expect" to do when you tell it to lay out your interface.

- One `IBOutlet` whispers into Interface Builder's ear that the `UILabel` class is to use text indicated by the `*label` pointer.
- The other `IBOutlet` whispers into Interface Builder's ear that the `UIImageView` class is to use the image referenced by the `*uiImageView` pointer.

But we're not done yet. After we tell Interface Builder what to expect, we need to tell your Mac's microprocessor—through the compiler—that an important event is about to descend upon it. One of the most important things your compiler wants to know is when an object is coming its way. This is because objects are independent masses of numbers and symbols that weigh upon the microprocessor and put significant demands on it, and so the processor needs to be told by you, the programmer, when it needs to catch the object and put it into a special place in memory.

Objects can come in a wide variety of flavors—as conceptually different as *bird*, *guru*, *soccer*, and *house*. So, to allow the processor to handle its job when the time comes, we need to inform it that each object we will be using in our code has two specific and unique parameters or features: *property* and *type*.

Don't freak out! Providing this information is really easy, and it consists of two steps.

The first step is what we just covered: we give the compiler a head's up about objects we will be using by defining their two specific and unique features, property and type. The second step is this: when the microprocessor receives this data, it utilizes this information by synthesizing it.

To restate:

1. First, we declare that our object has a *property* with a specific *type*.
2. Second, we instruct the computer to implement—or *synthesize*—this information.

We tell the compiler about our object by declaring it, including specific descriptive parameters. Then, we give the compiler the go-ahead to implement our object by telling it to synthesize the object.

But how do we do this declaring and implementing? We use tools in our code called *directives*. We signal directives by inserting "@" before stating our directive. This means that to declare what property our object has we put the "@" symbol in front of the word "property" to make it a property directive: `@property`.

When we see `@property` in our code, we know it's a property directive. Similarly, when we want to tell the compiler to process and synthesize, that is to do its stuff on our object, we put the "@" symbol in front of our synthesis statement: `@synthesize`.

Saying the exact same thing we said before, but translated into geekspeak, we get:

1. The `@property` directive declares that our object has a *property* with a specific *type*.
2. The `@synthesize` directive *implements* the methods we declared in the `@property` directive.

Easy, huh? OK, just two more points now, and then we'll get back to our code.

The first elaboration I want to make regarding step 1 is that we also need to specify whether this property will be *read-only* or *read-write*. In other words, we need to specify whether it will always stay the same or whether it can mutate into something new. In geekspeak, we call this "mutability." For the most part, we will use Apple code to handle the mutability of properties with respect to our objects.

Properties: Management & Control

In order to instruct the Apple code to handle the mutability property, we'll designate the property as "nonatomic." To apply this term meaningfully, try contrasting "nonatomic" with "atomic." Recall that "atomic" means powerful, and it implies the ability to go into the microscopic world and to effect change. Therefore, "nonatomic" must mean not-so-powerful, more superficial, and unmanipulable.

If we designate a property (such as mutability) as *nonatomic*, we are basically saying, "Apple, please handle our mutability and related stuff—I really don't care. I'll take your word for it!" At a later date, you may want to take direct control of this property, and then you *would* designate it as "atomic." At this time, though, we will use the more relaxed approach and let Apple handle the microscopic business. So, when it's time to choose one or the other designation, just use *nonatomic!*

The second elaboration I want to make regarding Step 1 deals with *memory management*. We need to address the issue of how to let the iPhone/iPad know, when we store an object, whether it shall be *read only* or *read-write*. In other words, we need to be able to communicate to the computer the nature of the memory associated with an object—in terms of who gets to change it, when, and how. Generally speaking, we will want to control this information, and keep it in our own hands—that is, to *retain* it. As you move through the remaining exercises in this book, we are going to keep the code in our own hands; we will retain the right to manage our memory.

We can summarize the addition of these details to the property directives, and how we would modify the code, as follows:

- The `@property (nonatomic, retain)` directive says the following:
  - Mutability should be nonatomic. Apple, please handle this!
  - Memory Management is something we want to retain. We will maintain control.
- The `@synthesize` directive implements the methods we declared in the `@property` directive.

We have one more layer of complexity to add to this mix. We add those directives in two different files. We define the `@property` directive with a statement in the header file, and then we implement it by using the `@synthesize` directive in our implementation file.

- Header File: `helloWorld_005_ViewController.h`
  - `@property (nonatomic, retain)` "our stuff"
  - Implementation File: `helloWorld_005_ViewController.m`
- The `@synthesize` "our stuff" defined in `@property`.

We will need to write two of these for each of our two `IBOutlets`: one for the text, and the other for the picture. And, because we're still in the header file, we need to repeat this when we synthesize it in the implementation file.

OK, time to go ahead and enter your code:

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

@interface testViewController : UIViewController {
IBOutlet UILabel *label;
IBOutlet UIImageView *uiImageView;
}

@property (nonatomic, retain)
@property (nonatomic, retain)

@end
```

Yes, that seemed like a lot of explanation just to say: `@property (nonatomic, retain)`. Remember, though, that we're deep in the trenches … we're telling the computer that

we want Apple to take care of mutability, but that we want to retain control of the memory. Later, we will synthesize these commands in the implementation file, for both IBOutlets.

IBOutlets? Remember them? Oh yeah—let's return to that part of your program.

The IBOutlet for the text is UILabel with pointer *label, so enter the code for the text:

```
IBOutlet UILabel *label;
```

The result will look like this:

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

@interface testViewController : UIViewController {
        IBOutlet UILabel *label;
        IBOutlet UIImageView *uiImageView;
}

@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, retain)

@end
```

The IBOutlet for the picture is UIImageView with pointer *uiImageView, so enter the code for the picture:

```
IBOutlet UIImageView *uiImageView;
```

```
The result will look like this:#import <UIKit/UIKit.h>

@interface testViewController : UIViewController {
        IBOutlet UILabel *label;
        IBOutlet UIImageView *uiImageView;
}

@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, retain) IBOutlet UIImageView *uiImageView;

@end
```

Are we done with the header file yet? Not quite. We need to add our IBActions. We've handled our IBOutlets, both of them, but now we're going to utilize an IBAction for something we really need here. Can you guess?

Adding IBActions

Yes, we need a button! So let's make an IBAction for a button. We could "go deep" again, into the code for the IBAction, but this has been a challenging section. Let's save the technical part of this element for "Digging the Code."

Meanwhile, just enter the new code that is highlighted here. See if you can anticipate the functions of the different pieces—or parameters—and we'll see how close you are later.

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

@interface testViewController : UIViewController {
        IBOutlet UILabel *label;
        IBOutlet UIImageView *uiImageView;
}

@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, retain) IBOutlet UIImageView *uiImageView;

- (IBAction)buttonPressed:(id)sender;

@end
```

Before exiting, copy (⌘C) the `IBAction` line:

```
 - (IBAction)buttonPressed:(id)sender;
```

and save your work by entering ⌘S. Of course, saving your work as you go is always a good idea!

Do you recall the reason for doing this? We just copied the `IBAction` method (for our button) so you can paste it into the implementation file, to save time as we always like to do.

OK, we're done with our header file! Go out now, and take a break.

Coding the Implementation File

Now that you're back from your much deserved break, let's continue. It's time to open up the `helloWorld_005ViewController.m` implementation file.

1. Use ⌘V to paste the line of code that you copied at the end of the previous section:

```
- (IBAction)buttonPressed:(id)sender;
```

The result should look like this:

```
#import "helloWorld_005ViewController.h"

@implementation helloWorld_005ViewController

- (IBAction)buttonPressed:(id)sender;
```

In the header file, we told the computer that we are going to do some action(s) when a button is pressed. Now that we've pasted this set of commands into the implementation file, we replace the ";" with a set of brackets. It is inside these brackets that we will tell the computer what needs to be implemented when the button is pressed.

```
#import "helloWorld_005ViewController.h"

@implementation helloWorld_005ViewController

- (IBAction)buttonPressed:(id) sender{

}
```

2. Before we get into the code that will spring into action each time the user presses the button, we need to take care of the second half of the synthesis we began in the previous section. Do you remember how we defined our `@property` directive with a statement in the header file, and how we left the corresponding `@synthesize` directive for our implementation file? Specifically, we had to have one `@property` directive for the `UILabel` and another for the `UIImageView`, as follows:

```
@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, retain) IBOutlet UIImageView *uiImageView;
```

What this means is that we merely need to have an `@synthesize` statement for both the pointers, one for the `UILabel` and another for the `UIImageView`. We named the pointer for the `UILabel` named `label`, and we named the pointer for the `UIImageView` named `uiImageView`.

We can put them both on the same line, so let's do it:

```
#import "helloWorld_005ViewController.h"

@implementation helloWorld_005ViewController
@synthesize label, uiImageView;

(IBAction)buttonPressed:(id) sender{

}
```

We've used a simple overlay of photograph and text to get you partially familiarized with a terribly difficult concept. Hey—if your head is spinning, don't feel bad! Wrapping your brain around *pointers*, *objects*, and *synthesis* is not trivial. Unfortunately, there is no way around it in an iPhone/iPad app programming lesson. In the class in which I taught this exact assignment, there were a couple of computer science students with backgrounds in C++, C#, and Java who had a more difficult time than some of the liberal arts students in the class who just faithfully accepted that:

- The `@property` directive declares that our object has a *property* with a specific *type*.
- The `@synthesize` directive *implements* the methods we declared in the `@property` directive.

The advanced students had so much stuff in their heads that it took some of them longer than the newbies to accept these two statements. We'll be going over these points again and again, so let's move on for now. These ideas can ferment in the back of your brains, and tomorrow these will make more sense.

Providing for Synthesis

When the user runs this app and presses the button, the image of my wife, Kera, will appear instantly on top of the background staircase. She is going to "say" something via the embedded text. How about, "Hello World, I'm back!"

1. To accomplish this, we need to associate the label instance variable with a text property assigned with our desired text as follows:

```
#import "helloWorld_005ViewController.h"

@implementation helloWorld_005ViewController
@synthesize label, uiImageView;

- (IBAction)buttonPressed:(id) sender{
label.text = @"Hello World, I'm back!";
}
```

2. Having completed the task of coding for the text, we now need to add the code that will cause the image of Kera to appear. For this, we will use a *class method* called `imageNamed` that will display the `kera.png` image, the top-layer photo we prepared at the beginning of this project. Enter the line bolded in the following code, immediately under the code you just entered for the text.

```
#import "helloWorld_005ViewController.h"

@implementation helloWorld_005ViewController
@synthesize label, uiImageView;

- (IBAction)buttonPressed:(id) sender{
label.text = @"Hello World, I'm back!";
UIImage *imageSource = [UIImage imageNamed: @"kera.png"];
}
```

> **NOTE: :** On the corresponding video segment, instead of entering this line of code, one of us inadvertently typed:
>
> `"ui"Image *imageSource = [UIImage imageNamed: @"kera.png"];`
>
> This, of course, caused an error. We went back and subsequently changed it to reflect the correct code as just shown. For fun, however, we kept this mistake on the video to illustrate how to see a "fail," go back, find it, and correct it.

3. Our pointer's name for the image is `uiImageView`, but right now the `kera.png` image file is in UIImage's assigned pointer called `imageSource`. We need to assign this pointer to the image of `uiImageView` as shown in the following code:

```
#import "helloWorld_005ViewController.h"
```

```
@implementation helloWorld_005ViewController
@synthesize label, uiImageView;

- (IBAction)buttonPressed:(id) sender{

label.text = @"Hello World, I'm back!";

UIImage *imageSource = [UIImage imageNamed: @"kera.png"];
uiImageView.image = imageSource;
}
```

If this doesn't quite make sense at the moment, that's OK. There sure are a lot of entities with "image" as part of their name, object, or association, and it *is* confusing. We'll be examining this topic more thoroughly in Chapter 7, so, right now, don't lose any sleep over it! Figure 5–8 illustrates how your code should appear at this point.

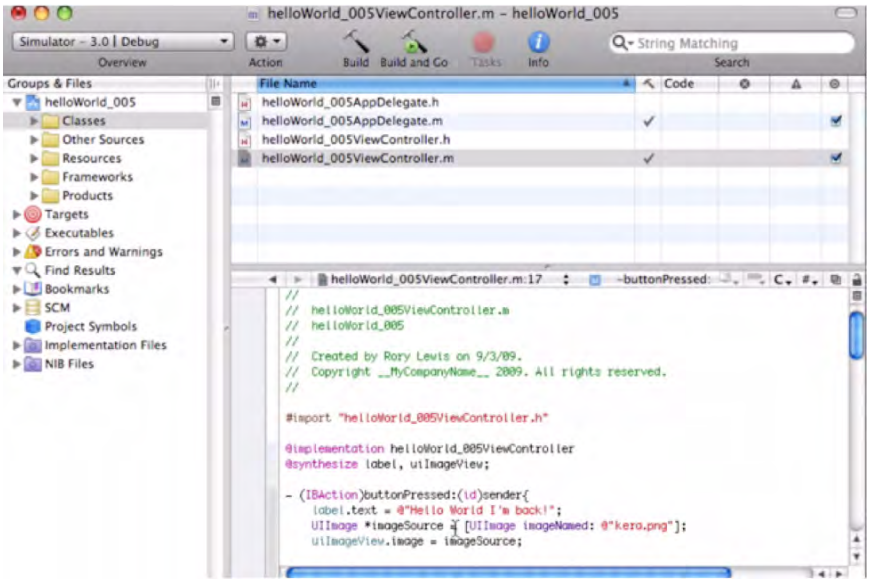


Figure 5–8. *This is how your code should appear, after you've completed the Synthesize and Button actions.*

Now save your work by entering ⌘S, and give yourself a pat on the back. You have worked through the header and implementation files at a much deeper level than in previous chapters. Even though you have walked through some of these technical functions before, you braved them again while remaining open to a deeper understanding. Also, you tackled a very difficult concept: *synthesis*.

Interface Builder: Making the Connections

Our code is finished. Now, we just need to place all of our images into the `Resources` folder, assign the icon, and then move to Interface Builder and hook things up. *No problem!*

1. So that the pointers that you have just coded have something to which to refer, you need to place some files at the appropriate address. Therefore, drag your first image (i.e., bottom layer photo) into the `Resources` folder.

> **NOTE: :** When the folder highlights, it means the object is selected. Focus on where your cursor is—that is the point at which the folder will react. Once it highlights, drop the object in by releasing the mouse.

2. After dropping the image into the `Resources` folder, you will be prompted to define whether the image will always be associated with its position on your desktop or will be embedded with the code and carried along with the application file, as shown in Figure 5–9.

   We want it to be embedded, of course, so click the "Copy the items into destination's group folder" box. Also, click on the "Recursively create groups for any added folders" box. Then click Add (or press Enter).

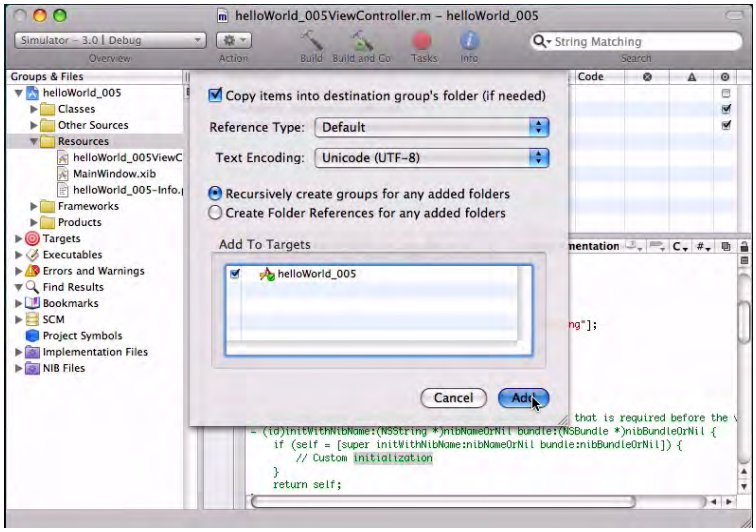


Figure 5–9. *Click in the "Copy items into destination group's folder (if needed)" box to ensure that all related images follow along with the code. Also, click in the "Recursively create groups for any added folders" box.*

3. Because these actions need to be applied to all three of our images, move the other two into the "Resources" folder just as you did this one.

4. We created an icon image file called `icon.png`. We want this one to show up on the iPhone/iPad, rather than the generic one. To do this, double-click on the info.plist file in the `Resources` folder, as shown in Figure 5–10, and then double-click on the Icon file Value cell. In that space, enter the name of your icon file: `icon.png`. Now, save your work.

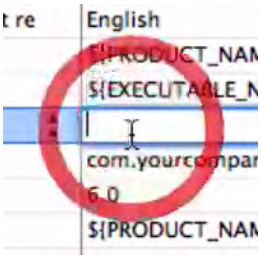


Figure 5–10. *Associate your custom icon with the info.plist.*

The *plist* (property list), by the way, is another area that we will explore later. For now, we're ready to move on to Interface Builder in order to connect and associate various pieces of our puzzle.

5. Remember that our top-layer image will be placed over the base layer when the user pushes the button. Therefore, we want to handle the base layer in the same manner as in the previous chapter when you used a wallpaper image of yourself. Scroll down in your Library to the Cocoa Touch item folder and locate the Image View icon. Drag one onto your View frame. Refer to Figure 5–11.

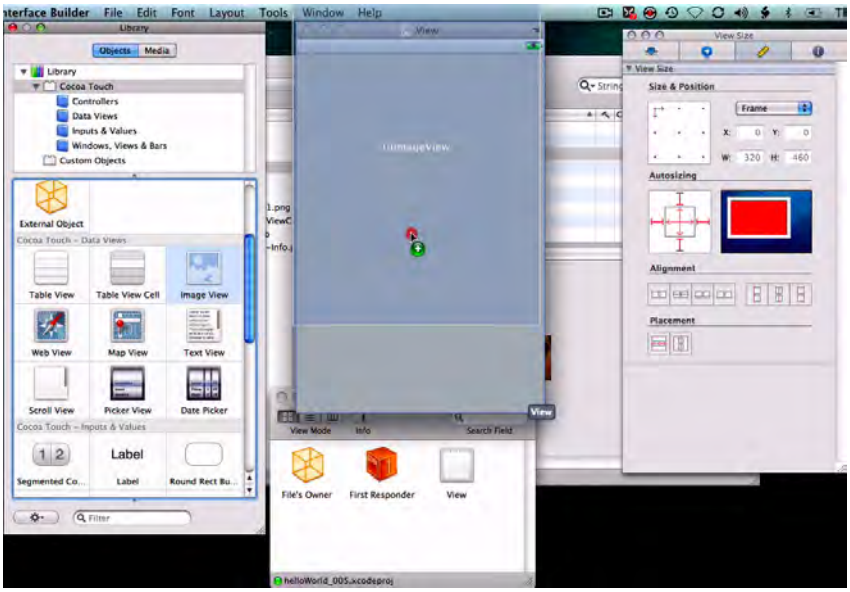


Figure 5–11. *Open Interface Builder and drag an Image View onto the View frame.*

6. We want to connect `320_480_stair.png` to our Image View so that it will appear. Go to the Information tab of the Image View Attributes window, open the drop-down window, and select the image as shown in Figure 5–12.

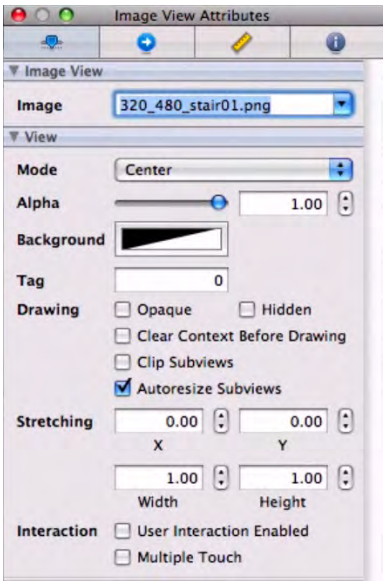


Figure 5–12. *Assign the `320_480_Stair.png` file to the Image field at the top of the Image View Attributes frame.*

7. Earlier, we decided that when the user presses the button, Kera Lewis should appear and announce, "Hello World, I'm back!" We decided that the method we would employ would be a *label instance variable*—with a text property assigned with "Hello World, I'm back!" as follows:

```
#import "helloWorld_005ViewController.h"

@implementation helloWorld_005ViewController
@synthesize label, uiImageView;

- (IBAction)buttonPressed:(id) sender{
label.text = @"Hello World, I'm back!";
UIImage *imageSource = [UIImage imageNamed: @"kera.png"];
uiImageView.image = imageSource;

}
```

So, drag out a label that will be our instance variable, and we will assign the text “Hello World, I’m back!” onto the Base View. When you put it there, repeat the way that you adjusted the size in the earlier assignments, which is to widen it so it can fit this text. And, as you’ve done before, center the text and change its color to white in the Properties frame.

We want the picture and the text to appear when a button is pressed—so we need a button. Go ahead and drag one onto your base layer, and, in its title field enter “Guess who’s on campus?” as shown in Figure 5–13.

When users see a button asking this question, they will be compelled to press it. When they do, we want Kera to appear saying, “Hello World, I’m back!”

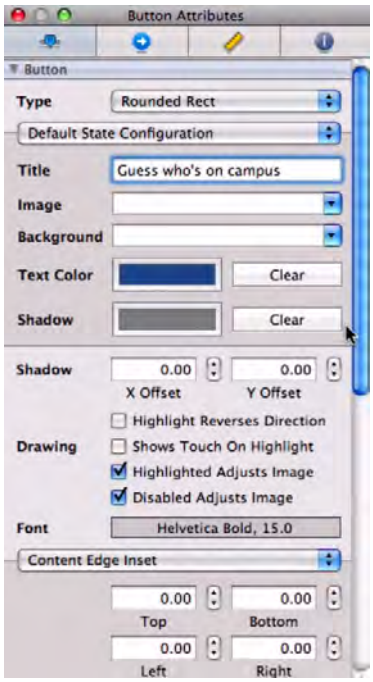


Figure 5–13. *Drag a Button onto the background layer “View” frame.*

You may want to adjust the size of the button as we’ve done before. Of course, we don’t want the button to be like some blob on top of the picture; we want it to look pretty cool and show some of the underlying image. While still in the Image View Attributes window, scroll down and shift the Alpha slider to about 0.30.

8. Again, we're using two `IBOutlets`, and each category "whispers" something to Interface Builder. One says that we want a `UILabel` class to use text to which the pointer `*label` points; the other says that the `UIImageView` class will put up an image located at a place to which the pointer `*uiImageView` points.

   Well, what have we done so far in Interface Builder? We've installed the background image and inserted a button that will trigger these two `IBOutlets`. Now, recall that, earlier, you entered

```
- (IBAction)buttonPressed:(id) sender
```

   in the implementation file. This line, in fact, invoked our two friends, the label, with

```
label.text = @"Hello World, I'm back!";
```

   and the image, with

```
UIImage *imageSource = [UIImage imageNamed: @"kera.png"];
```

   In fact, while working in the header file earlier, you set this all up by declaring the label with

```
IBOutlet UILabel *label
```

   and declaring the image with

```
IBOutlet UIImageView *uiImageView
```

   Then, you synthesized it correctly with your two "@property" statements for the label:

```
@property (nonatomic, retain)
IBOutlet UILabel *label
```

   and for the image:

```
@property (nonatomic, retain)
IBOutlet UIImageView *uiImageView
```

   Also, you synthesized it with

```
@synthesize label, uiImageView
```

   So, that means we're ready for action. Because we've just created the button that will call our two friends, all we need to do now is create the image and the label, then associate them with the appropriate pieces of the code.

When the button is pressed, the `kera.png` image has to arrive. On what does it arrive? It's carried onto the screen by way of an Image View. Therefore, drag an Image View onto the screen, as shown in Figure 5–14.

Figure 5–14. *From the Cocoa Touch—Data Views tab, select and drag an Image View onto your work space.*

9. After you have dragged an Image View onto the screen, we want to place it flush to the bottom edge of the iPhone/iPad screen. We don't want the image floating in the middle of the screen, but instead to appear as if it's projecting from the bottom. Once you've dragged the image to the screen, just let it go. We have not yet configured the size or placement of the image. That's next!

Go to the Image View Application dialogue frame and then click on the View tab. Here, you will see that the alignment option of Center is checked by default. We want to change that to Bottom, as illustrated in Figure 5–15.

Before moving onto the next step, take a minute to align the label and button with one another, and in context to the center of the screen.

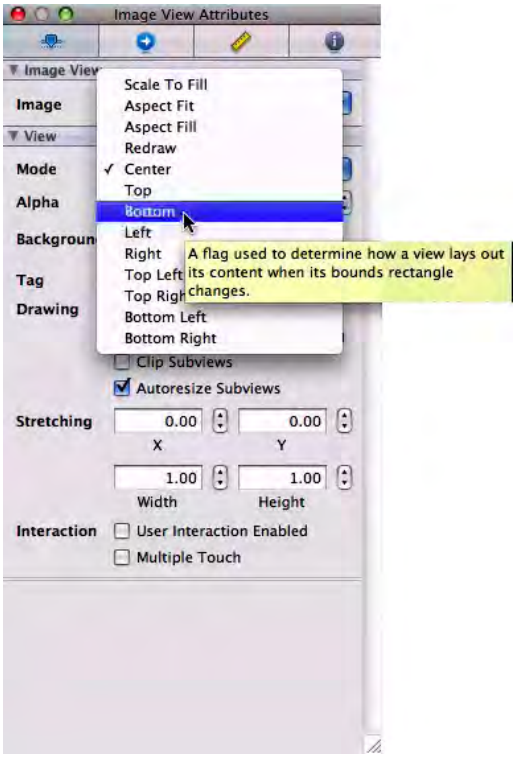


Figure 5–15. *Position the* `UIImageView` *onto the View screen, flush with the bottom.*

10. As depicted in Figure 5–16, you now want to connect the File's Owner to the Label. You'll do this just as you've done in the past—by ⌘+dragging from your File's Owner icon to the button.

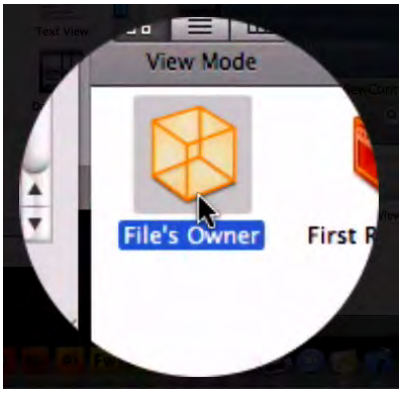


Figure 5–16. *Select the File's Owner icon to begin establishing the various connections.*

11. Once you get to the label, select the Label selection on the black drop-down menu, as shown in Figure 5–17.

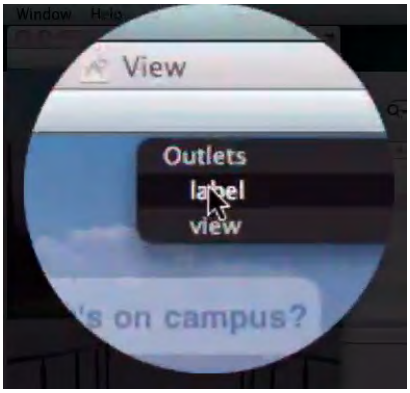


Figure 5–17. *Once you get to the button, select the "label" option from the pull-down menu.*

In earlier exercises, you blindly connected to whichever selection I asked you. However, as I mentioned before, here in Chapter 5 we're digging a little deeper into the code. So without getting completely buried in compiler details, I just want you to get a taste of some beautiful aspects of the code as it makes selections and connections in Interface Builder. Earlier, we associated the `UILabel` with a pointer called `*label`. We also made a pointer called `*uiImageView` that directed the `UIImageView`:

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

@interface testViewController : UIViewController {
IBOutlet UILabel *label;
IBOutlet UIImageView *uiImageView;
}

@end
```

So, we connect the File's Owner to the User Interface called `label`. There's actually a lot more that goes on here, but, for now, it's important that you see how we associate the `label` and the image view by virtue of connecting them with the previous lines of code.

12. After selecting the "label" option, perform another ⌘ drag from the Button to your File's Owner as shown in Figure 5–18, and then select the buttonPressed option. Again, in the past we just skipped over what this meant, but let's see if we can get some ganglia to twitch with excitement as we consider these connections.

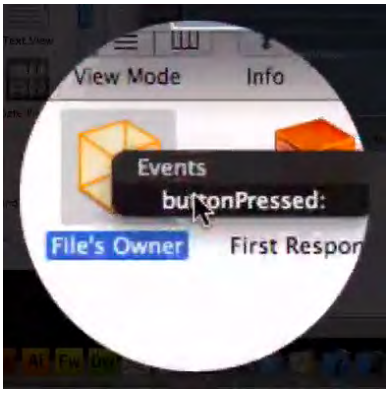


Figure 5–18. *Select the* `buttonPressed` *option.*

Final Step: File's Owner & uiImageView

Earlier, you wrote the following code:

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

@interface testViewController : UIViewController {
IBOutlet UILabel *label;
IBOutlet UIImageView *uiImageView;
}

@property (nonatomic, retain) IBOutlet UILabel *label;
@property (nonatomic, retain) IBOutlet UIImageView *uiImageView;

- (IBAction)buttonPressed:(id)sender;

@end
```

Before you exited, I suggested that you copy the line

```
-(IBAction)buttonPressed:(id)sender;
```

Then, you opened your implementation file and pasted this line in place. You replaced the semicolon with brackets, within which you inserted all the stuff you wanted the button to do when pressed by the user.

```
#import "helloWorld_005ViewController.h"

@implementation helloWorld_005ViewController
@synthesize label, uiImageView;

- (IBAction)buttonPressed:(id) sender{
label.text = @"Hello World I'm back!";
UIImage *imageSource = [UIImage imageNamed: @"kera.png"];
uiImageView.image = imageSource;

}
```

In a broad sense, I want you to see that the two items to be invoked, the Label and the Image, are inside `buttonPressed`, and that you connected the stuff inside the brackets to the Button.

We've already connected the Label, so all we have left to do is to connect the File's Owner with the second of the two items: our picture. Here we go …

1. We now need to ⌘ drag from the File's Owner icon to the Image View and connect it to the uiImageView option in the drop-down menu, as shown in Figure 5–19. Save your work (⌘S); quit Interface Builder (⌘Q).

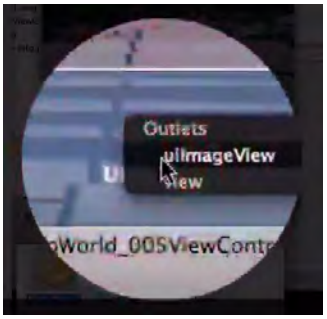


Figure 5–19. *Connect the File's Owner icon with the uiImageView in the pull-down menu.*

2. Enter ⌘0 to run the code.
3. Press the button that reads "Guess who's on campus?" As shown in Figure 5–20, my lovely wife, Kera, magically appears, and she says, "Hello World, I'm back!"

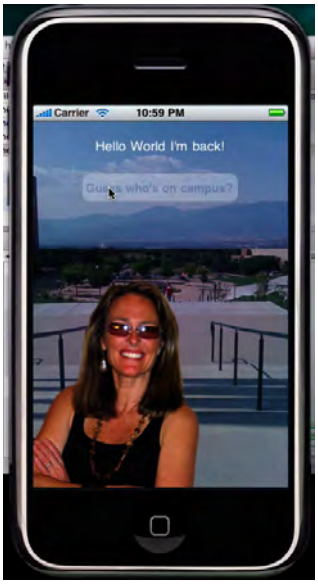


Figure 5–20. *The top-layer image and text appear when the button is pressed.*

In Figure 5–21, you can see that the iPad view of the iPhone Simulator correctly displays our base image. The button is just begging for the user to push it … to find out who *is* on campus.

Figure 5–21. *The iPad Simulator displays the iPhone-sized images in sharp detail.*

As you can see in Figure 5–22, the same image is portrayed at the “2x” enlargement – which is actually *four times* the area. Note the button in the lower right now reads “1x”, for the only zooming option open to the user now is to return to the original size that was fitted, for the purposes of this exercise, primarily for the iPhone (320 x 480 pixels).

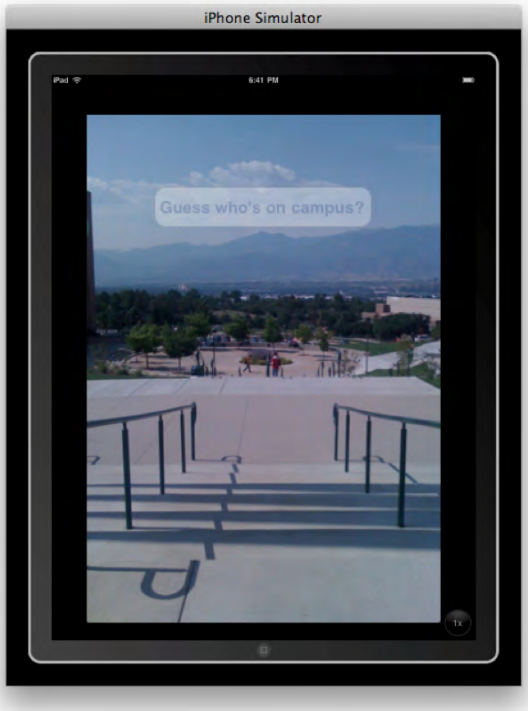


Figure 5–22. *By pressing the “2x” button in the lower right-hand corner, you can see the Simulator gives this expanded view. Note that the button label changes to give the reducing option: “1x.”*

The final image for this exercise, Figure 5–23, shows the enlarged version of the overlay with accompanying text. Kera announces her return to university life at the click of a button.

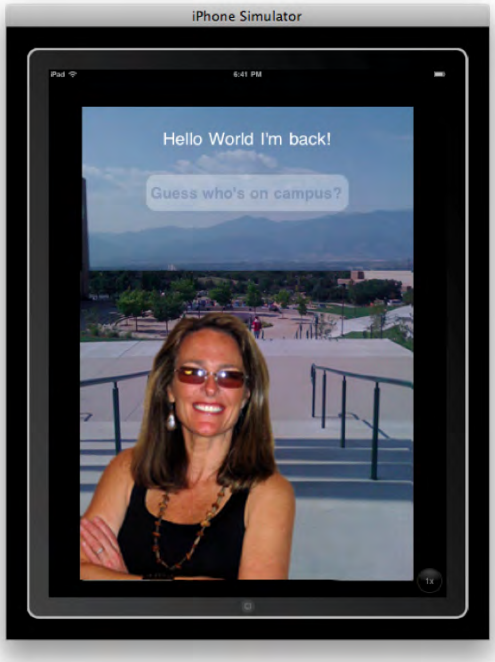


Figure 5–23. *The iPad image of the resultant overlay of views: "Hello World, I'm back!"*

Congratulations! This was your most demanding program so far. We delved into several technical areas in the code, but you survived. If you'd like, we can dig even further. Or, if you'd rather not, skip the "Digging the Code" section and proceed to the next chapter.

Digging the Code

In this section, let's zoom into some of the key components that we encountered earlier in this chapter. I want to talk a little more about IBOutlets and IBActions – specifically how these include keywords … and even *quasi*-keywords. We'll also touch on pointers and their relationship to addresses in the code.

IBOutlets and IBActions

Earlier, we worked with `IBOutlet` and `IBAction` keywords, and now we're going to talk about a couple of related concepts. Strictly speaking, these are considered by many programmers to be "quasi-keywords."

The Appkit of Objective-C has converted original C language preprocessor directives, such as `#define`, into usable preprocessor directives. In geekspeak, we would pronounce this as "pound-define."

> **NOTE:** In the US, the "#" sign is often called the "pound" sign, especially in Objective-C and other programming contexts. In the United Kingdom, it is referred to as the "hash" character. Many iPhone/iPad developers have recently begun to refer to the `#define` preprocessor directive simply as the "define directive."

The `#define` preprocessor directive tells the computer to substitute one thing for another. That's an easy concept, right? For example, if I were to program the computer to substitute "100" every time it sees an instance of your name, our code in C would look like this:

```
#define yourName 100
```

This would tell the computer to substitute "100" each time it processes `yourName` – a variable that recognizes instances of your actual name.

Back to Xcode now, and our topic. In this context, the `IBOutlet` and `IBAction` quasi-keywords aren't really defined to be anything. In other words, they don't do anything substantial *for the compiler*, which is the core of the computer.

Quasi-keywords are flags, though, and they are important in the communication with the Interface Builder. When it sees the `IBOutlet` and `IBAction` quasi-keywords, it gets some of its internal code ready to perform specific tasks. It gets itself ready to deal with instance variables and all the hooks and connections that we make in that programming arena.

More About Pointers

It's difficult for many programming students to understand the concept of "pointers"—also sometimes known as the concept of *indirection*. It's not easy to explain this idea because it's one of the most sophisticated features of the C programming language.

Earlier in this chapter, I presented the analogy of seeing a criminal doing something, and then calling the police and pointing the police to where he is—so they, not you, can arrest the criminal. This analogy works for many students, but now let's go a little deeper.

If you were to ask a Computer Science professor what a "pointer" is, he would probably say something like "Pointers hold the address of a variable or a method."

"The *address*?" you ask. Well, consider this new analogy in the way of explanation.

Have you ever seen a movie in which a detective or some frantic couple is traveling all over the place, looking for clues to the treasure map, or the missing painting, or the kidnapped daughter? Sometimes they will spot a fingerprint, or a receipt, or even an envelope with a piece of paper containing a cryptic message—and these take the people one step closer to their goal – of finding the missing objects themselves.

We can call these pointers; they indicate the next place to go – for the solution of the given problem. They don't necessarily give the ultimate address, at which everything is handled and resolved, but they give us intermediate addresses or places to continue our work.

Thus, what the professor of Computer Science means is that pointers do not actually contain the items to which they direct us; they contain the locations within the code—the addresses—of the desired objects or actions or entities. This important feature makes the C-family of languages very powerful.

This simple idea makes it very efficient to turn complex tasks into easy ones. Pointers can pass values to types and arguments to functions, represent huge masses of numbers, and manipulate how we manage memory in a computer. Many of you are perhaps thinking that pointers are similar to variables in the world of algebra. *Exactly!*

In our first analogy, a pointer enabled an unarmed citizen to arrest a dangerous criminal by using indirection—that is, by calling the police to come and solve the problem. (Yes, the term "indirection" is an odd choice given that we are actually being *directed* toward the goal.)

Consider the following example where we use a pointer to direct us to the amount you have in your bank balance. To do this, let's define a variable called `bankBalance` as follows:

```
int  bankBalance = $1,000;
```

Now, let's throw another variable into the mix and call it `int_pointer`. This will allow us to use indirection to indirectly connect to the value of `bankBalance` by the declaration:

```
int  *int_pointer;
```

The star, or asterisk, tells the family of C-languages that our variable `int_pointer` is allowed to indirectly access the integer value of the amount of money in our variable (placeholder): `bankBalance.`

To close, I want to remind you, and to acknowledge, that our digging around here is not an exhaustive or rigorous exploration into these topics … just a fun tangent into some related ideas. At this point, there is no reason for you to be bothered if you don't fully understand pointers. Seeds have been planted and that's what counts for now!

In the Chapter Ahead

In Chapter 6, we will move into the next level of complexity: switch view applications. We will examine how a team of characters or roles within your code will work together to direct an outcome, or series of outcomes, that will give the user the sense of seamless flow.

You will learn about delegators and switch view controllers, classes and subclasses, and "lazy loads." We will get into the nitty-gritty of the `.xib` files, examine the concept of memory deallocation, and learn about imbedded code comments. It's getting curioser and curioser …

Onward to the next chapter!

Chapter 6

Switch View with Multiple Graphics

In this chapter, we will explore one of the most remarkable aspects of the iPhone: its unique ability to switch seamlessly between one view and another. We have all seen the wonderful iPhone and iPad ads on television, in which a person's fingers direct an amazing flow of vivid images, within interactive applications, and cause one view to just slide or roll directly into another, giving the impression of performance art. The concept behind this is what Apple calls *Switch View* methodology. As a professor of computer science, I have learned of several pitfalls regarding the teaching—and learning—of the Switch View methodology.

I intend to take advantage of the interesting variety of results that I have experienced as I have presented my students with some of the shortcuts those smart people at Apple have given us. As I introduce new concepts and techniques, I will be contrasting several tried-and-true pathways to the goal of creating an app with the capabilities I've described. Specifically, you will become familiar with the Tab Bar application template; it creates an Xcode SDK that lays everything out for a simple tab bar look and feel. For our goal of coding switch views with multiple graphics, you will barely have to program anything—the boilerplate code within the template does most of the work!

Ironically, this "easier and softer way" was a disaster for some of my students. I now realize that when too many shortcuts are taken, without due exploration of the underlying code, some programming students get confused. I knew that implementing switch views in Objective-C was by no means a trivial undertaking, and I had originally supplied my class with a way of simply pulling levers via the Tab Bar Application template. When this didn't have the positive results I had hoped for, I went back to the drawing board.

I held a Saturday workshop in which I divided the lesson into three sections—each of which would be a variation on my original plan. Each of the three alternative methods would use the same two pictures as depicted in Figure 6–1.

NOTE: One thing you may have noticed already is that this is a *long* chapter. This is due to the point I just made – that this is actually a three-in-one chapter. Because it really will be instructive to show you the long, detailed way; the short, no-brainer way; and the combination way … to meet the same basic objective, I decided to keep this all in the same chapter. Nevertheless, I expect you to pace yourself … *breaks are important!* Please treat Chapter 6 as a long sub-chapter, followed by a short sub-chapter, then followed by a medium sub-chapter. *Happy Coding!*

As you can see, there are three ways in which we will accomplish our switch view objectives in this chapter. In each scenario, we will start with the first view – a photo of my grandfather as a handsome young bachelor, and then switch to the second view – of my grandfather and grandmother, both of whom raised me as their son. When I was a child, he was a larger than life figure, my first hero, and thus I've labeled this exercise with the nickname of another "smart guy," by which I apparently called him on occasion: Einstein.

The first way we will accomplish the switch view (left) has the "YOU" there because *you*, my good friend, will be writing the code. Note that the buttons/tabs are a bluish color, which is how you will create them.

The second way (middle) is represented with an icon of XCODE that takes us from the one picture to the next. This pathway is where you build the app using the no-brainer Tab View Application template. Note that the tab keys are not bluish in color, but black, since that is the default in the boilerplate code within this utility.

The third way to switch the views (right) represents a combination of the two other approaches. Note that we will use the ready-made black tabs, but there will be areas that you will personally build and customize. This path is a bridge approach, a morph between YOU and Xcode.

So that was how we proceeded on that Saturday morning workshop, and it worked like a charm. One more thought about this three-way agenda: every student still expressed a preference for a given method, but there was unanimous appreciation for the presentation of all three in a side-by-side lesson.

Figure 6–1. *Left, the "long way"; middle, the "no-brainer"; right, the "combination"*

Before We Begin

There are three more points to note before we begin:

Point 1: Don't become too comfortable. Something I realized on the Saturday sessions was that students would gloss over steps they thought were the same, but which were slightly different. So please pay careful attention to steps that may seem familiar to you.

Point 2: Get over using the same pictures. After the weekend was over, one student complained, "*Dr. Lewis, the pictures are boring. Can't we use our material?*" My response to her was, "*We're all using the same 'boring' pictures so you can* better *focus on the mechanics of how these three approaches to creating Switch Views work."* If we keep these two pieces of the puzzle nice and consistent, then we'll be able to stay on task.

Point 3: The three versions and their screencasts. It may be instructive to first see me go through all three methodologies on three separate screencasts. I will walk you through them here in detail, explaining parts that I purposefully skip over in the screencasts for the sake of expediency. Go ahead, then, and check out the three videos. I will meet you back here. Note also the "Correct Code" links for each exercise provided here for your future reference:

1. ***From Scratch***: einSwitch_001

   Video: `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/006_einSwitch_001.htm`

   Correct Code:
   `http://www.rorylewis.com/cCode/006a_einswitch01.zip`

2. ***Tab-Bar***: einSwitch_002

   Video: `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/007_einSwitch_002.htm`

   Correct Code: `http://www.rorylewis.com/xCode006b_enswitch02.zip`

3. ***Custom Tab-Bar***: einSwitch_003

   Video: `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/008_einSwitch_003.htm`

   Correct Code: `http://www.rorylewis.com/xCode006c_enswitch03.zip`

OK … take a deep breath, turn off your cell phone, and calm yourself. The next part of our journey will earn you some bragging rights within the Xcode community. Soon enough, you'll be able to say, "Yeah, I code in Objective-C to perform switch views. I can do it the long way, the short way, or by customizing templates. *Move aside!*"

einSwitch_001—a Window-Based Application

Go back for a moment, if you will, and recall the analogy of the automobile showroom, with six different body styles. As you see from this heading, we are hopping into one of our more versatile and sturdy vehicles: the window-based application. Although we will drive to the same destination in each section of this three-part chapter, we will do a little car hopping. Don't let this distract you; instead, let this be an adventure in test-driving different models.

Preliminaries

As always, let us start off with a clean desktop. Then we will prepare for this first example by adding four items to our desktops to join the ever-present Macintosh HD icon: three .png image files and a text file. As shown in Figure 6–2, download these from the zip file located at `http://www.rorylewis.com/xCode/006_Chapter 6h_EinSwitch01.zip`. Once you have downloaded the zip file, you may need to go to your Downloads folder.

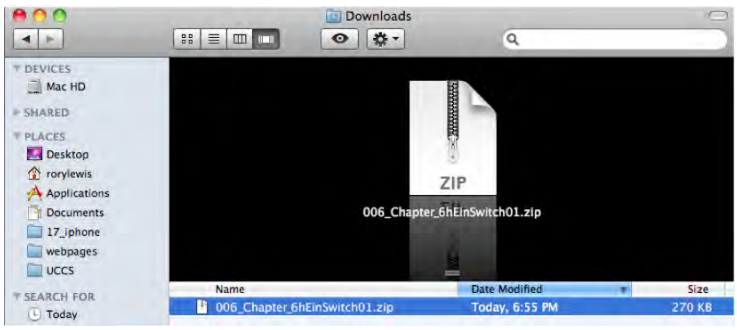


Figure 6–2. *Your downloaded zip file will most likely default to your Downloads folder.*

After you have located the zip file, drag it onto your desktop, as shown in Figure 6–3. Double-click on the file to access the `006_Chapter_6hEinSwitch01` folder and extract the four items onto your desktop, as shown in Figures 6–3 and 6–4.

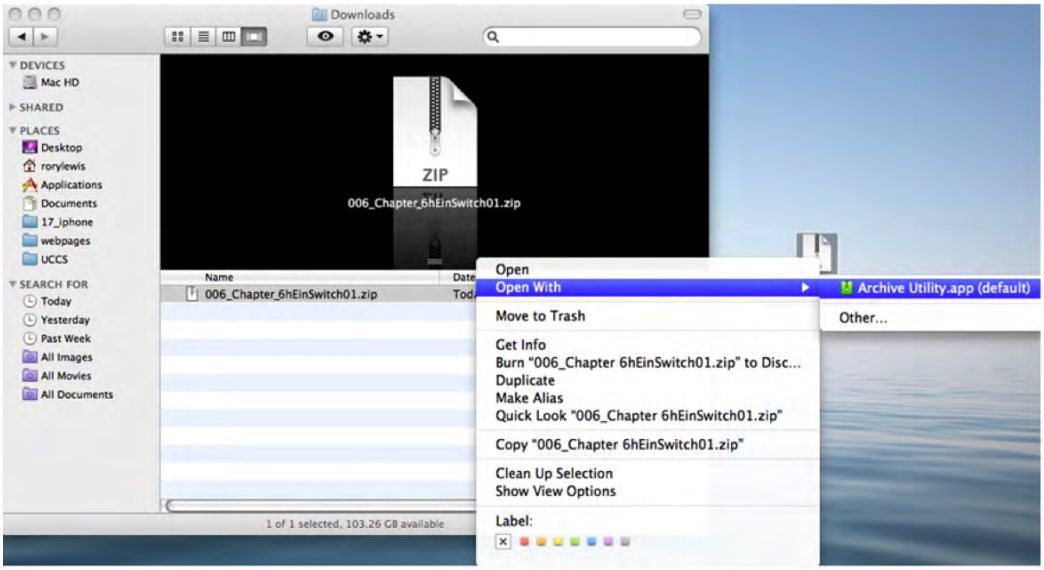


Figure 6–3. *Extract 006_Chapter_6hEinSwitch01.zip from your Downloads folder and move it to your desktop.*

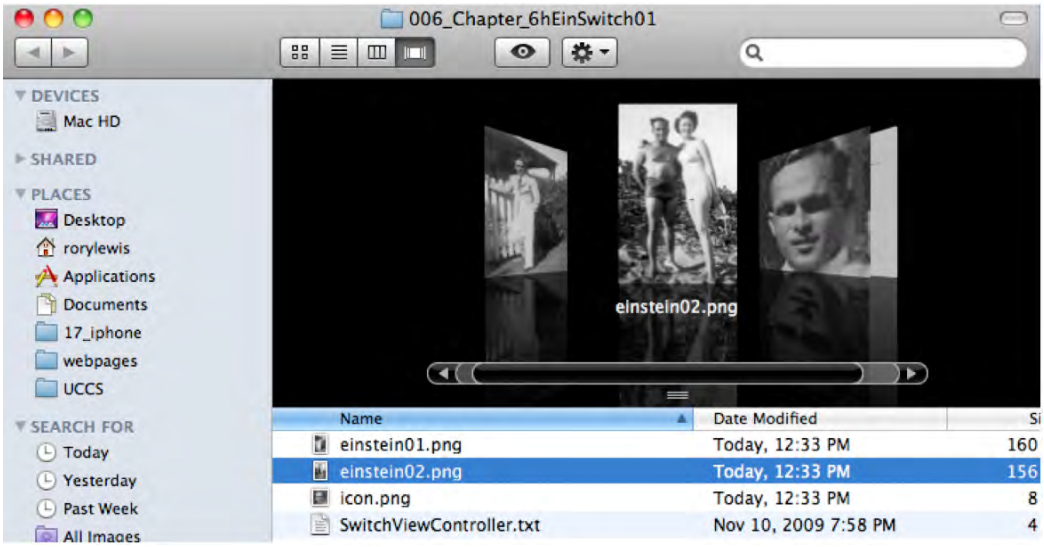


Figure 6–4. *Make a note of the four files located in the 06_Chapte_6hEinSwitch01.zip folder.*

NOTE: I changed the desktop image in Figures 6–3 and 6–4 because the dark background blended into the black background inside the folder. Don't let this confuse you.

The first two items are the black and white photographs of my grandfather, alone and with his bride. As I stated in Point 2, I really want you to use these images I have supplied, for it will make the rest of the chapter flow more easily and be less distracting. One less variable to worry about!

The third item in the 006_Chapter_6hEinSwitch01.zip file is the icon image for our app, with smaller dimensions of course. Once you have all three of these images: the bottom layer – my grandfather as a bachelor (Figure 6–5), the top layer – grandpa and his bride (Figure 6–6), and the icon (Figure 6–7), save them on your desktop.

Figure 6–5. *We'll use this image as the bottom layer of our switch view app.*

Figure 6–6. *Here is the top layer image, a smart switch!*

Figure 6–7. *This smaller image is for the screen icon.*

The fourth item in the zip file is the `SwitchViewController.txt` document, which has boilerplate code that you will see again and again. In this chapter, you will be directed to copy and paste portions of this text into your code to manage certain functions. We will go over it all, and I will explain it to you at length, but, at this point in your coding journey, just save it conveniently on your desktop.

At the point when we do insert this ready-made code, we call it a *Lazy Load*. You can see what it looks like below … *mark this page*. I will be referring back to this code in a little while—when it is time to copy and paste it.

```
#import "SwitchViewController.h"
#import "Ein1Controller.h"
#import "Ein2Controller.h"

@implementation SwitchViewController
@synthesize ein1Controller;
@synthesize ein2Controller;

- (void)viewDidLoad
{
Ein1Controller *ein1Controller = [[Ein1Controller alloc]↩
 initWithNibName:@"Einstein1View"
bundle:nil];
        self.ein1Controller = ein1Controller;
        [self.view insertSubview:ein1Controller.view atIndex:0];
        [ein1Controller release];
}

- (IBAction)switchViews:(id)sender
{
        // Lazy load - we load the Einstein2View nib the first time the button↩
 is pressed
        if (self.ein2Controller == nil)
        {
                Ein2Controller *ein2Controller =
                [[Ein2Controller alloc] initWithNibName:@"Einstein2View"
                                                bundle:nil];
                self.ein2Controller = ein2Controller;
                [ein2Controller release];
        }

        if (self.ein1Controller.view.superview == nil)
                //This is with no animation
        {
                [ein2Controller.view removeFromSuperview];
                [self.view insertSubview:ein1Controller.view atIndex:0];
        }
        else
        {
                [ein1Controller.view removeFromSuperview];
                [self.view insertSubview:ein2Controller.view atIndex:0];
        }
}

-(id)initWithNibName:(NSString *)nibNameOrNil
                        bundle:(NSBundle *)nibBundleOrNil {
        if (self = [super initWithNibName:nibNameOrNil
                                bundle:nibBundleOrNil]) {

        }
        return self;
}
```

```
-(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↩
interfaceOrientation {
return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning];
        if (self.ein1Controller.view.superview == nil)
                self.ein1Controller = nil;
        else
                self.ein1Controller = nil;
}

- (void)dealloc {
        [ein2Controller release];
        [ein1Controller release];
        [super dealloc];
}

@end
```

NOTE: Before we move on, you need to *DELETE* the `06_Chapter_6hEinSwitch01` folder. You've taken the three images and one file out of it, so it's empty. Now delete it. This is critical. If you open up Xcode and save one of your files with the same name as the one above, there will be mass confusion. Drag the empty `06_Chapter_6hEinSwitch01` folder to the trash now, along with the original zip file item, if you haven't already deleted that.

Name your Project "einSwitch01"

Open Xcode and enter ⌘⇧N, as shown in Figure 6–8; this will open the New Project window, in which you will click the Window-based Application template. As you know, your default may not be set to this option, so make sure you enable it. Name your new program "einSwitch01," as shown in Figure 6–9; then save your Window-Based Application to your desktop by selecting ⌘S. I am going to be quite strict about the naming protocol here because we will have three similar applications that will have very similar labels. Your code will "expect" items to be named as I have named them.

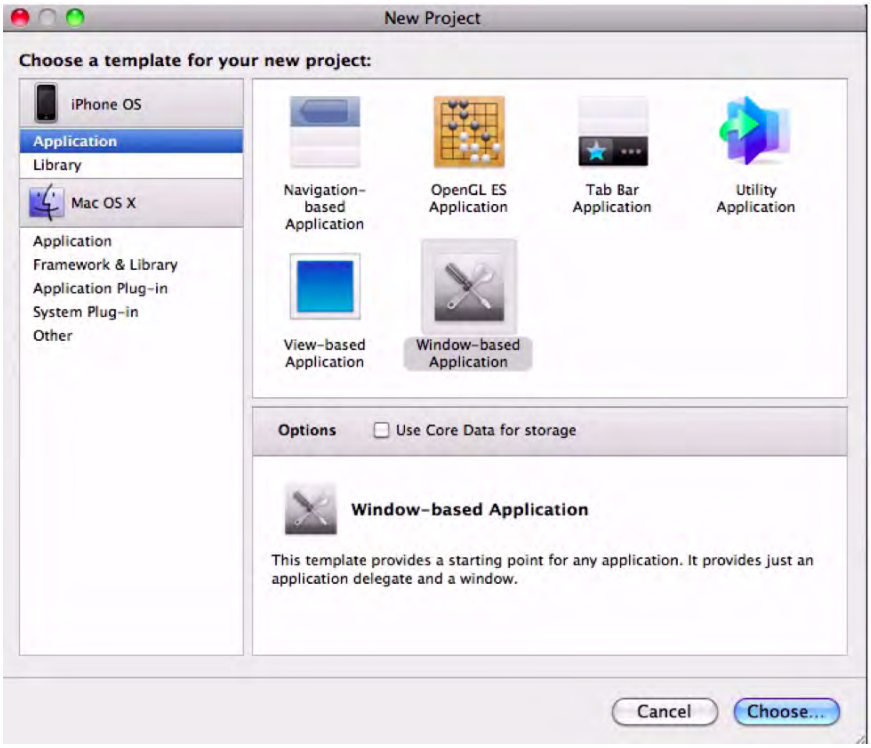


Figure 6–8. *Enter ⌘⇧N to open a new Window-based Application.*

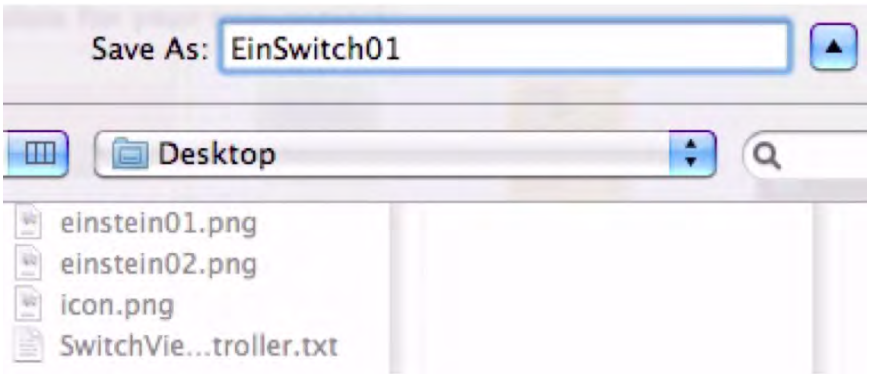


Figure 6–9. *Save your new Window-based Application as "EinSwitch01" to your desktop by entering ⌘S.*

A ROAD MAP: HOW SHALL WE PROCEED?

> **NOTE:** You may jump ahead to the Create the first UIViewController Subclass section and skip this slight digression. Because this is the most complex code you will have encountered so far, we will use a road map. Most students find this useful—just as we all do when it comes to large-scale projects.

Let's sit back and draw out a road map. This project is going to allow a person to switch between two views, each of which contains a photograph of my grandpa at different stages in his life. Look at Figure 6–10 to see a diagram of an analogy that I hope you will find useful. The top-level character, einSwitchAppDelegate, tells SwitchViewController when to activate. The second-level character tells the Ein#Controllers to use their `.xib` tools to hold up their respective photos. You can also see how we use Xcode to program einSwitchAppDelegate and SwitchViewController and then how we use Interface Builder to work with the two Ein#Controllers and the `.xib` files.

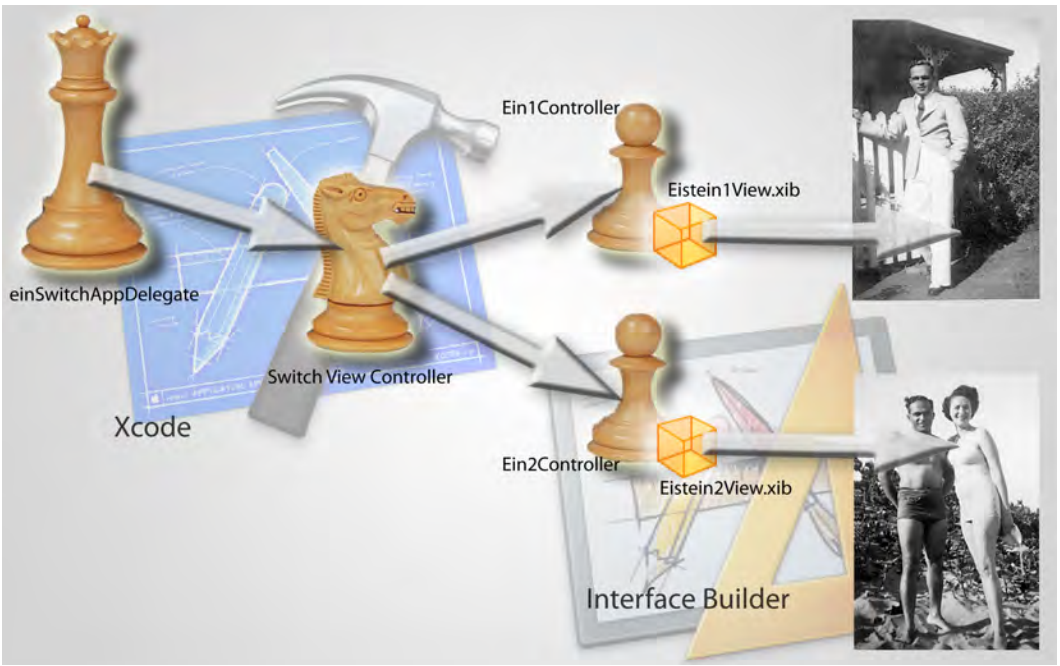


Figure 6–10. *Here is a road map of the einSwitch001 application.*

Here's a summary:

`Head Honcho Delegator`: Delegates work to the Switch View Controller

`Switch View Controller`: Tells which Controller to appear on the screen

`Einstein 1 Controller`: Carries the 1st photo

`Einstein 2 Controller`: Carries the 2nd photo

Unfortunately, we cannot use the aforementioned nomenclature. In Objective-C, as in many other computer languages, we need to use a special style of designation: *camelCase*. Referring to the "hump" in the middle, that's the name for the way geeky coders write compound words or phrases. The first letter is usually *not* capitalized, the elements are joined without spaces, and each element's initial letter *is* capitalized *within* the compound. As some of you now realize, we have already been using this nomenclature throughout the book. It's a very easy pattern, and you'll catch on immediately.

For example, if you were to write code that gets a map from a Google parser, you might call it `getMapFromGoogleParser`. This is a personal thing, and it's up to you … but try to be consistent. I will typically use lower case for the first word as long as it's not a proper noun (for example, the name of a city or a person) or a Class. Using these basic guidelines, let's rename the players in our code.

In essence, follow Apple's lead in their naming of entities. In general, when making up your own names, letTheFirstLetterNotBeCapitalized! Here are the names we'll use:

`einSwitchAppDelegate`: Delegates work to the Switch View Controller

`SwitchViewController`: Tells which Assistant Controller to appear on the screen

`Ein1Controller`: Carries the bottom layer photo

`Ein2Controller`: Carries the top layer photo

Now that we've named our characters, we need to define the tools the carriers use to hold up those pictures. They will pick up and present the photographs with nib (`.xib`) files, which we've come to love. This is what we have then:

- `einSwitchAppDelegate`
    - `SwitchViewController`
        - `Ein1Controller`
        - Nib file: `Einstein1ViewwxibbEin2Controller`
            - Nib file: `Einstein2View.xib`

The last step in identifying all the files we will need is to remember that each of those files needs both a *header* and an *implementation* file, as shown in Figure 6–11. Redrawing our road map we have:

- `einSwitchAppDelegate.h` and `einSwitchAppDelegate.m`
    - `SwitchViewControllerr.h` and `SwitchViewController.m`
        - `Ein1Controller.h` and `Ein1Controller.m`
            - `Einstein1View.xib`
        - `Ein2Controller.h` and `Ein2Controller.m`
            - `Einstein2View.xib`

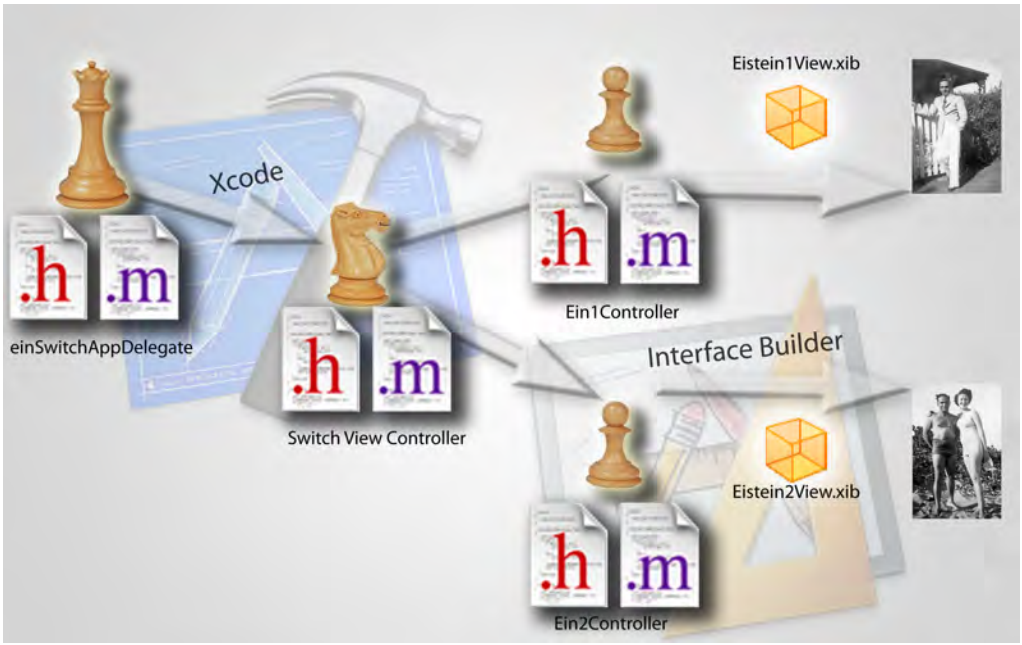


Figure 6–11. *Each of these needs its own header (.h) file and implementation (.m) file.*

Cool, huh?! Now we need to create them. Specifically, we need to create three Cocoa Touch sub-classes called `SwitchViewController`, `Ein1Controller`, and `Ein2Controller`, and we will also create each player's header and implementation files. Then, we need to make two nib files: `Einstein1View.xib` and `Einstein2View.xib`.

Create the 1st UIViewController Subclass

Once you have saved your window-based application as `einSwitch01` to your desktop, click the Classes folder in your Groups and Files sidebar. Enter ⌘N to open up the New File dialog window, as shown in Figure 6–12. Select the Cocoa Touch Class item in the *iPhone OS* sidebar, and then click the UIViewController subclass icon. Check the Also Create "SwitchViewController.h" box, and then click Finish to save the file as `SwitchViewController.m`. See Figure 6–13.

Figure 6–12. *Create the first of three Cocoa Touch UIViewController subclasses.*

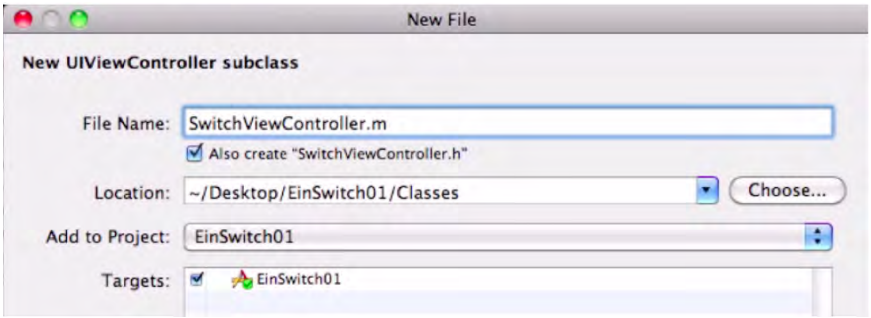


Figure 6–13. *Save the file as "SwitchViewController.m." Remember to check the box Also Create "SwitchViewController.h."*

In this chapter's other projects, we will select the With XIB for User Interface option, but right now we are going to build our nib files from scratch. Once you've done it here, you will feel good knowing what happens when you let Apple build the nib files. This way, you can always make a change and add your own bells and whistles when you want. On the other hand, if you always have this key file built for you, you won't ever get to know what's going on under the hood.

According to our road map …

- `einSwitchAppDelegate.h` and `einSwitchAppDelegate.m` **Done**
  - `SwitchViewController.h` and `SwitchViewController.m` *Doing Now*
    - `Ein1Controller.h` and `Ein1Controller.m`
      - `Einstein1View.xib`
    - `Ein2Controller.h` and `Ein2Controller.m`
      - `Einstein2View.xib`

Create the Ein1Controller

Here, we repeat what we have just done to create the second and third of the three Cocoa Touch UIViewController subclasses. We name them `Ein1Controller` and `Ein2Controller`. Remember—these two aforementioned characters are the subordinates of `SwitchViewController`, and they will present the photos to the user of your app. If you can do this on your own, go ahead now and try it. If not, just read along and follow me. You'll see that we're just repeating these steps again.

Once you have saved `SwitchViewController`, enter ⌘N to open the New File dialog window. Select the Cocoa Touch icon in the iPhone OS sidebar, click on the UIViewController subclass icon, press Return (or select the next button), and then save it as `Ein1Controller`. See Figure 6–14.

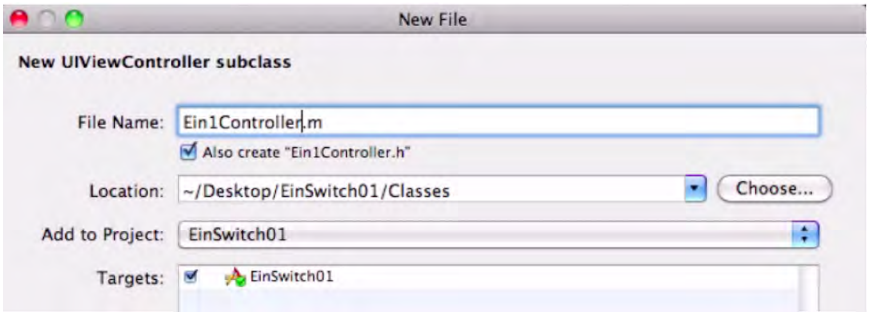


Figure 6–14. *Create the second of the three Cocoa Touch UIViewController subclasses.*

According to our road map …

- `einSwitchAppDelegate.h` and `einSwitchAppDelegate.m` **Done**
  - `SwitchViewController.h` and `SwitchViewController.m` **Done**
    - `Ein1Controller.h` and `Ein1Controller.m` *Doing Now*
      - `Einstein1View.xib`
    - `Ein2Controller.h` and `Ein2Controller.m`
      - `Einstein2View.xib`

Check Header and Implementation Files

Quickly check to see that both the header and implementation files have been created for `Ein1Controller`. This step may seem redundant, but often we forget to check all the appropriate boxes, such as the one to create the .h file. If I can help you get into the habit of double-checking this requirement here, you will save time in the long run. Once you proceed and write actions to one file *without* the other, it can be a real drag trying to recreate matching files down the road.

Also, while we are here, let's quickly review the relationship between an item's header and implementation files. Recall from Chapter 1 that all classes consist of two parts: a header (`.h`) file and an implementation (`.m`) file. Do you remember how I asked you to read aloud the following phrase? "*We inform the computer in our header files about the types of commands we will have it execute in the implementation files.*" This is still the case, but now I'd like you to add a little phrase in front: "Each class is a *product* of its header (`.h`) and implementation (`.m`) files. We inform the computer in our header files about the types of commands we will have it execute in the implementation files."

First, we inform; then, we execute!

Create the Ein2Controller

Once you have saved `Ein1Controller`, select ⌘N to open a new window in your Classes folder to open up the New File dialog window. Select the Cocoa Touch icon in the

iPhone OS sidebar, click on the UIViewController subclass icon, press Return (or select the next button), and then save it as `Ein2Controller`. See Figure 6–15.

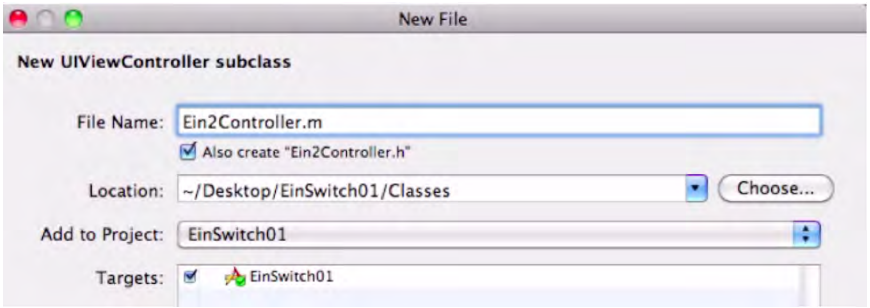


Figure 6–15. *Create the third of the three Cocoa Touch UIViewController subclasses.*

According to our road map …

- `einSwitchAppDelegate.h` and `einSwitchAppDelegate,m` **Done**
  - `SwitchViewController.h` and `SwitchViewController.m` **Done**
    - `Ein1Controller.h` and `Ein1Controller.m` **Done**
      - `Einstein1View.xib`
    - `Ein2Controller.h` and `Ein2Controller.m` *Doing Now*
      - `Einstein2View.xib`

Make Sure Images Are Embedded

We created the `einSwitchAppDelegate.h` and `einSwitchAppDelegate.m` files when we created the Window-based app at the very beginning of this example. We have just finished creating the Cocoa Touch UIViewController subclasses and named them `SwitchViewController`, `Ein1Controller`, and `Ein2Controller`. At this point, we have created all the characters in our play, but we still need to equip some of them with the tools they will use to display the bottom and top layer photographs – i.e. our switch view images.

For this, we will create two sets of tools, `.xib` files, that each of our helpers will use. In the New File dialog, go to your Resources folder, give it a click, enter ⌘N, and select the User Interface folder. Then, select the View XIB icon, as shown in Figure 6–16. Now we create the next nib file. Save the first of your two `.xib` files, `Einstein1View.xib` as shown in Figure 6–17, and immediately enter ⌘N to create your next nib file, `Einstein2View.xib`. Again, select the User Interface folder in the New File dialog, and then select the View XIB icon.

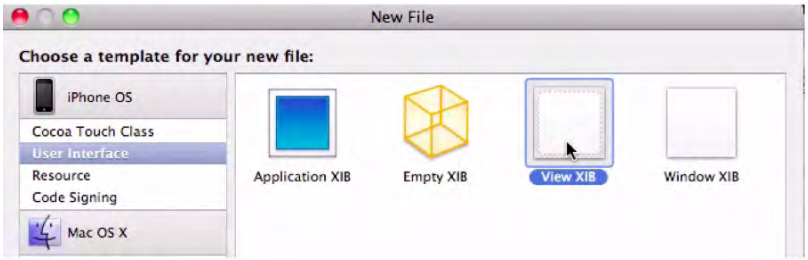


Figure 6–16. *Select the View XIB option to create the first of your two nib files.*

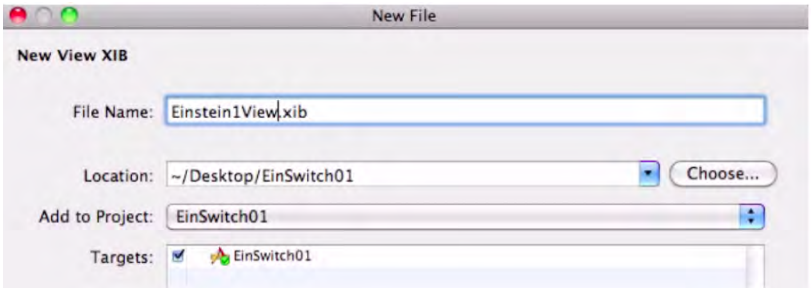


Figure 6–17. *Save the first of your two nib files, "Einstein1View.xib."*

Before moving on through the Objective-C forest, let's take a break and see where we are:

- `einSwitchAppDelegate.h` and `einSwitchAppDelegate.m` **Done**
  - `SwitchViewController.h` and `SwitchViewController.m` **Done**
    - `Ein1Controller.h` and `Ein1Controller.m` **Done**
      - `Einstein1View.xib` **Done**
    - `Ein2Controller.h` and `Ein2Controller.m` **Done**
      - `Einstein2View.xib` *Doing Next*

Save Einstein2View.xib

Save `Einstein2View.xib` as shown in Figure 6–18, and check that your Resources folder contains both the `.xib` files you have created. Now, as we glance at our road map, we see that we have come a long way:

- `einSwitchAppDelegate.h` and `einSwitchAppDelegate.m` **Done**
  - `SwitchViewController.h` and `SwitchViewController.m` **Done**
    - `Ein1Controller.h` and `Ein1Controller.m` **Done**
      - `Einstein1View.xib` **Done**
    - `Ein2Controller.h` and `Ein2Controller.m` **Done**
      - `Einstein2View.xib` **Done**

So, you're looking at this list and thinking that we've got all our characters now, and we can start to code. *Right?!* Hmm … not so fast. Do you see what is missing?

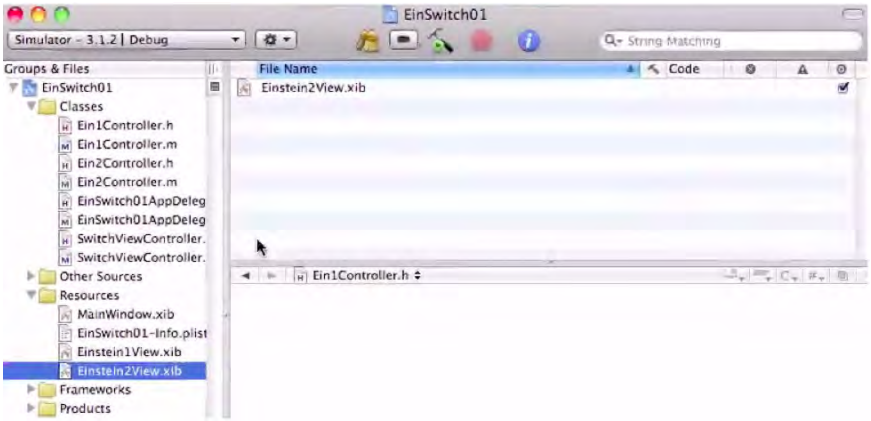


Figure 6–18. *Save the second of your two nib files, Einstein2View.xib.*

We do indeed have all our characters and their tools, but we lack some essential materials. We need to insert three items into our Resources folder: the top and bottom layer photos, as well as the icon image.

Drag the Images into Xcode

As you did in Chapter 5, Step 7, drag your first image into the Resources folder. Make sure, as you did before, to check the "Copy items into destination's group folder (if needed)" box, as well as the "Recursively create groups for any added folders" box. Then, click Add or press Return. Similarly, drag the second image into the Resources folder, and grab the icon image while you're at it.

As some of you may not know, you can multi-select by click-dragging and boxing in all the desired items, by control-clicking (non-adjacent items), or by shift-clicking (adjacent items) the first and last items desired (in a list). See Figures 6–19 and 6–20.

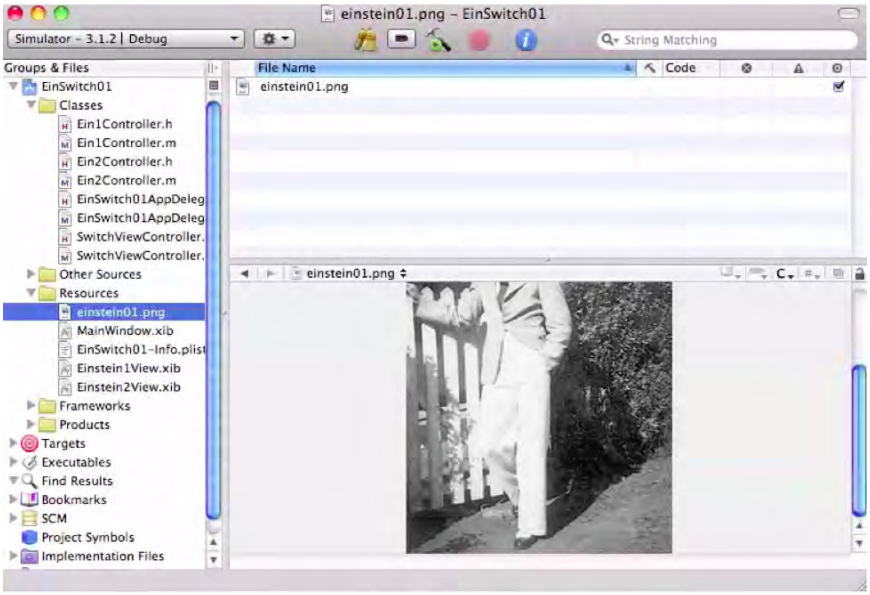


Figure 6–19. *Save the first of your two image files, einstein01.png.*

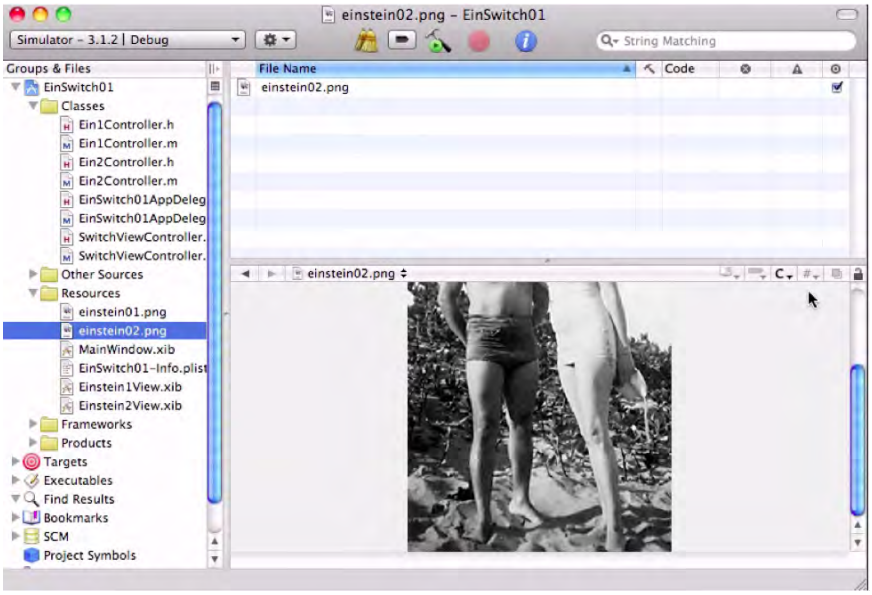


Figure 6–20. *Save the second of your two image files, einstein02.png.*

Assign your Icon in the “plist”

As you did before, you need to associate the icon image with this project. Open up the `Info.plist` file in the Resources folder and double-click the Icon file Value cell. In that

space, type in the name of your icon file, **icon.png**, as shown in Figure 6–21. Now, save your work by entering ⌘S.

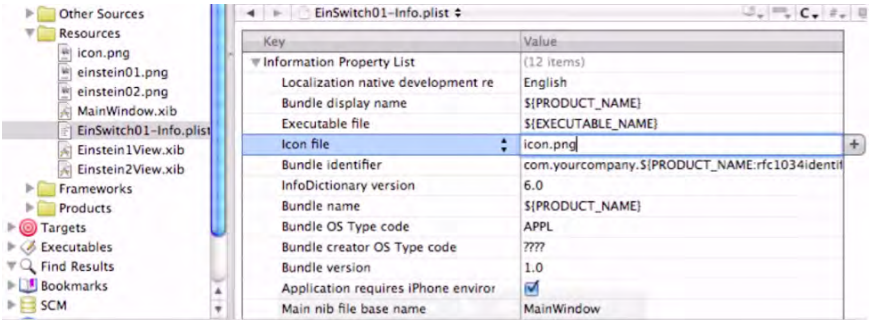


Figure 6–21. *Assign your icon file in the Information Property List: the "plist."*

Code the AppViewDelegate

OK! We've created all the players, we've associated all the files, and we've associated the icon file with the program. This means only one thing my fellow geeks! We're ready to code!

Let's go back to the Classes folder, as shown in Figure 6–22, and do what we always do: open the application's delegate header file. As we do this, let's review what we now know about how the file structures are created. When Xcode instantiates our project, it creates the role that's going to delegate and preside over the other files, `EinSwitch01AppDelegate` (see Figure 6–11). In the instantiation process, we saw how the system creates both a header (`.h`) and an implementation file (`.m`) for our `AppDelegate`, and it places the program name right before the phrase `AppDelegate`.

To summarize, because we named our program `EinSwitch01`, Xcode then automatically labeled our AppDelegate file `EinSwitch01AppDelegate`. Furthermore, within this file, the system generated the two more specific files: `EinSwitch01AppDelegate.h` and `EinSwitch01AppDelegate.m`.

Go to your Classes folder, then, and open up your `EinSwitch01AppDelegate.h` file. It should look as follows:

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

@interface EinSwitch01AppDelegate   NSObject <UIApplicationDelegate>{
UIWindow *window;
}

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

@end
```

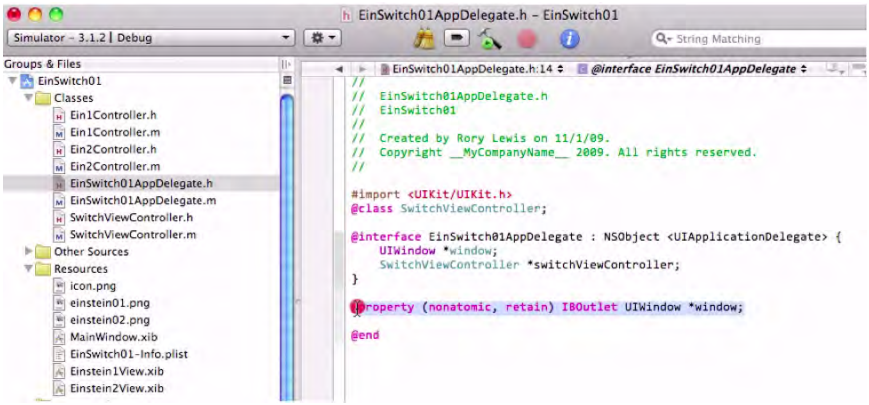


Figure 6–22. *Prepare to insert new code into the EinSwitch01AppDelegate.h file.*

At this point, we could easily do what we did in the past: enter the three lines of code we want to add … and move on in blindness. But this time, we're going to open the hood on our car and examine the engine. I am not going to overload you with too much information. I don't expect you to remember everything. My goal is to challenge you here, and to see if you can make sense of some of the details of the code.

Going back to Figure 6–10, you will notice that `EinSwitch01AppDelegate` communicates with and delegates work to `SwitchViewController`. The `SwitchViewController.h` file needs to have all the goodies and stuff that the `SwitchViewController.m` needs to have in order to tell the `SwitchViewController` which pawn will display its image. We provide this by adding the `SwitchViewController` class that we created (see Figures 6–12 and 6–13).

Recall that when we add a class, we put the `@` symbol in front of it to get the computer's attention. We will insert what is called an *@class precompiler directive* that announces our intention to call the `SwitchViewController` in the implementation file. These @class compiler directives tell the compiler that a class (of whatever type we're interested in creating) will be accessible for our use. Some programmers call this a *forward declaration* because we're giving the compiler a head's up *before* the class is declared in the implementation file. Therefore, we add `@class SwitchViewController` right after the `#import` code as follows:

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

@class SwitchViewController;

@interface EinSwitch01AppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
}

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

@end
```

We now know that when we code the implementation file that corresponds with this header file, to delegate work to the `SwitchViewController`, it will have the ability to do so. *Cool!*

Next, we move to the `IBOutlet` for the UIWindow portion of the code: the user interface window and its associated pointer window. We need to create an `IBOutlet` for all of the `SwitchViewController` bells and whistles. Therefore, after the `UIWindow *window` section, add `SwitchViewController *switchViewController` into the code. Once this is done, we can add an @ property directive and take care of the IBOutlet:

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

@class SwitchViewController;

@interface EinSwitch01AppDelegate : NSObject <UIApplicationDelegate> {
        UIWindow *window;
        SwitchViewController *switchViewController;
}

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

@end
```

Working SwitchView

To address the property directive line, we shall use a shortcut. Highlight and copy this entire line

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

and then paste it below itself. Then, highlight and copy this line:

`SwitchViewController *switchViewController`

See Figure 6–23.

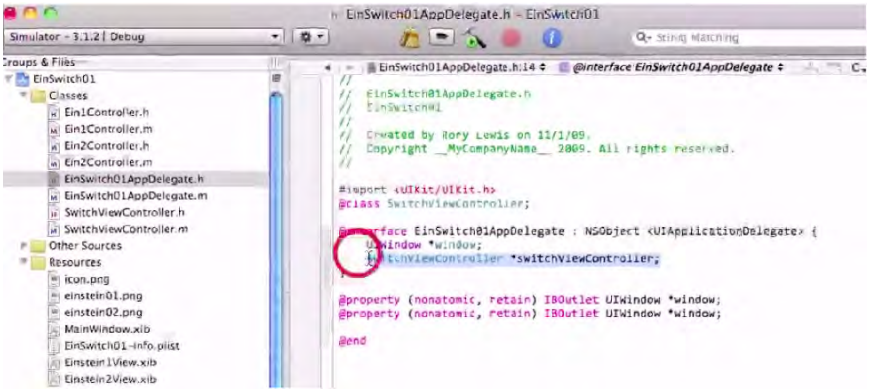


Figure 6–23. *Copy your SwitchViewController *switchViewController code.*

As shown in Figure 6–24, paste the

```
SwitchViewController *switchViewController;
```

code by entering (⌘V) *over* the highlighted

```
IBOutlet UIWindow *window;
```

portion of the code you just inserted.

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

@class SwitchViewController;

@interface EinSwitch01AppDelegate : NSObject <UIApplicationDelegate> {
        UIWindow *window;
        SwitchViewController *switchViewController;
}

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

@end
```

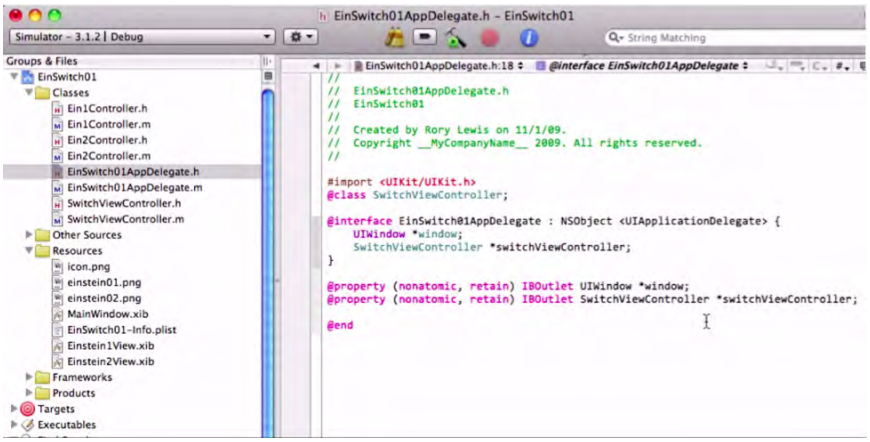


Figure 6–24. *Paste your SwitchViewController *switchViewController code.*

Remember that the `@property` directive declares that our object has a property with a specific type, while the `@synthesize` directive, which we are just about to address, puts into play the methods we declared in the `@property` directive. Save your work on the header file and move on to the implementation file, `eEinSwitch01AppDelegate.m`. See Figure 6–25.

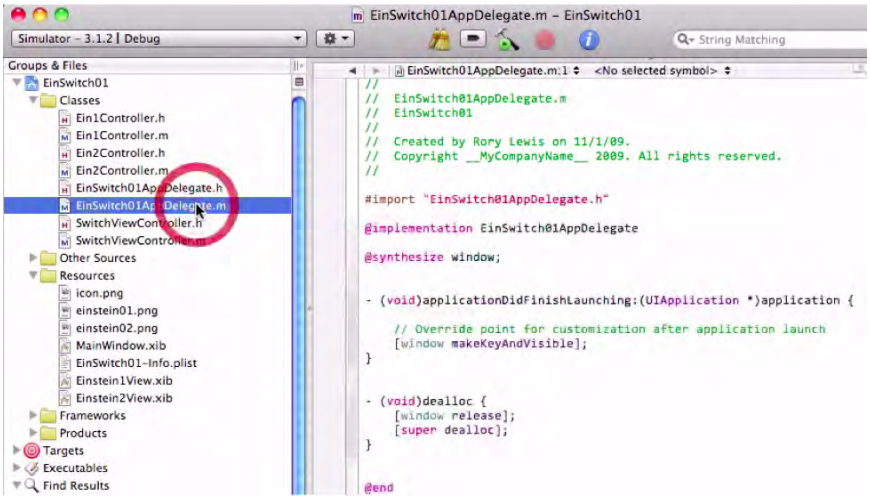


Figure 6–25. *Save your header file. It's time to move onto the implementation file!*

SwitchViewController and AppDelegate

In the Classes folder, scroll down to the corresponding implementation file, `EinSwitch01AppDelegate.m`, and open it. As shown in Figure 6–26, you will see that it looks as follows.

```
#import "EinSwitch01AppDelegate.h"

@implementation EinSwitch01sAppDelegate

@synthesize window;
- (void)applicationDidFinishLaunching:(UIApplication *)application {
// Override point for customization after application launch
    [windowmakeKeyAndVisible];
}
- (void)dealloc {
    [windowrelease];
    [superdealloc];
}
@end
```

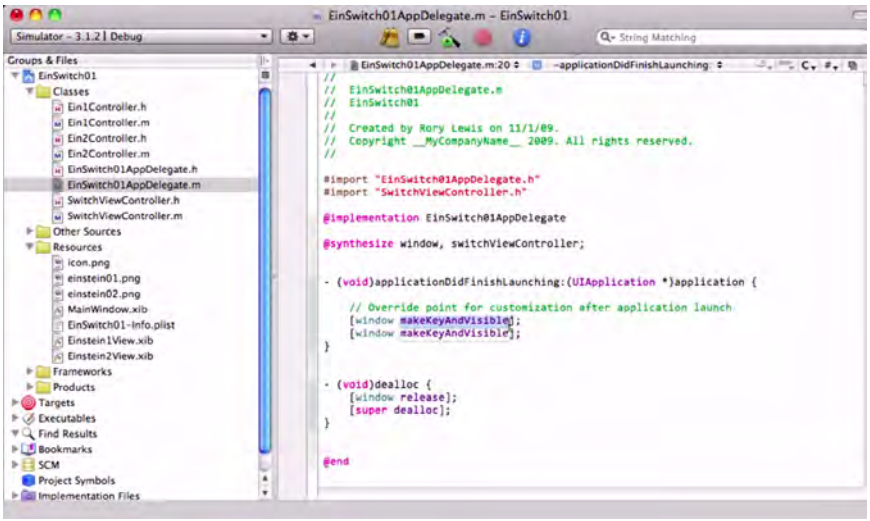


Figure 6–26. *Open the implementation file.*

As you can see, Apple has been kind enough to import the header file of the AppDelegate on which we have been working, `EinSwitch01AppDelegate.h`. Therefore, all the shout-outs we asked it to send this implementation file will be in here. Hmm … let's think about this for a second.

What other shout-outs or notices do we need here? If you go back to Figure 6–10, then you may recall that `EinSwitch01AppDelegate` orders the `SwitchViewController` to tell its subordinates to do specific actions—like hold up photographs of our subject. Thus, we need to also import all the shout-outs of the `SwitchViewController.h`, as seen here:

```
#import "EinSwitch01AppDelegate.h"
#import "SwitchViewController.h"

@implementation EinSwitch01AppDelegate

@synthesize window;

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

    // Override point for customization after application launch
    [window makeKeyAndVisible];
}
- (void)dealloc {
    [window release];
    [super dealloc];
}
@end
```

Note that the following line has code that Apple automatically instantiated for us:

```
@implementation EinSwitch01AppDelegate
```

This line has been something that we have skipped over in the past, but let's consider it now. One way that we can think of this, or pronounce this, is: "In this implementation

file, we hereby *realize* those commands and directives that were announced and defined in the `EinSwitch01AppDelegate` header file.

> **NOTE:** I often use the concept "realize" to deal with the term "instantiate." When the computer automatically instantiates a role or piece of the program structure, it is creating an object (i.e., making that character *real)*.

What else do we need to insert here? Recall that the `@property` directive, which is always located in the header file, declares that our object has a property with a specific type. In contrast, the `@synthesize` directive, located in the implementation file, notifies the compiler about these directives. Recall, too, that we had an `@property` directive

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

for the *window IBOutlet*. Then we added another `@property` directive for the SwitchViewController IBOutlet:

```
 @property (nonatomic, retain) IBOutlet SwitchViewController *switchViewController
```

The `@synthesize` directive for the window is done for us already by Apple, but we need to add the missing element, the `@synthesize` directive for the switchViewController. Let's do this as illustrated in the following code:

```
#import "EinSwitch01AppDelegate.h"
#import "SwitchViewController.h"

@implementation EinSwitch01AppDelegate

@synthesize window, switchViewController;

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

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

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

@end
```

Cool, huh? I hope you are beginning to see the logic behind adding these statements. Now, look at the next line:

```
(void)applicationDidFinishLaunching:(UIApplication *)application
```

and right-click on the `applicationDidFinishLaunching` portion. Scroll down and click on the `Find Selected Text Documentation` option. This will take you to a place where we have not yet ventured in this book, a wonderful little rabbit hole that leads to the Apple documentation of Xcode and Objective-C. Whatever portion of the code you do not

know in the months and years to come—no problem, just check it out in the Apple documentation.

By clicking on it, we see that it states – in part – the following:

```
applicationDidFinishLaunching:
Tells the delegate when the application has finished launching.
- (void)applicationDidFinishLaunching:(UIApplication *)application
Parameters
application
The delegating application object.
Discussion
This method is the ideal place for the delegate to perform various initialization↩
 and configuration tasks, especially restoring the application to the previous state↩
 and setting up the initial windows and views of the application…
```

What this is basically saying is: "Hey, I wrote this out for you without you having to really deal with it. If you really must know, it's where your delegates perform various initialization and configuration tasks."

Right below this, inside the brackets, Apple has indeed handled the initialization and configuration of an essential task: `[window makeKeyAndVisible]`. This bit of code causes the `UIWindow` to become visible and makes it the "first responder" of touches by the user when your app is run on the iPhone/iPad.

We need something else here, though. Can you think of it? We also need to make sure that the character that the Head Honcho Delegate is ordering around, the `SwitchViewController`, is visible to the user. We do this by adding a *subview*. The code that accomplishes this is `[window addSubview:switchViewController.view]`.

As the name suggests, this procedure adds a view to `switchViewController's` subviews. What's going on here is that, when `switchViewController` tells its subordinates to display the photographs of my grandpa, we want to add a view to a window as its *subview*. So, we're really asking `switchViewController` for the view it controls, which, when handled, displays the window to the user and enables it to accept *touches* and *other input*. Insert that line as shown here:

```
#import "EinSwitch01AppDelegate.h"
#import "SwitchViewController.h"

@implementation EinSwitch01AppDelegate

@synthesize window, switchViewController;

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

    // Override point for customization after application launch
    [window addSubview:switchViewController.view];
    [window makeKeyAndVisible];
}

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

```
@end
```

Our last step here is to *deallocate* the memory of all these actors in our play. Apple knows it has to deallocate the memory of various windows, and, for safety, it also performs a "super-deallocation." Just accept it for now … we're close to brain overload here.

We need to deallocate the memory we are using for … what? Hmm. Have a guess. The one item to which we have been continually referring in this section? The `switchViewController`?

Yes. Let's do it!

```
#import "EinSwitch01AppDelegate.h"
#import "SwitchViewController.h"

@implementation EinSwitch01AppDelegate

@synthesize window, switchViewController;

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

    // Override point for customization after application launch
    [window addSubview:switchViewController.view];
    [window makeKeyAndVisible];
}

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

@end
```

Now enter ⌘S to save your work. *Holy implementation file, Batman!* You're done!

You have taken care of the `EinSwitch01AppDelegate` in terms of both its header and its implementation files. Referring back to Figure 6–10, we see that we have created the code for the `einSwitchAppDelegate`—represented by the chess queen!

Whew - time for a break!

SwitchViewController Header File

Continuing with our chess metaphor, you can tell yourself that you are finished working with the AppDelegate file—the queen. Now it's time to focus on her immediate subordinate, the `SwitchViewController`. As you know, this figure, represented by the knight, has the role of commanding either of its underlings to hold up its image. In the steps ahead, we will deal with the header (`.h`) file and the implementation (`.m`) file of the `SwitchViewController`.

Now we need code that tells the `Ein1Controller` to displays its photograph of our subject; then we can tell the `Ein2Controller` to display *its* photograph. So, scroll down in your Classes folder and open up the header file: `SwitchViewController.h`. We first need to make sure that the precompiler will even know who the `Ein#Controllers` are. See Figure 6–27. Remember that the `@` symbol gets the computer's attention for the establishment of a specific relationship, and we use the `@class` precompiler *directive* to do this.

```
#import <UIKit/UIKit.h>
@class Ein1Controller;
@class Ein2Controller;

@interface SwitchViewController : UIViewController {

}

@end
```

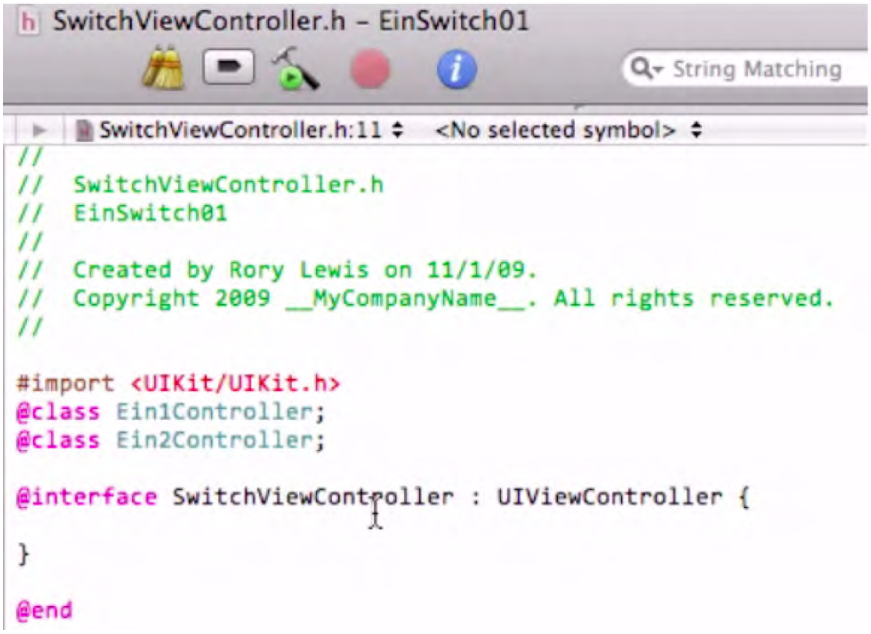


Figure 6–27. *Copy the @class precompiler directive for Ein1Controller in order to create one for Ein2Controller.*

Next, we need to make sure that the implementation file will know who these `Ein#Controllers` are … that they even exist. In technical terms, we say that we need to declare the *instance variables* that we need to use throughout the class. This is done inside the brackets, immediately following the directive `@interface SwitchViewController: UIViewController`.

We do this by using a *pointer*—yes, the asterisk, which, thus far, I've been telling you to ignore. Well, it's time to consider it. We need to tell the implementation file to reserve a place in memory for `Ein1Controller` and `Ein2Controller`, and we do this by using a pointer … for *each* of the subordinate roles:

```
#import <UIKit/UIKit.h>
@class Ein1Controller;
@class Ein2Controller;

@interface SwitchViewController : UIViewController {
        Ein1Controller *ein1Controller;
        Ein2Controller *ein2Controller;
        }
@end
```

Next, we need to use the `@property` directive to define these variables as properties, and we do this with the same code we've used many times before:

```
@property (retain, nonatomic) Ein#Controller *ein#Controller
```

making sure to do it *individually* for each `Ein#Controller` that presents the user with a photograph of my grandfather. We do this as follows:

```
#import <UIKit/UIKit.h>
@class Ein1Controller;
@class Ein2Controller;

@interface SwitchViewController : UIViewController {
        Ein1Controller *ein1Controller;
        Ein2Controller *ein2Controller;
}

@property (retain, nonatomic) Ein1Controller *ein1Controller;
@property (retain, nonatomic) Ein2Controller *ein2Controller;

@end
```

Our next item to address is that we need an action of some type to switch views. We've called this an *instance* before, and we will still do so. Technically, we say: "We need an instance method (and use the "minus" sign) to advertise to the implementation file that we will be incorporating an `IBAction`." In other words, it will "shout out" to the implementation file that a method in your code needs to be triggered, or called into action, and that these commands will be implemented in Interface Builder.

We need to give this new action that's going to switch views a name, so let's call it … hmm … `switchViews`! *Yeah!* Thus, we will enter this code:

```
-(IBAction)switchViews:
```

This segment of our code, in turn, needs to point to a specific construct or role, and we use (`id`) for this purpose. Finally, we will need to add the "sender" component that will trigger the event.

Thus, our latest code insertion is

```
(IBAction)switchViews:(id)sender
```

By the way, remember that we generally follow up all these lines of code with a semicolon, to alert the computer that we are finished with that line.

```
#import <UIKit/UIKit.h>
@class Ein1Controller;
```

```
@class Ein2Controller;

@interface SwitchViewController : UIViewController {
        Ein2Controller *ein2Controller;
        Ein1Controller *ein1Controller;
}
@property (retain, nonatomic) Ein2Controller *ein2Controller;
@property (retain, nonatomic) Ein1Controller *ein1Controller;

-(IBAction)switchViews:(id)sender;

@end
```

As shown in Figure 6–28, it's time to enter ⌘S to save your work. You're done with that file … *Scooby-Dooby-Doo!* Now, we move on to the implementation file. Well done!

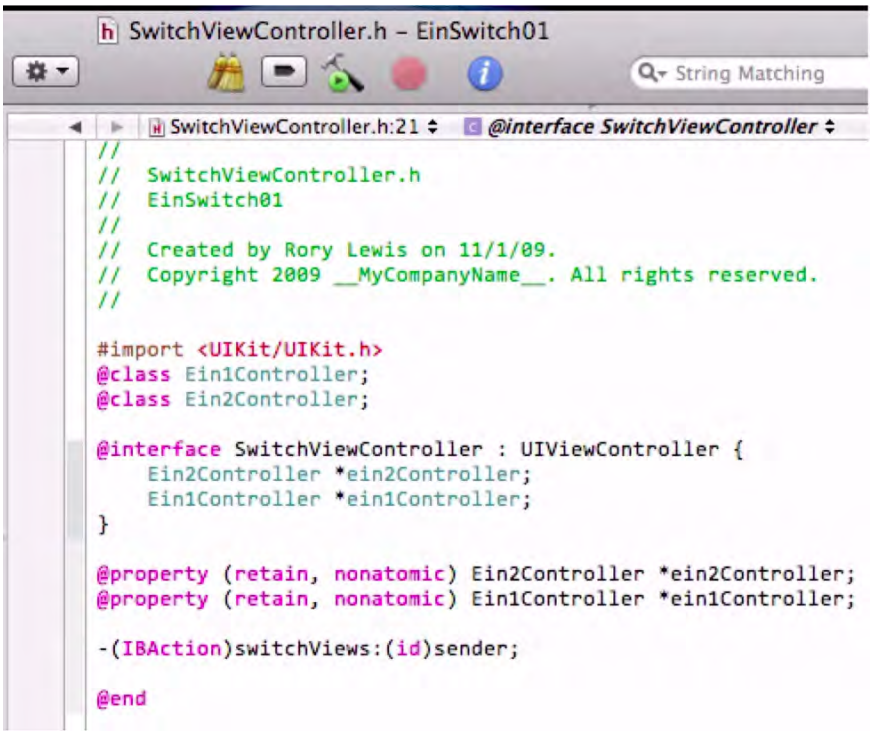


Figure 6–28. *Once SwitchViewController.h is complete, save it, and go to the Lazy Load!*

Ready for Lazy Load—Implementation File

In the Classes folder, go to the `SwitchViewController`'s implementation file, `SwitchViewController.m`, and click it open. When you open it, you will see all the basic code we've seen before, which Apple automatically and conveniently instantiates for us. As shown in Figure 6–29, the details of this code are invisible to the compiler because each set of classes is placed inside a *comment*. I think now is a good time to address the nature and function of comments—in the context of coding apps.

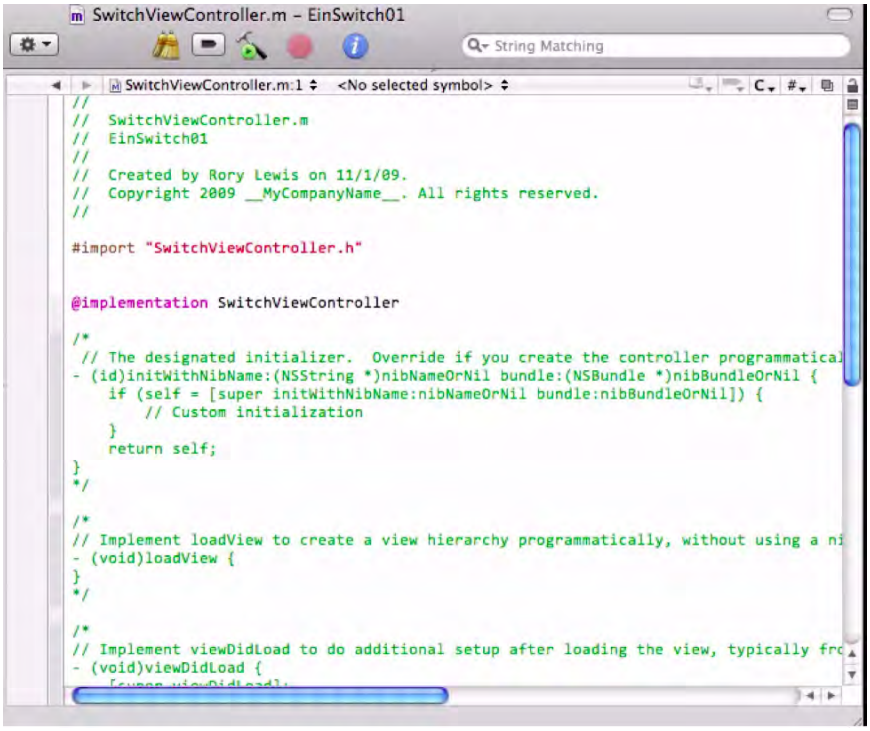


Figure 6–29. *SwitchViewController's implementation file is loaded with code that's associated with comments.*

> **NOTE:** If you are knowledgeable about comments and lazy loads, skip this section and go to Step 16. If you're not sure, stay with us here and read on.

A Note about Comments and Lazy Loads

We know that Xcode uses, and is based on, the programming language Objective-C, and that applications are run by virtue of code getting compiled into ones and zeroes that microprocessors understand. In Objective-C, as in other languages—particularly the C language it's based on, our preprocessor supports two styles of comments. These comments, in essence, make things *invisible* to the innards of the machine.

We have already examined and discussed the double forward slash signal: //, after which comments can be inserted—and which prohibits the compiler from seeing those comments. These are called *BCPL-style* comments. There is also the slash-asterisk: /* and the asterisk-slash: */, between which comments can be placed. These are known as *C-style* comments. For example, we might see something like this:

```
/* This is material that will be "invisible" to the compiler. */
```

Apple knows that, most of the time, we will use at least one of the classes, so it inserts comments for our convenience as follows:

```
#import "SwitchViewController.h"

@implementation SwitchViewController

/*
 // The designated initializer.  Override if you create the controller↩
programmatically and want to perform customization that is not appropriate↩
 for viewDidLoad.
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil {
    if (self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil]) {
        // Custom initialization
    }
    return self;
}
*/

/*
// 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 for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
*/

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

Going further into this brief tangent on comments would not serve us at this time. I wanted you to take a quick peek, though, so that you would have some appreciation for where and how some of the details are hidden away. I want you to realize that there are some very successful iPhone/iPad app developers out there who *do not* understand all the gobbledy-gook in the example above. However, a good developer *does* know how to utilize these comment styles when the need arises—and even how to beg, borrow, and steal other developers' work along these lines.

At the beginning of this chapter, I mentioned that we would be dumping—into the implementation file—one of the most widely used, generic, boilerplate chunks of code that loads views onto the iPhone or iPad. This method is called a *Lazy Load*. This process has this name for a number of reasons. First, we are somewhat lazy for using it! *True enough.*

Second, it's lazy in terms of *memory management*. This is OK for smaller programs, but as we move to creating larger applications, we will need to get more sophisticated. We will probably use Shark and other tools to fix "memory leaks," which are often caused by lazy loads. Don't worry, though! You can still create excellent (and lucrative) apps that rely on a lazy load.

Copy Contents of SwitchViewController.txt

Earlier in this chapter you downloaded the `006_Chapter_6hEinSwitch01.zip` file. Inside this zip file, you found and copied the `SwitchViewController.txt` file to your desktop. Now it's time to open this file and to select all the contents by entering ⌘A. Once the entire file is selected and highlighted, make a copy by entering ⌘C. See Figure 6–30.

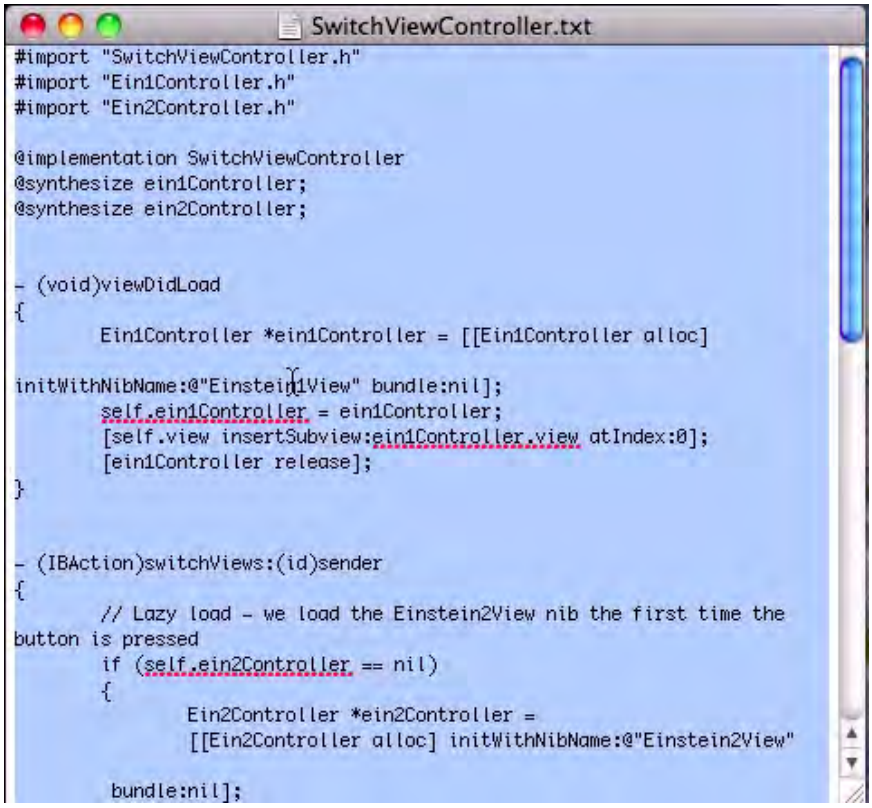


Figure 6–30. *Click the SwitchViewController.txt file on your desktop, click inside the window, select all by entering ⌘A, and then copy the highlighted text by entering ⌘C.*

With your Lazy Load copied onto your clip board, open the implementation file, `SwitchViewController.m`, enter ⌘A to select all of its contents, and then, without doing anything else, immediately paste the copied text by entering ⌘V, as shown in Figure 6–31. Then, enter ⌘S to save your "work."

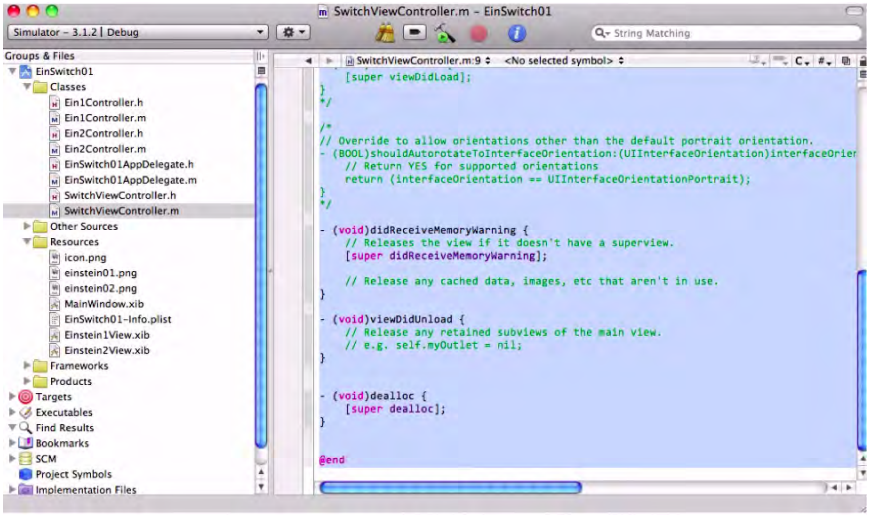


Figure 6–31. *Open the SwitchViewController.m file and paste your lazy load into it by entering ⌘A and, then immediately, ⌘V.*

Guess what? You're done with the code for that particular character of our play! You can now move directly to the next step, if you like, and start connecting everything in your nib files. That would be perfectly acceptable.

> **NOTE:** Following is a brief review of the boilerplate Lazy Load. If this doesn't interest you at this time, it will be fine to go directly to the Select the File's Owner section.

A Note about Apple's Boilerplate Implementation File

You don't have to memorize it, or understand every symbol, but I do want you to come away with at least a vague understanding of how this all works. The first portion of this file is code I programmed for you, which I included in the Lazy Load. I simply imported and synthesized the two `Ein#Controller.h` header files as follows:

```
#import "SwitchViewController.h"
#import "Ein1Controller.h"
#import "Ein2Controller.h"

@implementation SwitchViewController
@synthesize ein1Controller;
@synthesize ein2Controller;

// We've done the above about 7 times already, so I figured you'd be cool↩
```

```
 with me inserting the import
// and synthesis files for you.   The next piece of code in here is the↩
 (void)viewDidLoad.

- (void)viewDidLoad
{
        Ein1Controller *ein1Controller = [[Ein1Controller alloc]

        initWithNibName:@"Einstein1View" bundle:nil];
        self.ein1Controller = ein1Controller;
        [self.view insertSubview:ein1Controller.view atIndex:0];
        [ein1Controller release];
}
```

The - (void)viewDidLoad method, given to us by Apple, is called after the SwitchViewController view is loaded into memory. You may recall that this method, or process, is called (or activated) regardless of whether the views were stored in .xib files or created programmatically via the loadView method. Right now, give all this code a simple and friendly nod of acknowledgement. You will find yourself cutting and pasting this stuff as you do further work, and I want to help you make sure that your .xib files properly load. Moving on, the next method we view is the essence of the Lazy Load:

```
- (IBAction)switchViews:(id)sender
{
        // Lazy load - we load the Einstein2View nib the first time the button↩
 is pressed
        if (self.ein2Controller == nil)
        {
                Ein2Controller *ein2Controller =
                [[Ein2Controller alloc] initWithNibName:@"Einstein2View"
bundle:nil];
                self.ein2Controller = ein2Controller;
                [ein2Controller release];
        }

        if (self.ein1Controller.view.superview == nil)
                //This is with no animation
        {
                [ein2Controller.view removeFromSuperview];
                [self.view insertSubview:ein1Controller.view atIndex:0];
        }
        else
        {
                [ein1Controller.view removeFromSuperview];
                [self.view insertSubview:ein2Controller.view atIndex:0];
        }
}
```

Can you gather what is going on with this portion of the code? You can see that we first command the computer to load our second photograph and then, depending on whether the other one is loaded or not, we swap them around—according to when the user presses the button to switch views. Apple provides us with the next set of methods. These initialize the nib files, enable the screen to use certain tools (such as "auto-rotate," if the user rotates the iPhone), receive memory warnings (if the user starts to run out of memory), and deallocate memory once the user is done with the app.

```
// Initialization code
- (id)initWithNibName:(NSString *)nibNameOrNil
                        bundle:(NSBundle *)nibBundleOrNil {
        if (self = [super initWithNibName:nibNameOrNil
                                                    bundle:nibBundleOrNil]) {
        }
        return self;
}
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↩
interfaceOrientation {
        // Return YES for supported orientations
        return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
- (void)didReceiveMemoryWarning {
        [super didReceiveMemoryWarning]; // Releases the view if it doesn't↩
 have a superview
        // Release anything that's not essential, such as cached data
        if (self.ein1Controller.view.superview == nil)
                self.ein1Controller = nil;
        else
                self.ein1Controller = nil;
}
- (void)dealloc {
        [ein2Controller release];
        [ein1Controller release];
        [super dealloc];
}
@end
```

This brief overview is sufficient given your level of understanding. Meanwhile it's worth mentioning that, at a recent conference, I heard a husband and wife team, who are making about $50K/month, admit to hundreds of iPhone developers that they barely understand Apple's boilerplate implementation code. The key, as you may be gathering, is that they are clever enough to know where to paste the names of their views! For instance, do you see where I inserted `Einstein1View` and so forth in the `(void)viewDidLoad` method? *Very good!*

Let's go to Interface Builder and connect everything up. We're almost done!

Working on the .xib Files

As indicated by Figure 6–32, go to the Resources folder and open the `MainWindow.xib` file. As always, this will open Interface Builder.

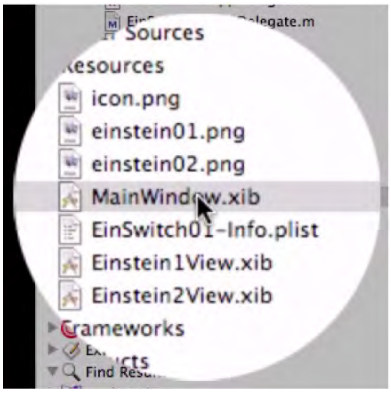


Figure 6–32. *Go to the Resources folder and open the MainWindow.xib file.*

Once Interface Builder is open, go to your Library window by entering ⌘L. Navigate to the Controllers folder in *Cocoa Touch* and drag a *View Controller* into your Document window, as shown in Figure 6–33.

> **NOTE:** The Document window is the window that holds your documents. It is not the open window of your desktop.

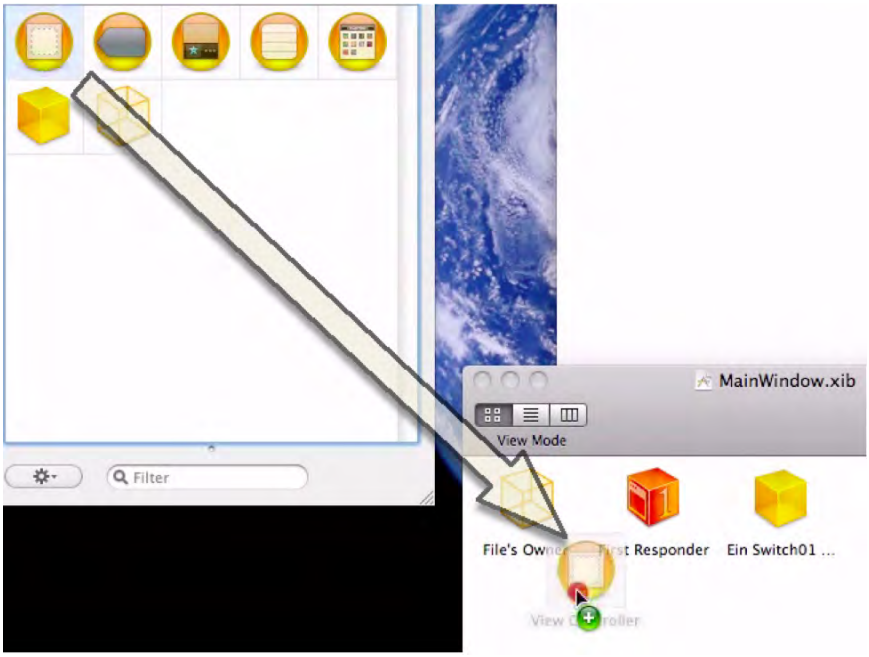


Figure 6–33. *Drag a View Controller (top left icon) into your Document window.*

Select the File's Owner

With the View Controller still activated (if it isn't, click on it once), have a look in the Inspector window and note that, by default, it is associated with a generic `UIViewController`. We really don't want that, though. We want *our* View Controller interacting with the code that we programmed for our *own* creation—a character that we have named `SwitchViewController`. So, click on the drop-down menu and select `SwitchViewController`, which will be associated with this View Controller in the Document window. Refer to Figure 6–34.

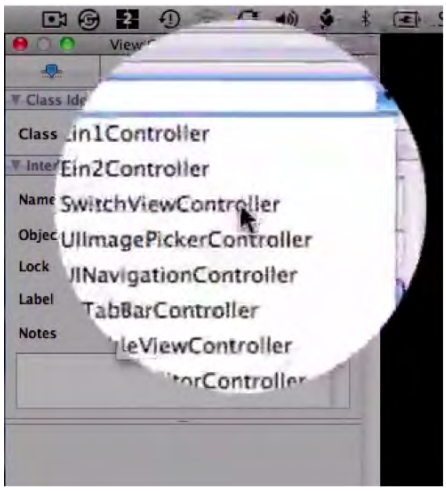


Figure 6–34. *Select SwitchViewController to associate it with the View Controller in the Document window.*

Drag a View onto the Screen

Scroll down to the Windows, Views & Bars folder in your Library and drag a view onto your screen as shown in Figure 6–35. We are doing this because we need views for the SwitchViews that we are generating. We could make our own buttons to switch between the different views, but we've already done this. You know how to do that. So, I thought it would be nice to use one of the pre-coded buttons that is in the Windows, Views & Bars folder. Go ahead then and drag a Toolbar onto your screen, and place it at the bottom of the workspace. Refer to Figure 6–36. We now want to give the button on the toolbar a name. Click on it, and call it *Switch Views*, as demonstrated in Figure 6–37.

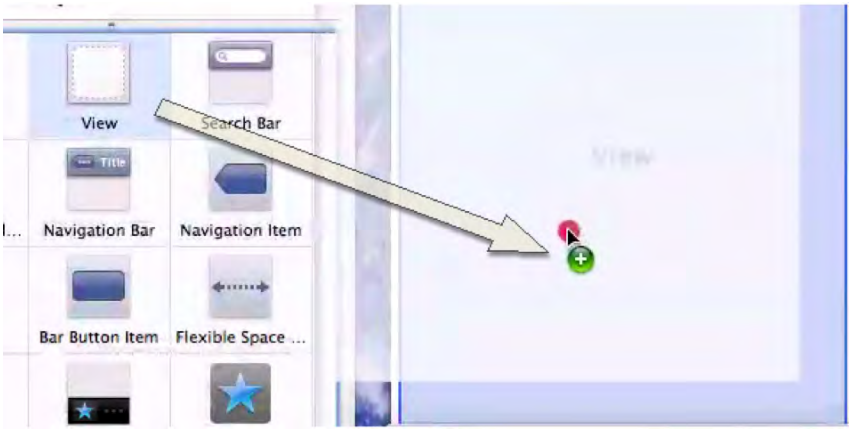


Figure 6–35. *Drag a view onto your screen.*

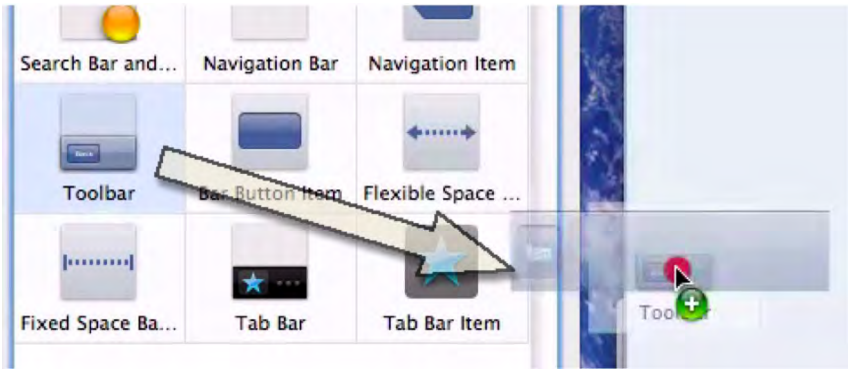


Figure 6–36. *Drag a toolbar onto your screen and place it at the bottom of the workspace.*

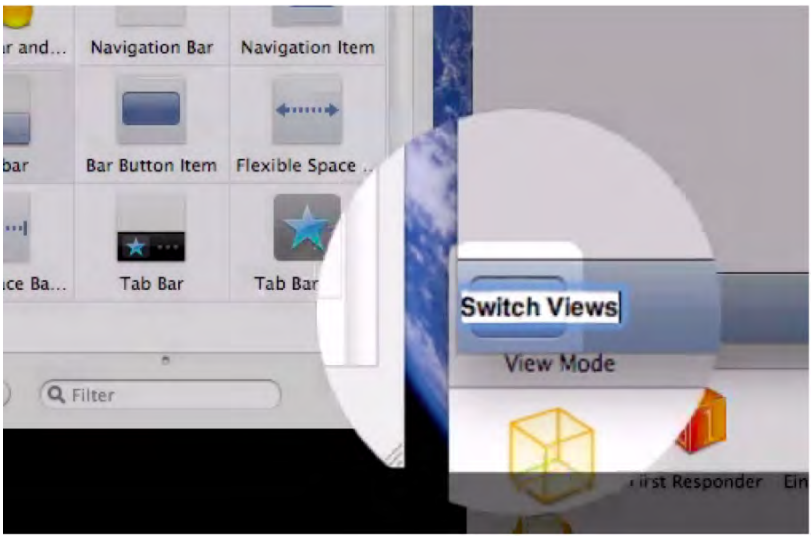


Figure 6–37. *Name the button "Switch Views."*

We need to connect the mechanism inside the Switch Views button to the code we have now associated with the `SwitchViewController`. Control-drag from the Switch Views button downward to the Switch View Controller icon. At this point, your screen should look similar to Figure 6–38.

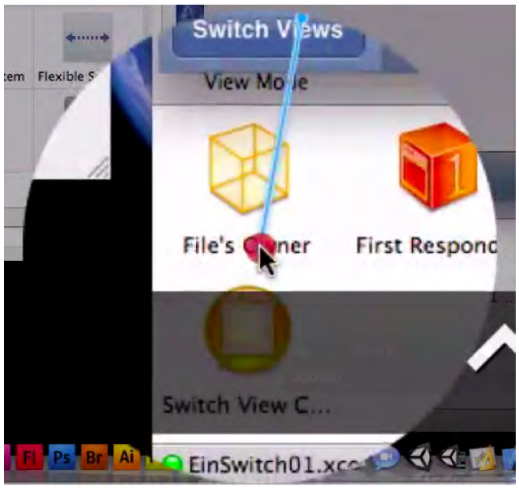


Figure 6–38. *Control-drag from the Switch Views button downwards to the Switch View Controller icon.*

Now we want to connect the Switch Views button to the appropriate option. As you drag over the Switch View Controller icon, the Sent Actions black drop-down menu opens, and you can select the switchViews option, as shown in Figure 6–39.

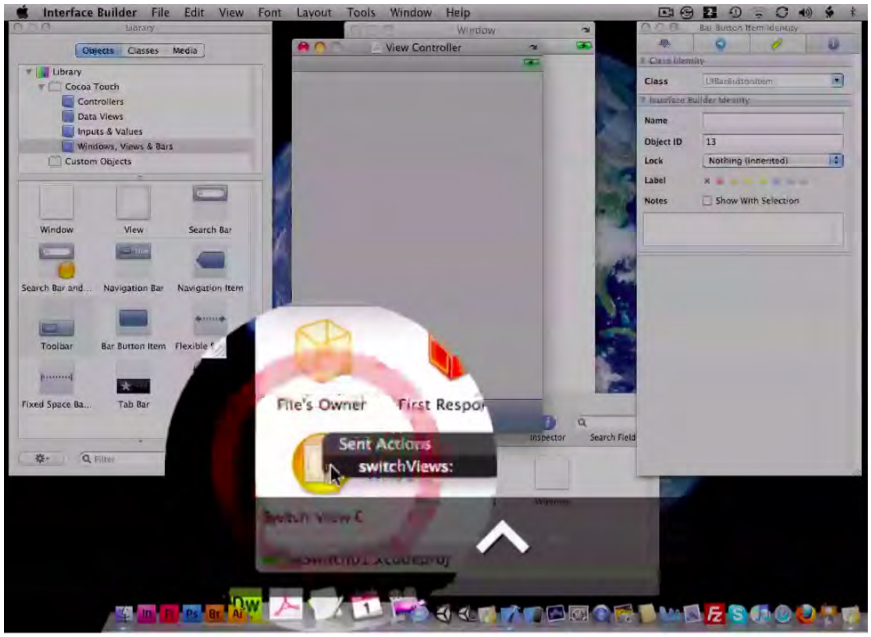


Figure 6–39. *Connect the Switch Views button to the switchViews option.*

What is going on here? Remember, back in Step 14, when we created an *action* that would advertise that we have a switch view? It looked like this:

```
-(IBAction)switchViews:(id)sender
```

The code we're seeing here, in the drop-down menu, is a result of that code. I have bolded it below to jog your memory.

```
#import <UIKit/UIKit.h>
@class Ein1Controller;
@class Ein2Controller;

@interface SwitchViewController : UIViewController {
        Ein2Controller *ein2Controller;
        Ein1Controller *ein1Controller;
}

@property (retain, nonatomic) Ein2Controller *ein2Controller;
@property (retain, nonatomic) Ein1Controller *ein1Controller;

-(IBAction)switchViews:(id)sender;
```

Now we have connected the `SwitchViewController` to the *View* Controller, and that is why we see the file in the drop-down menu. We connect the code that we have previously entered to the button so that, when the user presses the Switch Views button on the toolbar, it invokes and runs this code.

Now, we want to connect the `Ein1Controller` to the `SwitchViewController`. So … start to control-drag from the `Ein1Controller` to the *View Controller* that houses your `SwitchViewController`.

> **NOTE:** We start with one of the images present on the screen, and, as we decided in the Lazy Load `if` statements, we make `Ein1Controller` the active subordinate in charge of this initial display. See Figure 6–40.

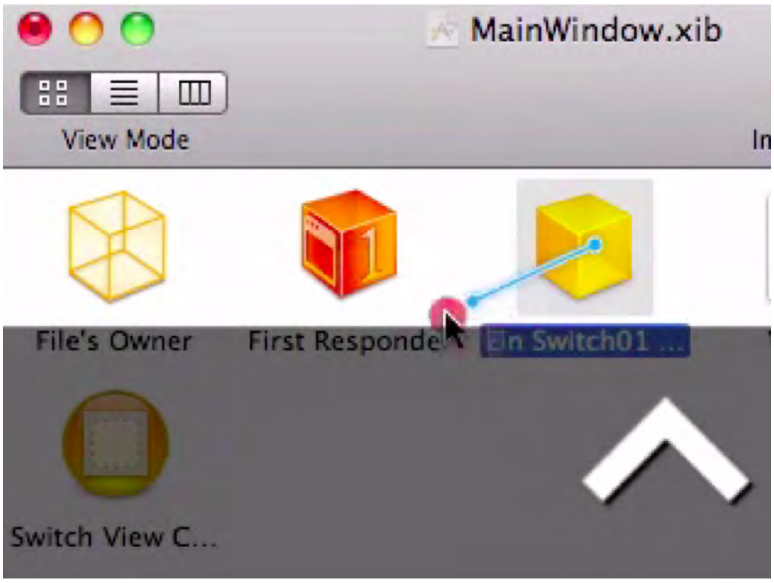


Figure 6–40. *Control-drag from the Ein1Controller to the Switch View Controller icon.*

Again, as you drag your cursor over the *View Controller*, you will see a black drop-down menu appear with `SwitchViewController` in it. Connect the fishing line to it, and then let go, as shown in Figure 6–41. Now, enter ⌘S to save your work.

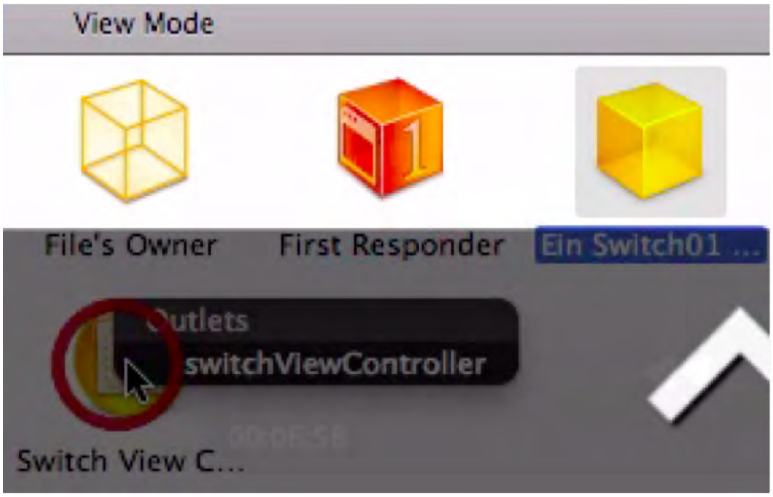


Figure 6–41. *In the drop-down menu, select the SwitchViewController option.*

Start Working on the Einstein#View.xib Files

We're almost done!

In reference to our network of characters depicted in Figure 6–10, we have everything working except for the two nib files, `Einstein1View.xib` and `Einstein2View.xib`. We have referred to these nib files as the tools that the Ein#Controllers use to hold up and display their respective photographs of my grandpa.

Let's open up `Einstein1View.xib` as shown in Figure 6–42. Both of these nibs are doing one thing, and one thing only, and that is holding up an image. Hence, it makes perfect sense that the first thing we do here is to drag in an Image View from the Data Views folder in our Library, as shown in Figure 6–43. We need to associate a specific image file with the Image View that we've just dragged onto the screen. As depicted in Figure 6–44, go to the property inspector and, after clicking on the pull-down menu, select `einstein01.png`, the image file to be associated with the Image View of the first of the two Ein#Controllers.

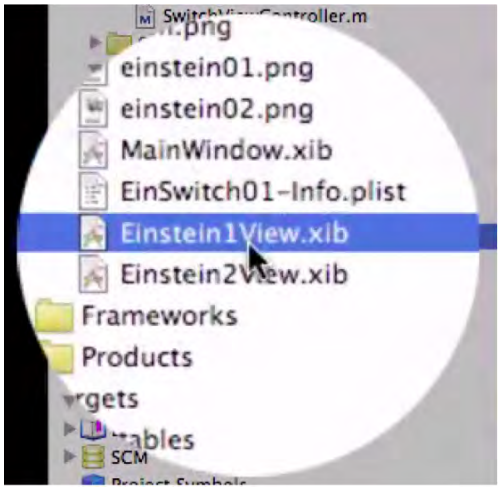


Figure 6–42. *Begin work on the nib files by selecting the Einstein1View.xib file.*

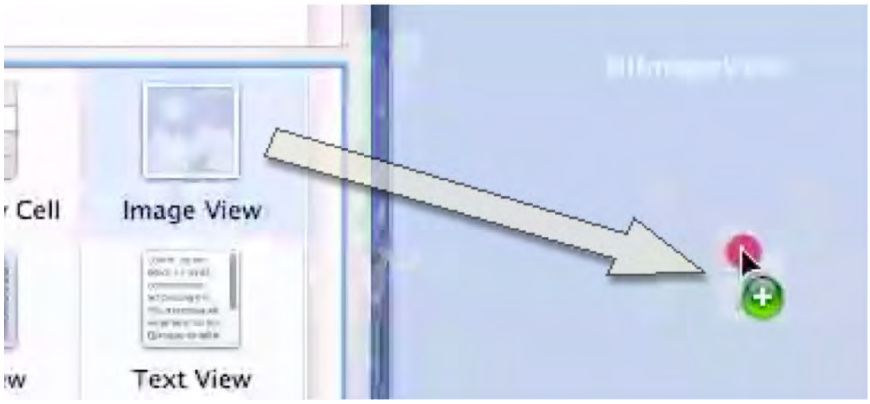


Figure 6–43. *Of course, images need Image Views.*

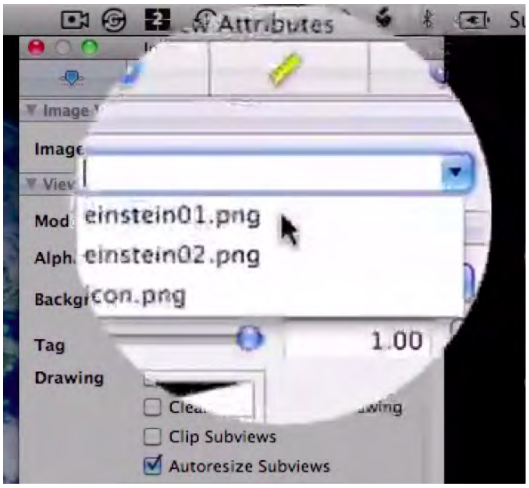


Figure 6–44. *Associate the first image, einstein01.png, with the Image View.*

Now, we need to address the toolbar button and make sure we assign the correct action to it. First, click the File's Owner icon as shown in Figure 6–45. Watch out here. For some reason, students forget that, before changing the label of the File's Owner icon, they need to first click on it.

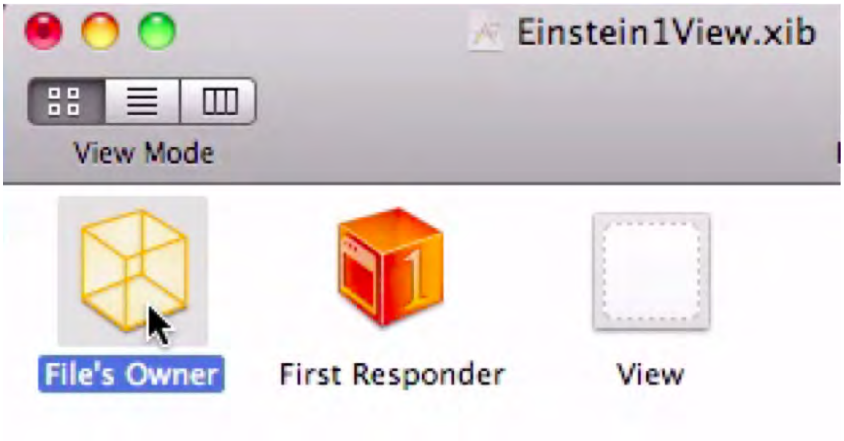


Figure 6–45. *Click the File's Owner icon.*

Once the icon is selected, go up to the Inspector Box. Once you get to the label, choose the Label option on the black drop-down menu, as shown in Figure 6–46.

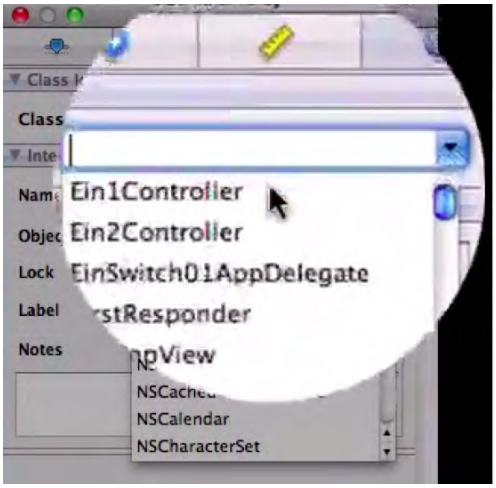


Figure 6–46. *Connect the Ein1Controller to the File's Owner.*

Do you recall how we entered code that directed the `Ein#Controllers` to use nib files to display their images? Well, we need to make clear that the guy who controls everything here, the File's Owner, is indeed `Ein1Controller`. Click the File's Owner icon and then associate it, via the drop-down menu in the property box, with `Ein1Controller`, as shown in Figure 6–46.

Now, we need to control-drag from the File's Owner icon to the View icon. When you control-drag over the View icon, a drop-down menu appears with View as an option. Point to this option and then release. See Figures 6–47 and 6–48. With this done, we have completed working on the `Einstein1View.xib` file. Enter ⌘S to save your work, and then enter ⌘Q to quit Interface Builder in order to return to Xcode.

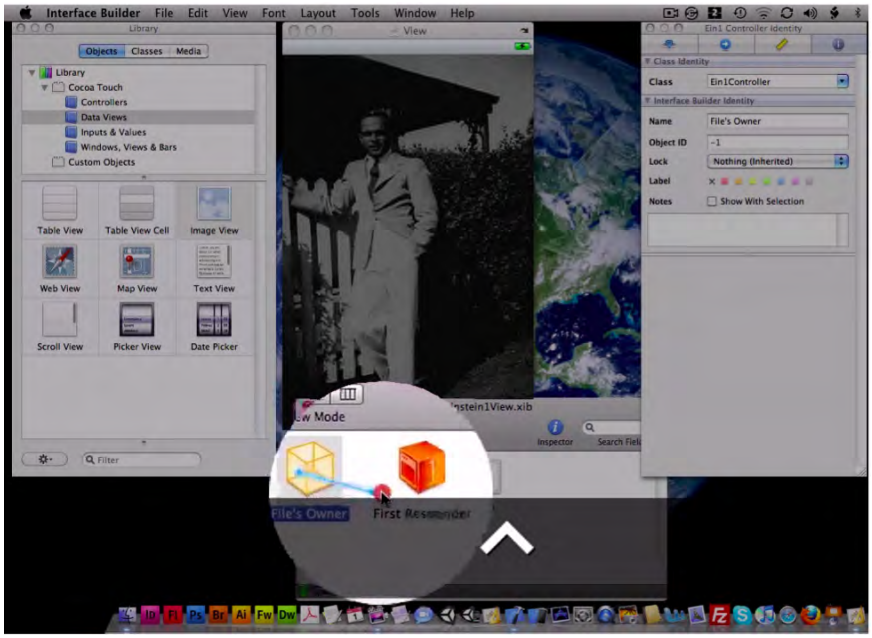


Figure 6–47. *Connect the File's Owner icon with the View.*

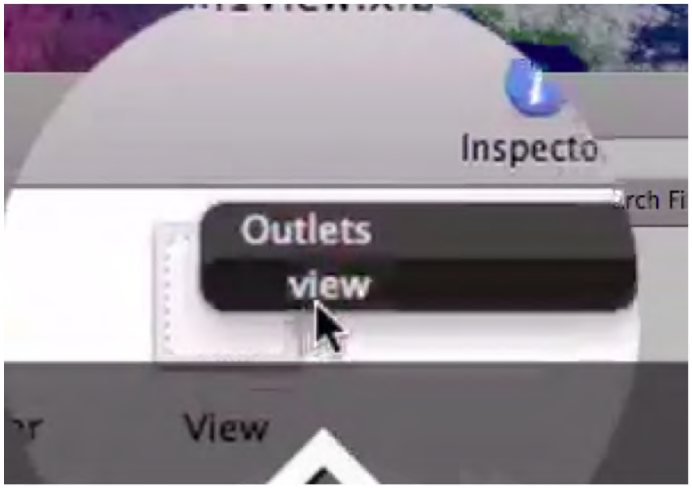


Figure 6–48. *Connect the File's Owner with the View option in the View.*

Repeat Process for Second Image

We need to repeat the same steps shown in Figures 6–42 thru 6–48 in order to connect the second photograph, with both my grandparents, to `Ein2Controller`. Begin this final series of actions by opening the `Einstein2View.xib` file and going back into Interface Builder, as shown in Figure 6–49.

Just as for the first image, we need an Image View to house the second photograph. So, drag it onto the screen of the `Einstein2View.xib`, and, as we did before, click on the File's Owner icon, and associate it with the code in the `Ein2Controller`. Control-drag the File's Owner icon of this second character to its View. Recall that when you control-drag over the View icon, a drop-down menu appears with View as an option. Point to this option and then release. With this done, we have completed working on the `Einstein2View.xib` file. Enter ⌘S to save your work, and enter ⌘Q to quit Interface Builder in order to return to Xcode.

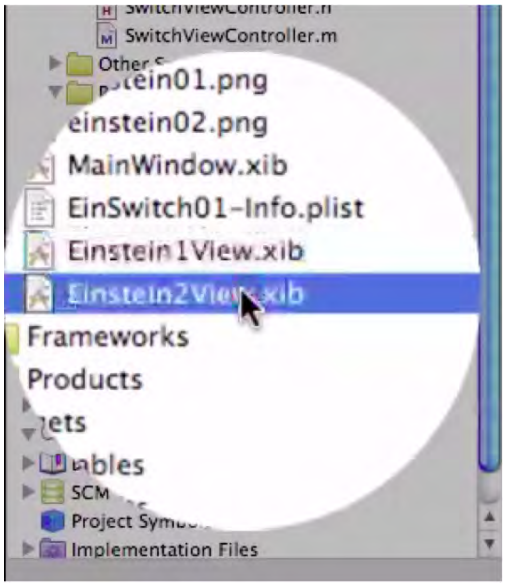


Figure 6–49. *Repeat the process for the second image.*

You are done. *Congratulations!*

Enter ⌘↩ to run the code. Pressing the button changes the picture … by switching the views … just as you programmed it to do. Well done! What you have just accomplished is really one of the most essential, non-trivial benchmarks of iPhone/iPad programming. By virtue of the fact that you can now code a Switch View app—to show one portion of code in one view and another portion of code in another view—you are immediately elevated beyond the novice level of programming. Figures 6–50 thru 6–54 show the fruits of your labor.

Figure 6–50. *Click the Switch Views button.*

Figure 6–51. *Hooray, our app works!*

Figure 6–52. *We see the initial image, the bachelor, in the iPad's* iPhone *View.*

Figure 6–53. *This is the initial image in 2x mode.*

Figure 6–54. *Here we see the switched view, the happy couple, in the iPad mode.*

Remember, we've only done the first of the three Switch Views examples!

1. ~~***From Scratch***: einSwitch_001~~

   ~~**Video:** http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/006_einSwitch_001.htm~~

   ~~**Correct Code:**~~
   ~~http://www.rorylewis.com/xCode/006a_einSwitch01.zip~~

2. ***Tab-Bar***: einSwitch_002

   Video: http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/007_einSwitch_002.htm

   Correct Code:
   http://www.rorylewis.com/xCode/006b_einSwitch02.zip

3. ***Custom Tab-Bar***: einSwitch_003

   Video: http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/008_einSwitch_003.htm

   Correct Code:
   http://www.rorylewis.com/xCode/006c_einSwitch013.zip

Holy cow! You should feel good about this accomplishment.

The hardest of our three exercises is behind us! Most people never do switch views this way. They do it the no-brainer way. Boy, is this next example going to be easy for you!

So come on—let's go for the *no-brainer* Switch View app. Check out how easy it is compared to what you've just done.

einSwitch_002—a Tab-Bar Application

As always, let us start off with a clean desktop. All you should have on your desktop are the `einstein01.png` and `einstein02.png` images from the `006_Chapter_6hEinSwitch01.zip` you downloaded for the previous example. Delete, or just file away in your Resources folder, the 57 × 57 pixel icon and the Lazy Load text file. Save the two images on your desktop so that it is organized like mine, as shown Figure 6–55.

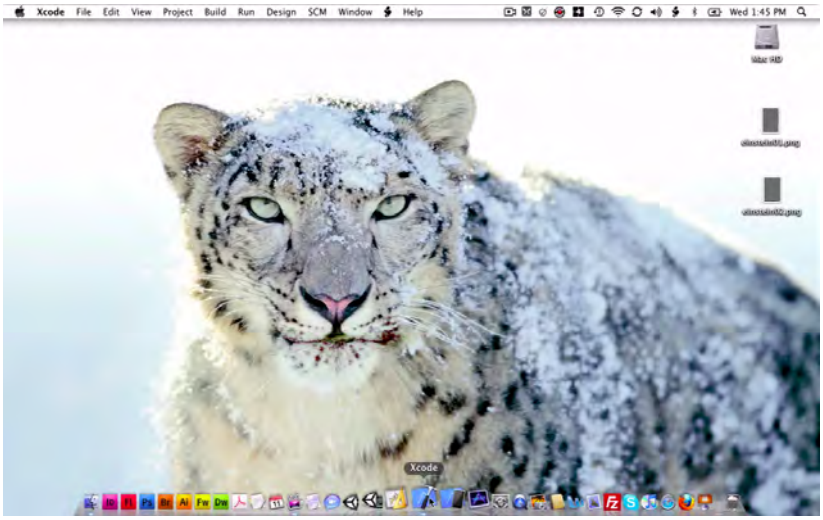


Figure 6–55. *A clean desktop at the start of the einSwitch002 example. We see only three icons: Mac HD and two image files.*

1. Open Xcode and enter ⌘⇧N as shown in Figure 6–56. This will open the New Project window in which you will click on the Tab Bar Application template. Name this project `einSwitch002`, as shown in Figure 6–57.

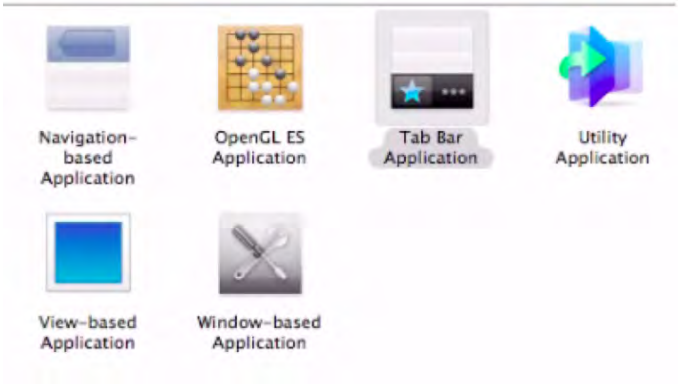


Figure 6–56. *After entering ⌘⇧N, create a Tab Bar Application.*

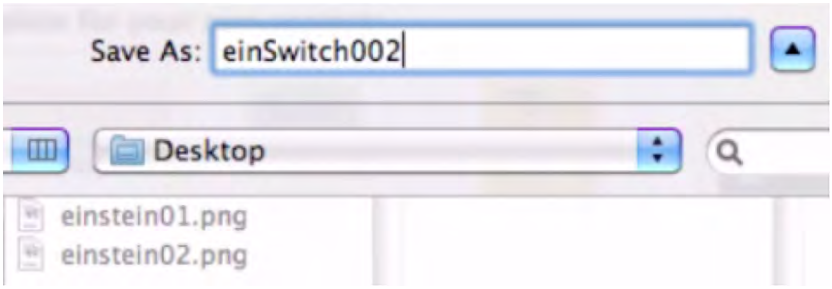


Figure 6–57. *Name it "einSwitch002" and save it to your Desktop by entering ⌘S.*

2. Open up your Resources folder. Go back to the desktop and select both of your images and drag them into your Resources folder as shown in Figure 6–58. While in the Resources folder, open up the `MainWindow.xib` file as shown in Figure 6–59.

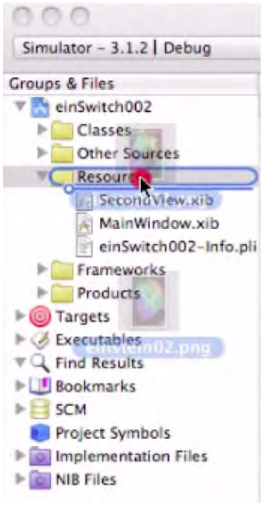


Figure 6–58. *Drag the two image files into your Resources folder.*

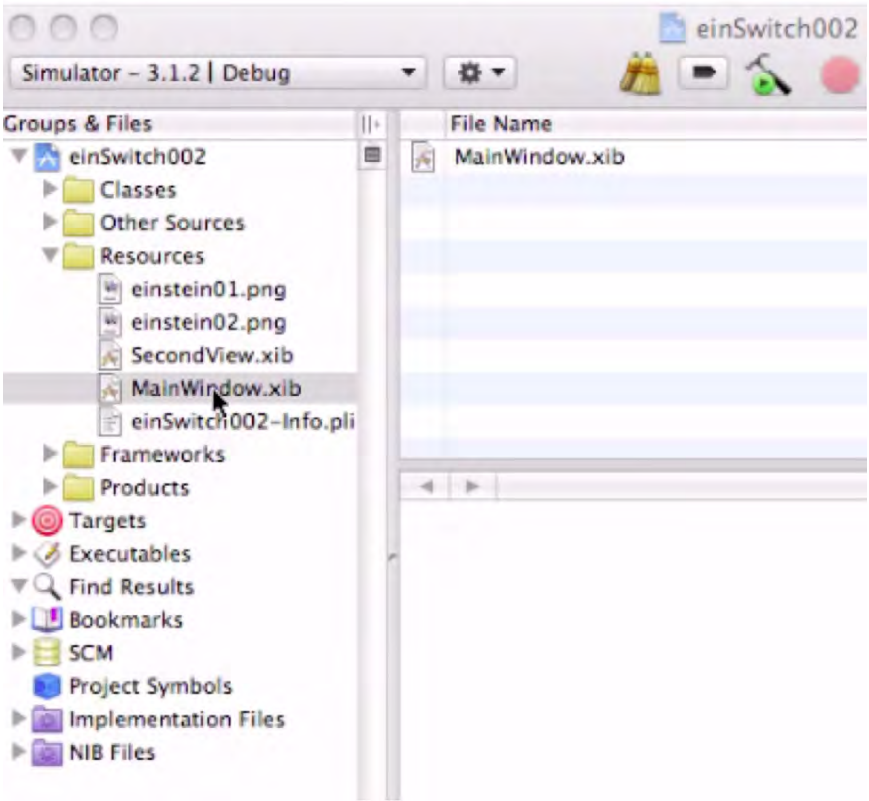


Figure 6–59. *Open the MainWindow.xib file.*

3. We will first work on the Main View, and then the Second View. Double-click on the `MainWindow.xib` file, which will automatically open Interface Builder. Once it opens, start by taking out the default text of the Tab Bar Controller frame, as shown in Figure 6–60.

   Once you've deleted all the default text, bring in a `UIImageView` from your Library, as shown in Figure 6–61. Once this object is placed in the screen, and while it is still active, go to your Image View Attributes window and select the `einstein01.png` file from your two choices, as shown in Figure 6–62.

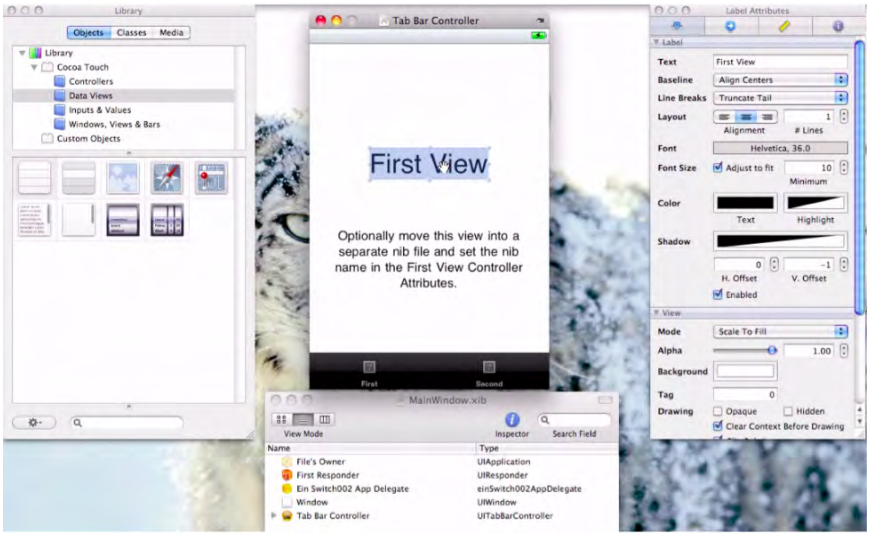


Figure 6–60. *Remove all of the default text from the Tab Bar Controller frame.*

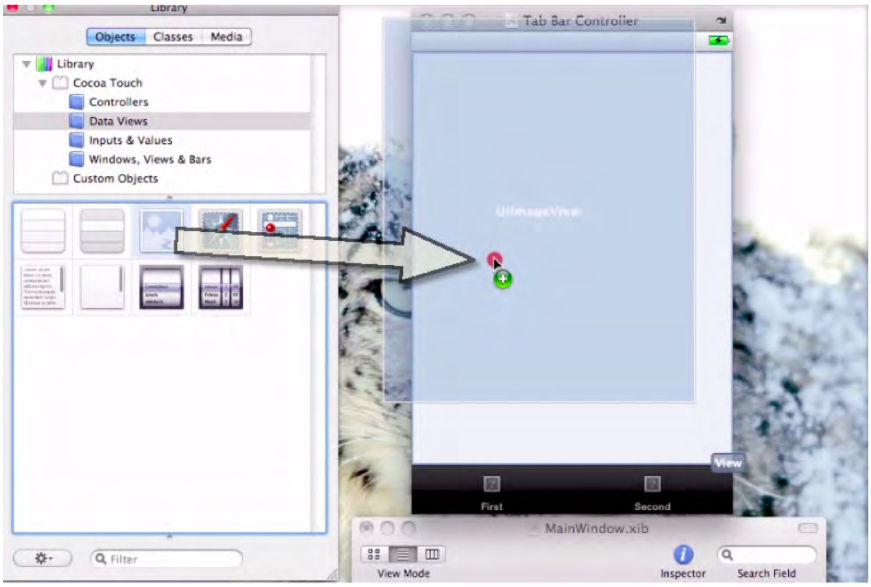


Figure 6–61. *Drag in a UIImageView from your Library.*

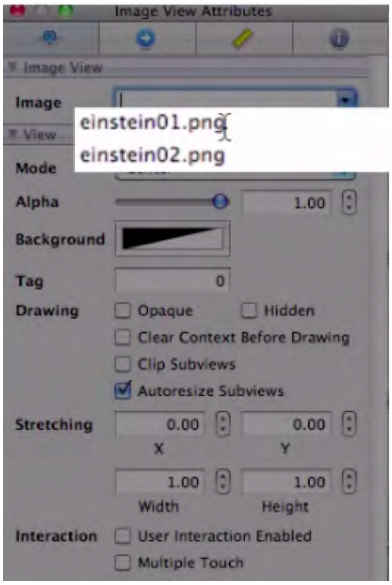


Figure 6–62. *Select the einstein01.png file from your two choices.*

4. Click the bottom-left black tab labeled "First," and then immediately go to your Tab Bar Item Attributes window. Click on the Title cell to delete the default title “First,” and then enter *Einstein01*, as shown in Figures 6–63 and 6–64. You may have to click twice in the cell to place your cursor properly.

   You know … it probably took you several hours to get to this point in the first example, coding this all from scratch. This is a bit easier, isn’t it?

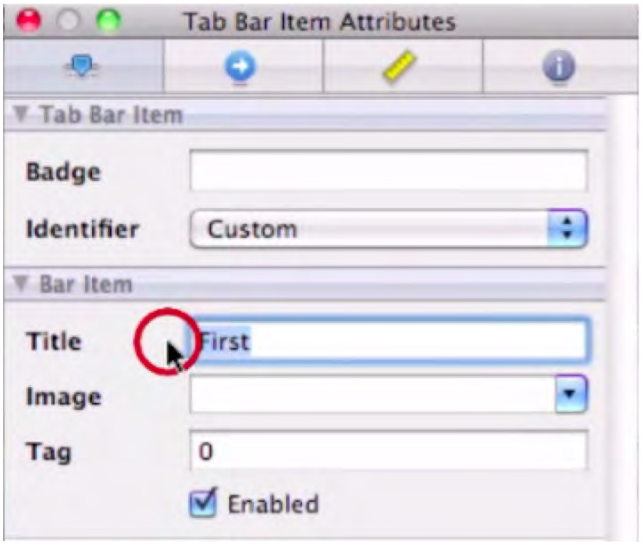


Figure 6–63. *Delete the left tab’s default name.*

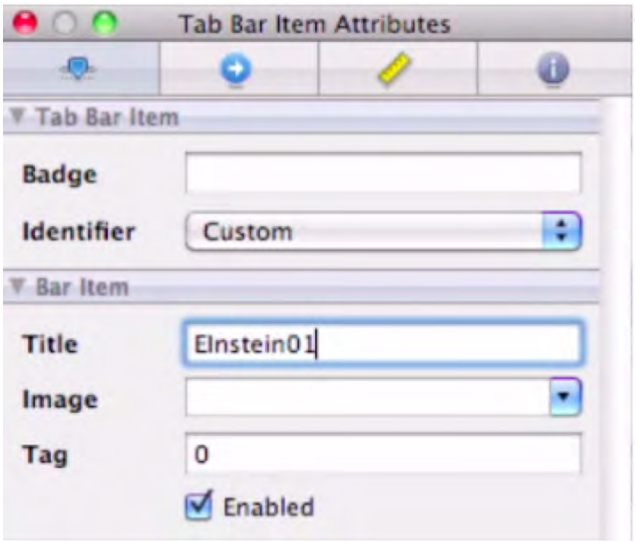


Figure 6–64. *Enter the left tab's new label: Einstein01.*

5. Click on the right-side black tab labeled "Second," and then immediately go to your Tab Bar Item Attributes window. Click on the Title cell to delete the default title "Second," and then enter *Einstein02,* as shown in Figures 6–65 and 6–66.

Hit Return. Save it and go back to Xcode.

Figure 6–65. *Click the Second tab button in order to repeat the steps.*

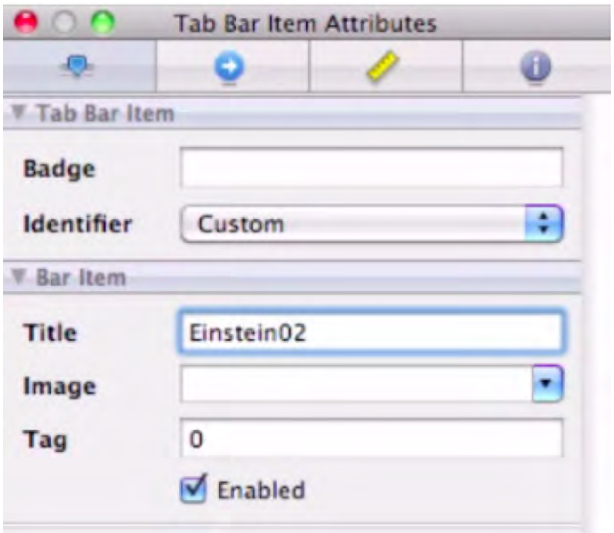


Figure 6–66. *After deleting the default, enter the left-hand tab's new label: Einstein02.*

> **NOTE:** When you start a Tab Bar application, the first View is called the Main View, and then those that follow are called the second, third, and so on. We've just connected the dots for the first View: the *Main View*. If we want the user to click on a tab that leads to another view, such as the second photo (both grandparents), we need to configure that View here.

6. Double click on the `SecondView.xib` file in your Resources folder as shown in Figure 6–67. As you did before, delete all the text on your Second View. See Figure 6–68.

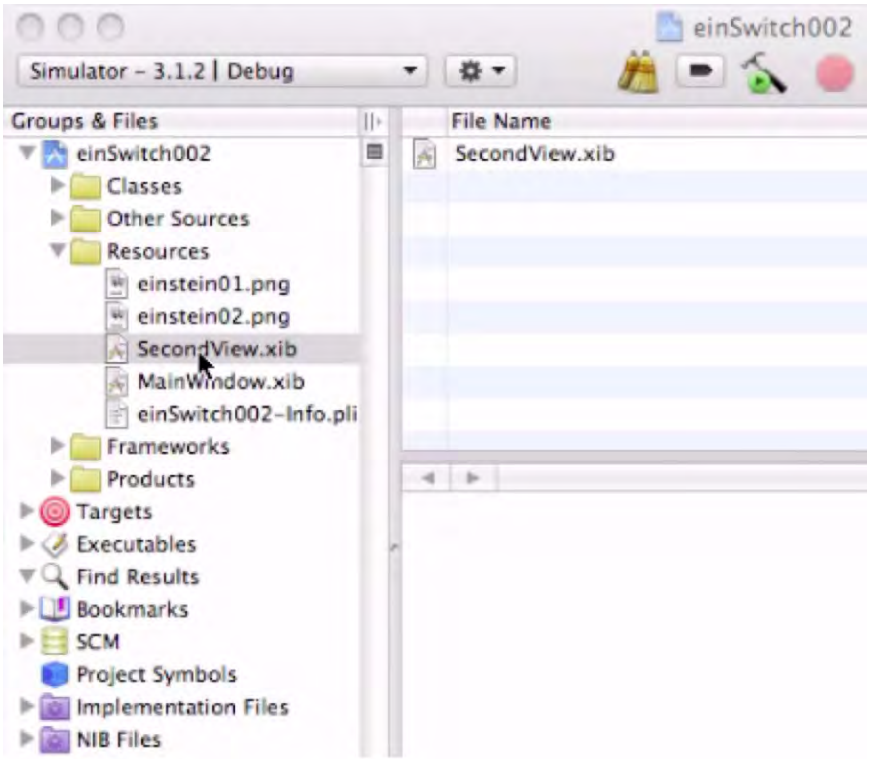


Figure 6–67. *Double click on the* `SecondView.xib` *file in your Resources folder.*

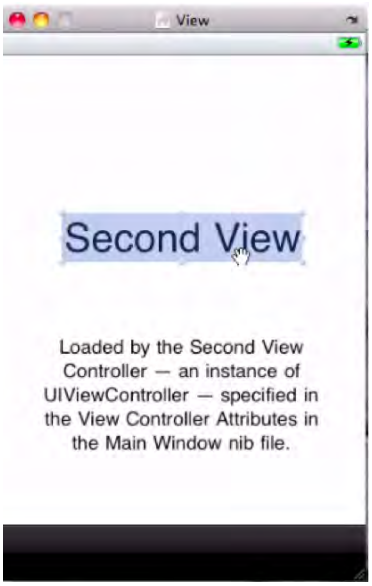


Figure 6–68. *Delete all the default text in your Second View.*

Drag in a UIImageView from your Library as shown in Figure 6–69 and place it inside the View frame.

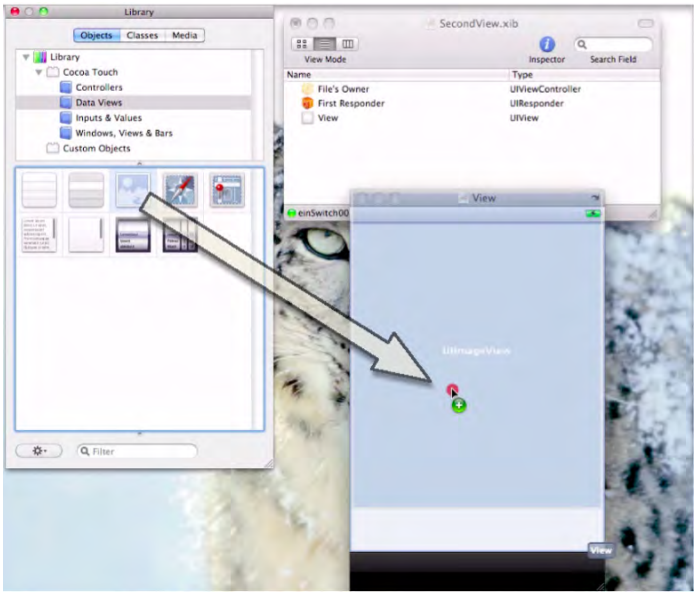


Figure 6–69. *Place a UIImageview on the Second View.*

While the screen is still active, go to your Image View Attributes window and select the `einstein02.png` image file, as shown in Figures 6–70 and 6–71.

Save your work by entering ⌘S, and then return to Xcode.

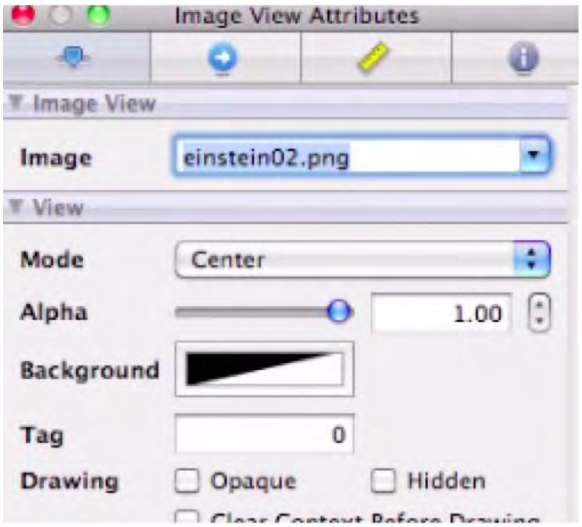


Figure 6–70. *Select the einstein02.png file from your two choices.*

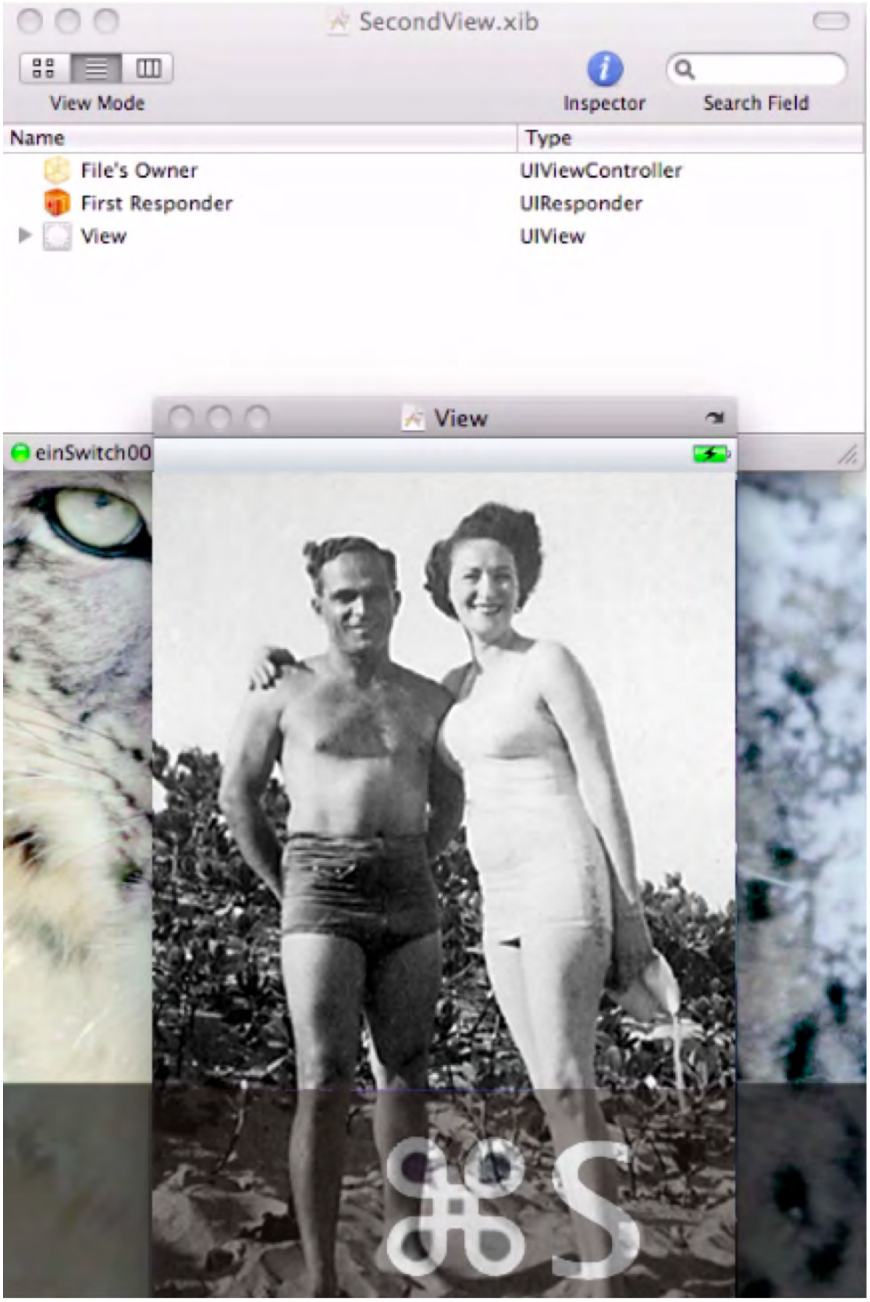


Figure 6–71. *Save your work by entering ⌘S.*

7. Go back to Xcode, open up your Classes folder, open a file, and Enter ⌘⮌ so we can run it as shown in Figure 6–72.

I know … you hate me for making you do the first example. If you had known that this *no-brainer* method was this easy, you would never have agreed to proceed with that one. Take it from me, though: students who depended on this simple path were not equipped to make any changes to their tabs or to be effectively creative.

Wait until the next example, einSwitch003. It is a hybrid of sorts, and *you will love it!*

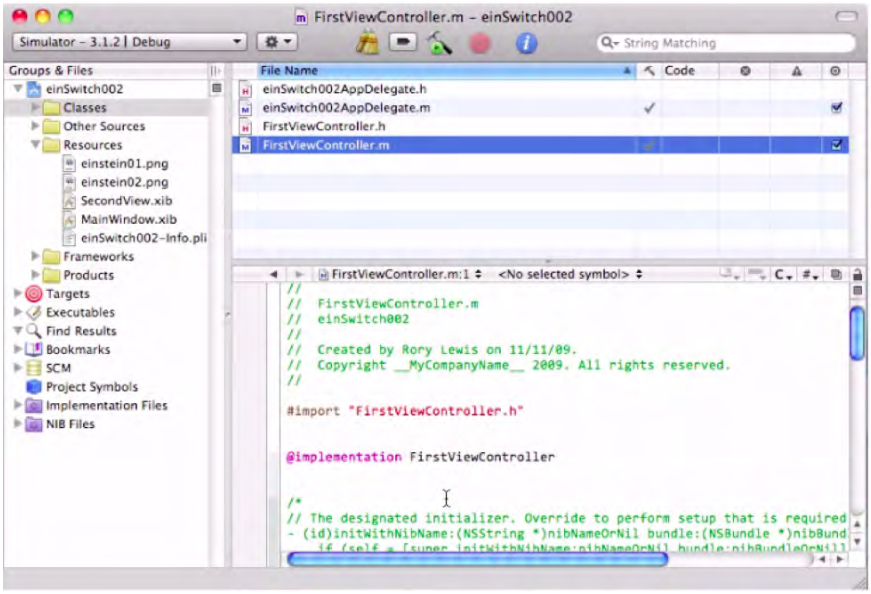


Figure 6–72. *Open up Xcode and run your code by entering⌘⮌.*

As you can see in Figure 6–73, the no-brainer method yields very decent results. The initial image is displayed—and the *Einstein01* tab is highlighted.

Figure 6–73. *Without a letter of code, our Tab Bar app appears with beautiful black tabs.*

Figure 6–74 illustrates the switched view by virtue of clicking the other tab, *Einstein02*. The second image appears, thanks to the boilerplate code you activated.

Figure 6–74. *These beautiful tabs work just fine—and they switch views without any coding!*

In Figure 6–75, we see the same nice results: the initial image is displayed and the *Einstein01* tab is highlighted, but in the iPad mode … with the iPhone view activated.

Figure 6–75. *The black tabs look very cool in the iPad's* iPhone *View.*

Figure 6–76 demonstrates that the iPad mode has no problem switching the view; the *Einstein02* tab is highlighted, and we see both grandparents.

Figure 6–76. *Even with black and white images, the iPad has a sophisticated look.*

Figure 6–77 shows the 2x magnification of the initial image. The resolution is crisp as the *Einstein01* tab is highlighted.

Figure 6–77. *The No-Brainer Tab Bar app works beautifully in the iPad's full view.*

The final variation is depicted in Figure 6–78. The no-brainer method appears to have done its job … and, of course, with minimal effort. Realize, though, that having completed the first exercise, your knowledge extends far beyond this simple two-tab project.

Possibilities are endless!

Figure 6–78. *Seamlessly switching in the iPad's full view. Not a single line of code!*

OK, we've completed two of the three switch view examples:

1. ***From Scratch***: einSwitch_001

   Video: `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/006_einSwitch_001.htm`

   Correct code: `http://www.rorylewis.com/xCode/006a_einSwitch01.zip`

2. ***Tab-Bar***: einSwitch_002

   Video: `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/007_einSwitch_002.htm`

   Correct code: `http://www.rorylewis.com/xCode/006b_einSwitch02.zip`

3. ***Custom Tab-Bar***: einSwitch_003

Video: `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/008_einSwitch_003.htm`

Correct code: `http://www.rorylewis.com/xCode/006c_einSwitch03.zip`

You've seen how easy the "no-brainer" method was, and you really appreciate it because the first approach was a royal pain. In this third iteration, I will show you how to tweak things a little bit. Your creative brain will connect the dots and find it relatively easy to make minor adjustments because *now* you understand the logic under the hood.

einSwitch_003—a Window-Based Application

The preliminary actions to take for einSwitch_003 are identical to einSwitch_002. The only items on your desktop will be the `einstein01.png` and the `einstein02.png` image files from the `006_Chapter_6hEinSwitch01.zip` you downloaded for einSwitch_001. Your desktop should be organized like mine, as shown in Figure 6–79.

Figure 6–79. *Your desktop at the start: Mac HD and two image files.*

1. Open Xcode and enter ⌘⇧N as shown in Figure 6–80; this will open the New Project window, in which you will click the Window-Based Application template. Name this project "`einSwitch003`," as shown in Figure 6–81.

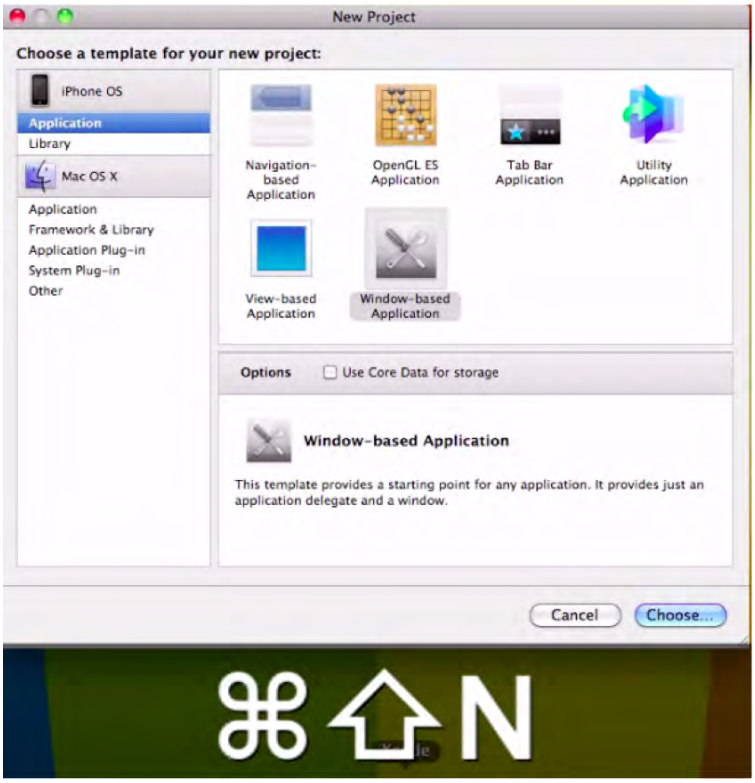


Figure 6–80. *After entering ⌘⇧N, select the Window-based Application template.*

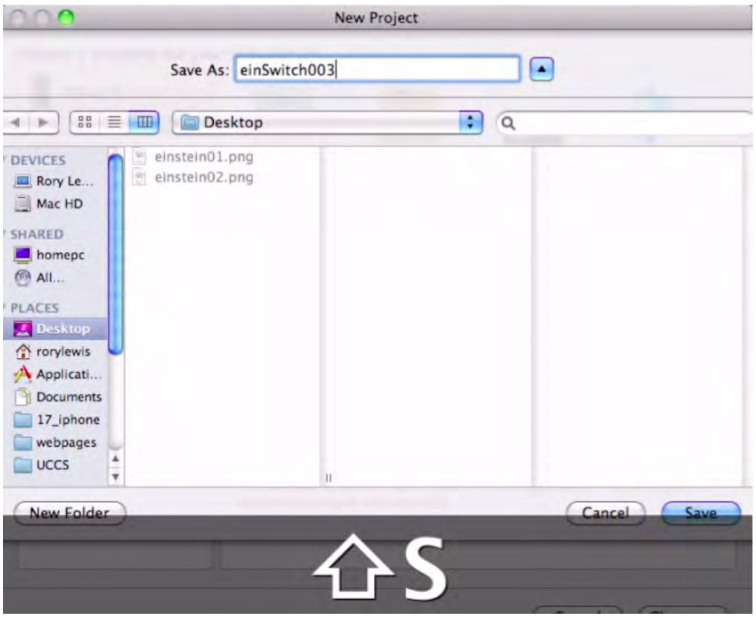


Figure 6–81. *Name your new project "einSwitch003," and save it to your desktop by entering ⌘S.*

2. One of the first things we need to do is go to the Classes folder. Click it open, and then open the header file by clicking open `einSwitch003AppDelegate.h`. Let's take a look at some of the places where modifications and customizing of the existing code might potentially serve us.

We need to bring in as much of the cool stuff that made the no-brainer methodology work while still being able to keep control over things.

Let's take a look at some boilerplate code that some clever Apple programmers have written, a chunk of code that is encapsulated in a Class Reference called the `UITabBarController`. It has all the code prewritten that displays tabs for selecting between different modes and for displaying the views for that mode. The `UITabBarController` class inherits from the code you programmed yourself in the first example, specifically the `UIViewController` class. Tab bar controllers have their own view made accessible through the `view` property.

See Figure 6–82 for a diagram that depicts how the views are assembled in the tab bar interface. We can change the look and feel of the tab bar, and toolbar views can change, but the views that manage these do *not* change.

Figure 6–82. *Apple's UITabBarController diagram. This arrangement looks simple at first, but trying to implement it can be quite complex … unless you have already completed this chapter's first example!*

So rather than struggling with this now, we will do three things:

1. Adopt the `UITabBarControlleDelegate` by inserting it into our `@interface einSwitchOO3AppDelegate : declaration`.

2. Add in a new declaration of `UITabBarController *tabBarController` after the `UIWindow *window` declaration, but before the } bracket.

3. Add a `UITabBarController` outlet.

   Once you have added these three steps, as shown in the boldface code below, enter ⌘S to save. These are shown in Figure 6–83.

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

@interface einSwitch003AppDelegate : NSObject <UIApplicationDelegate,↩
 UITabBarControllerDelegate> {
        UITabBarController *tabBarController;
        UIWindow *window;
}

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

@end
```

Figure 6–83. *After adding a UITabBarController outlet, save your work by entering ⌘S.*

4. We now need to code the implementation file. The first things we'll do here are

   a. Synthesize our `tabBarController`

   b. Add a subview View controller

   c. Release the `tabBarController`

   Once you have added these three steps, as shown in the boldface code following, your result should appear as in Figure 6–84. After this is all done, enter ⌘S to save your work.

```
#import "einSwitch003AppDelegate.h"

@implementation einSwitch003AppDelegate

@synthesize window;
@synthesize tabBarController;


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

    [window addSubview:tabBarController.view];
    [window makeKeyAndVisible];
}


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

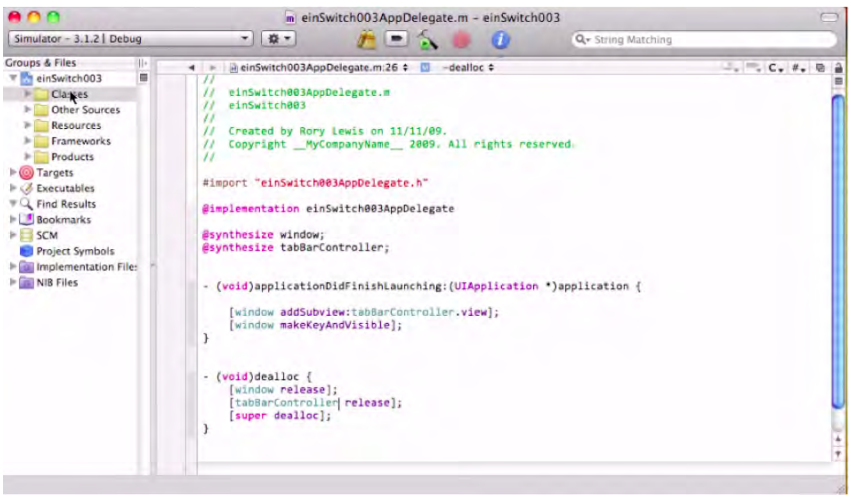


Figure 6–84. *After you synthesize the* `tabBarController`*, add a subview, and release the* `tabBarController`*, save your work by entering ⌘S.*

5. When we used the no-brainer method for the Tab Bar Application in einSwitch_002, it added all kinds of bells and whistles, files that we now need to add here. Let's think about this for a second. The challenge is this:

How can we figure out what files we need to add, but that, as yet, we do not have?

In the future, when you get to this crucial point, ask yourself: "How many views does my iPad/iPhone application need?" You may have eight different views in your program. This means you would have to add eight new views. We have two photographs, so we only need to add two views. We do this by going to the Classes folder, entering ⌘N, and selecting the UIViewController subclass, as shown in Figure 6–85. Make sure that you select the option "With XIB for user interface," as shown in Figure 6–86.

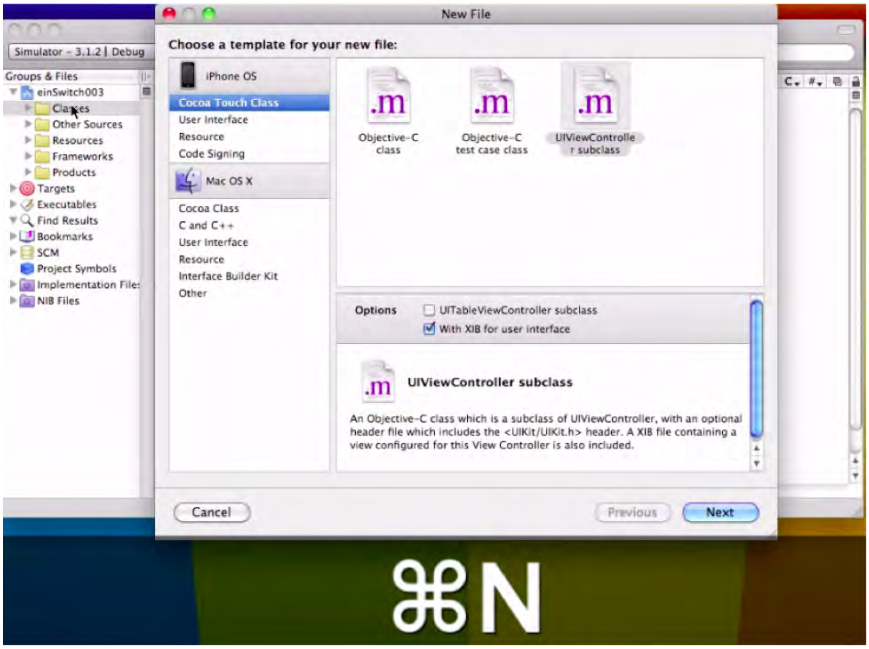


Figure 6–85. *In your Classes folder, enter ⌘N and select the UIViewController subclass.*

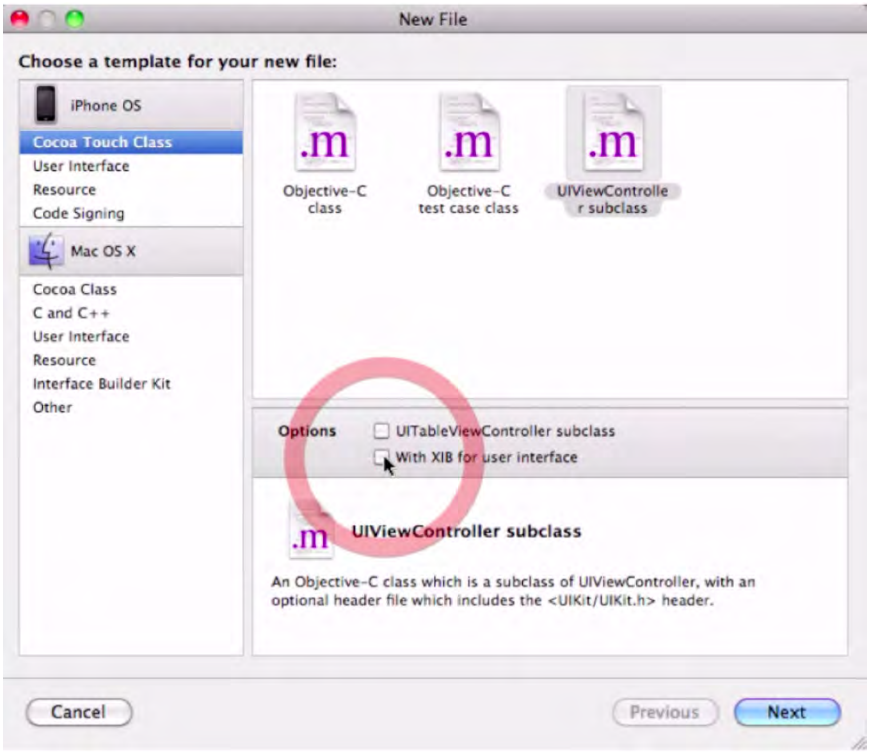


Figure 6–86. *Make sure you select the option to automatically make the nib file.*

Being incredibly creative, we shall name the first View Controller implementation file, `FirstViewController.m`, as shown in Figure 6–87.

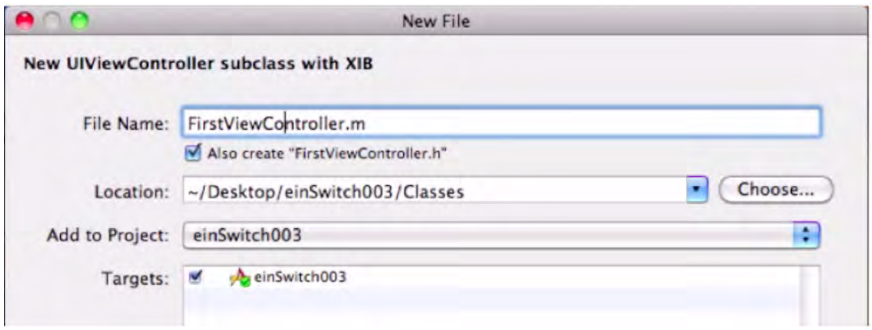


Figure 6–87. *Name the first UIViewController subclass—with XIB—FirstViewController.m.*

Now, we need to create and name the second View Controller. So, go back to your Classes folder, enter ⌘N, and select the UIViewController subclass as shown. Again, make sure that you check the option "With XIB for user interface." Save it as `SecondViewController.m`, as shown in Figure 6–88.

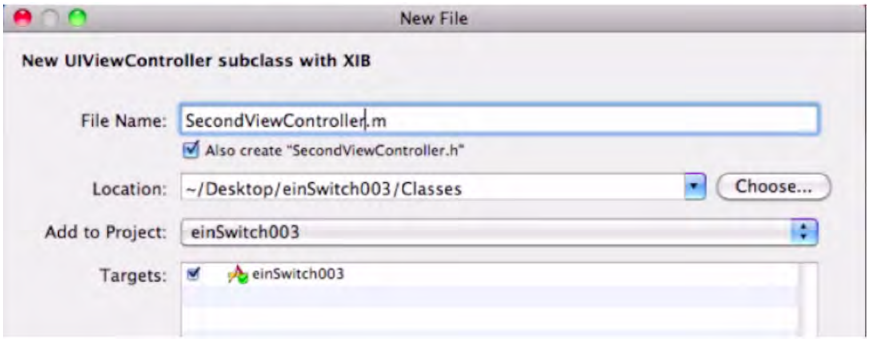


Figure 6–88. *Name the second UIViewController subclass—with XIB—SecondViewController.m.*

6. It's a very good habit to keep all your nib files in your Resources folder. However, the two automatically created nib files you just created are still in the Classes folder. Let's move them into your Resources folder, as demonstrated in Figure 6–89.

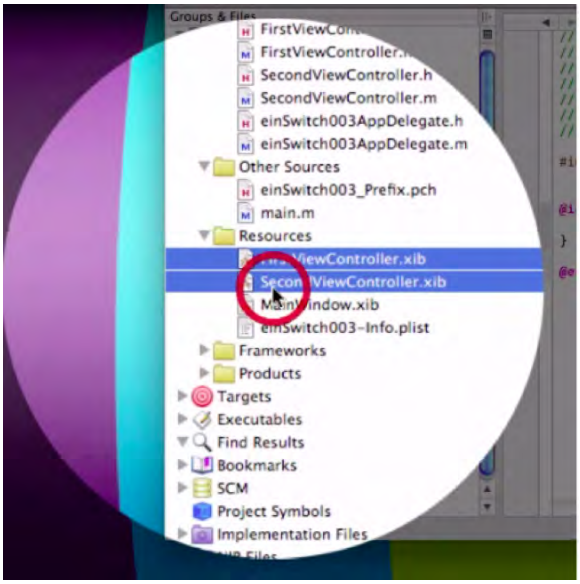


Figure 6–89. Drag the two new nib files into the Resources folder.

The next item on the agenda is to move our image files into the Resources folder. Go to your desktop, select both of our photographs, and drag them into your Resources folder, as illustrated in Figure 6–90. Make sure that, when you drop them into your Resources folder, you check the "Copy items into destination group's folder (if needed)" box. See Figure 6–91.

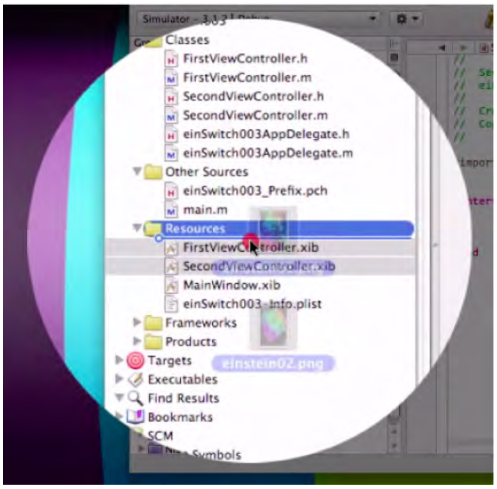


Figure 6–90. *Drag the two image files, einstein01.png and einstein02.png, into the Resources folder.*

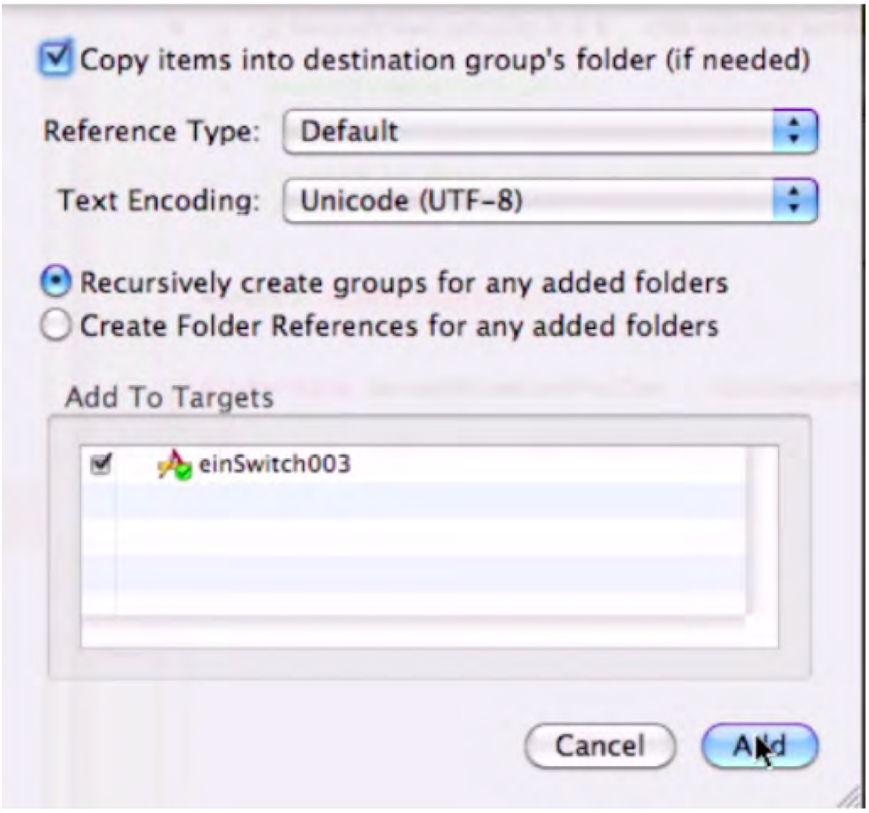


Figure 6–91. *Don't forget to check "Copy items into destination group's folder (if needed)"!*

TIP: This is (perhaps) the last time I will remind you of this. From here on, ***YOU will remember this!***

7. It's time to edit the ViewControllers. From your Resources folder, open the `FirstViewController.xib` file. When Interface Builder opens, drag an Image View onto your screen, as shown in Figure 6–92. Of course, we want the first image to show in the first View Controller. To accomplish this, go to your Image View Attributes Inspector and, in the Image drop-down menu, select `einstein01.png`, as shown in Figure 6–93. After this is done, enter ⌘S to save. Then, enter ⌘Q to get out of Interface Builder and return to Xcode.

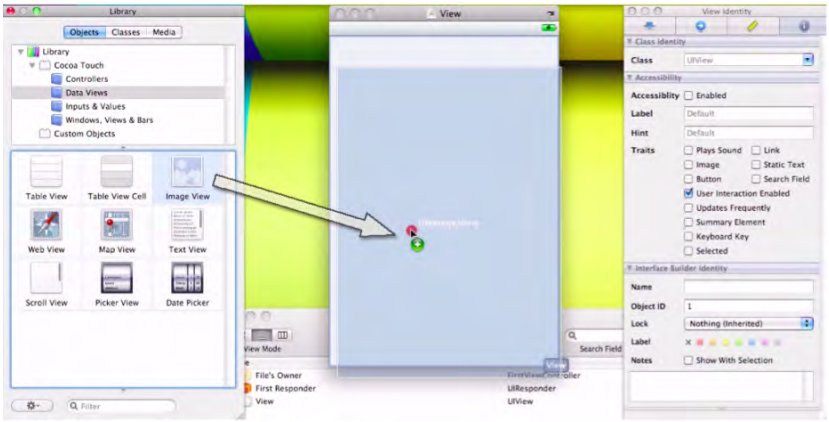


Figure 6–92. *Drag an Image View onto your screen.*

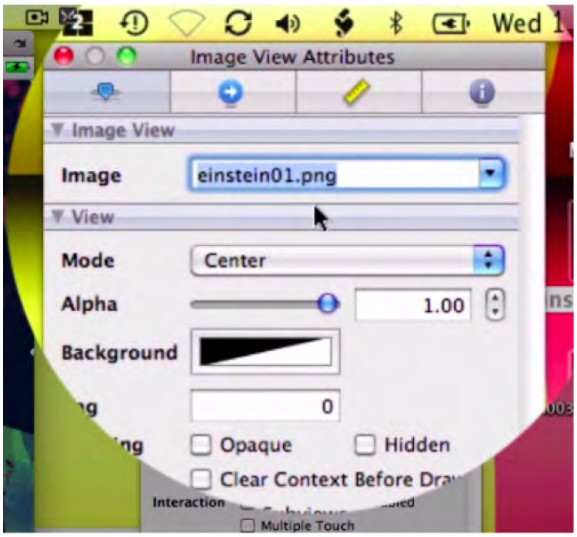


Figure 6–93. *Select einstein01.png from the drop-down menu.*

Of course, we need to repeat this entire process for the second View Controller file. So, open `SecondViewController.xib`, and then drag an Image View onto your screen. From the image drop-down menu, select `einstein02.png`. As before, enter ⌘S to save, and then ⌘Q to exit Interface Builder.

8. As indicated in Figure 6–94, go to the Resources folder and click `MainWindow.xib`. The first thing we need to do is to drag a Tab Bar Controller from the Library, into your Main Window frame, as shown in Figure 6–95.

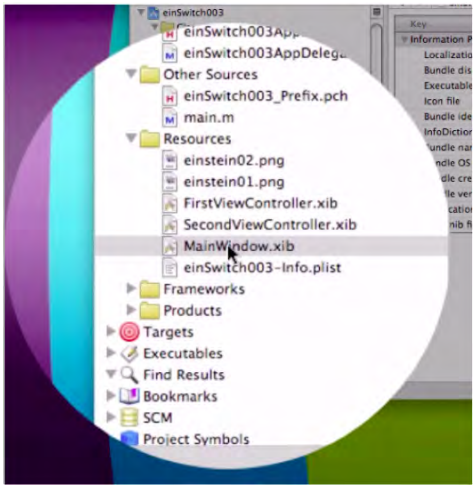


Figure 6–94. *From the Resources folder, open the MainWindow.xib file.*

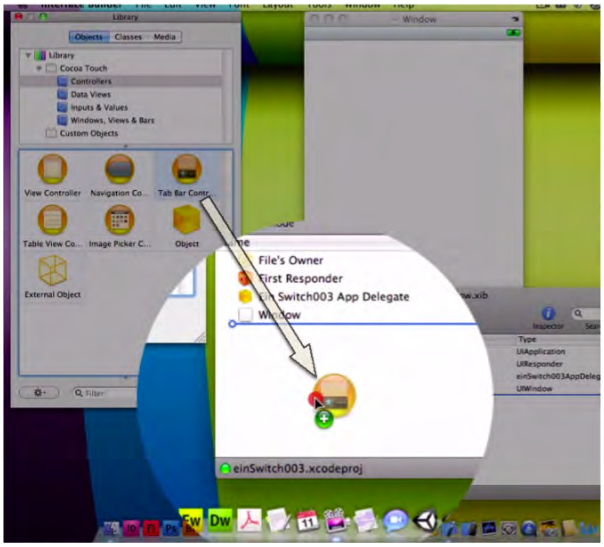


Figure 6–95. *Drag a Tab Bar Controller from the Library into your MainWindow frame.*

9. Next, we need to connect our Tab Bar to the `EinSwitch_003 App Delegate`. Control-drag from the `EinSwitch_003 App Delegate` to the Tab Bar Controller, as shown in Figure 6–96. As you pull in toward the Tab Bar Controller, the black drop-down menu appears with the option of one outlet being the `tabBarController`, which is exactly what we want. See Figure 6–97.

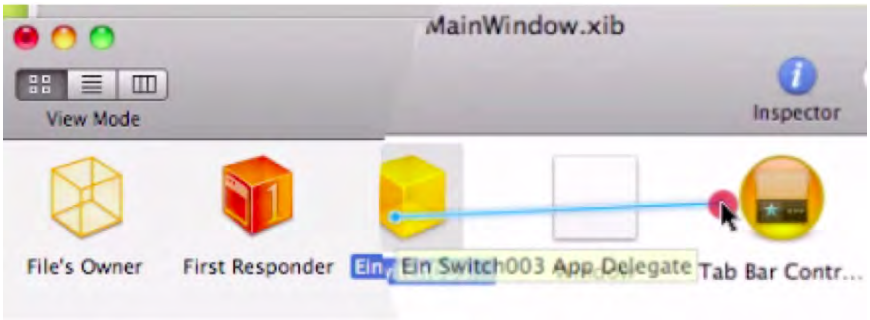


Figure 6–96. *Control-drag from the EinSwitch_003 App Delegate to the Tab Bar Controller.*

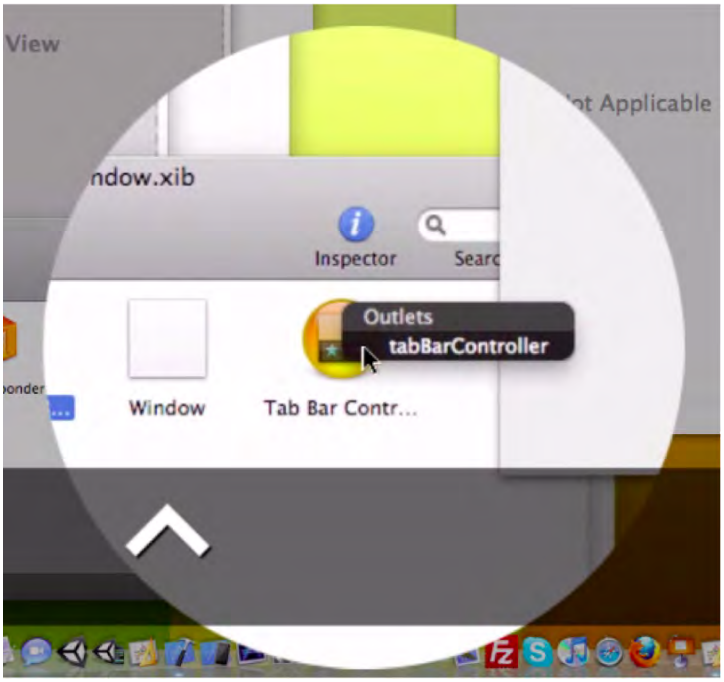


Figure 6–97. *There is only one option in the Outlets menu; select tabBarController.*

This is where we see that we have the best of both worlds: 1) We have control of what's under the hood, and 2) We have the luxury of having most of the code pre-written by Apple!

10. If this were a larger, more complex program, we could have connected with other controllers that, for example, change the icon pictures based upon where a user is in a game, or what language a user prefers within an app. There could be a million reasons why your program or game may need to have flexibility in the type of tab bar look and feel. Remember that, in the "no-brainer" approach, we had very few options. Here, though, with things looking still very much easier than in the first example, we see that control-dragging from one icon to another is relatively easy and intuitive.

    We're getting close to the end of our journey! Now, we want to connect our View Controllers to the correct nibs & UIViewControllers.

    We first need to expand the contents of our icons. Click your View Mode's middle button, located on the top left-hand side of your `MainWindow.xib` (Figure 6–98).

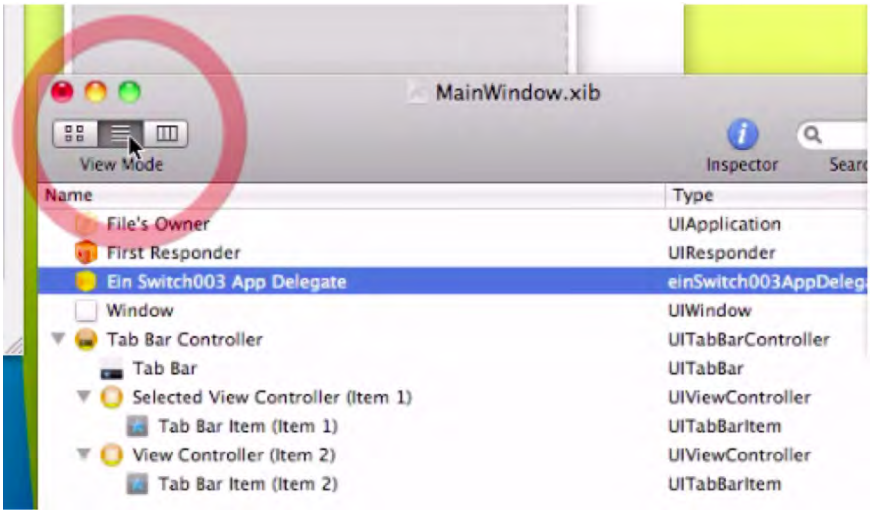


Figure 6–98. Select the middle button in the View Mode to expand the contents of the Tab Bar Controller.

    Select the first View Controller, labeled "Selected View Controller (Item1)," as shown in Figure 6–99, and give it a click.

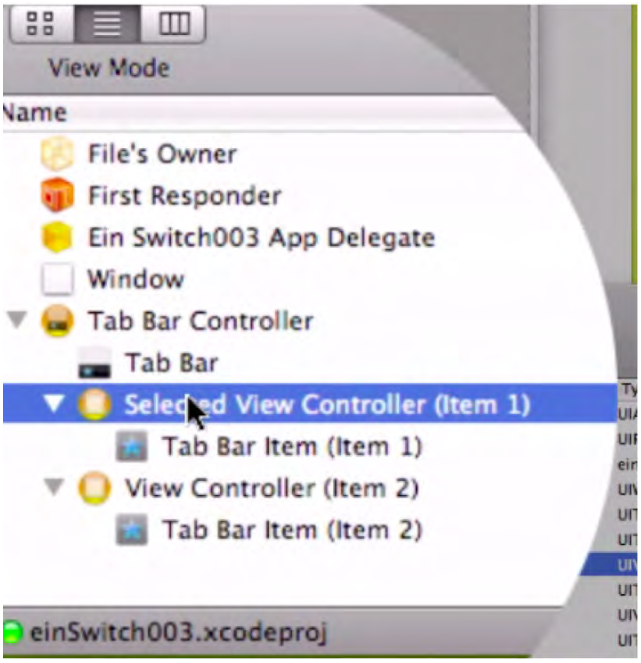


Figure 6–99. *Select the first View Controller: Selected View Controller (Item1).*

Then immediately go to your View Controller Attributes inspector, and connect it to the `FirstViewController` nib file, as shown in Figure 6–100.

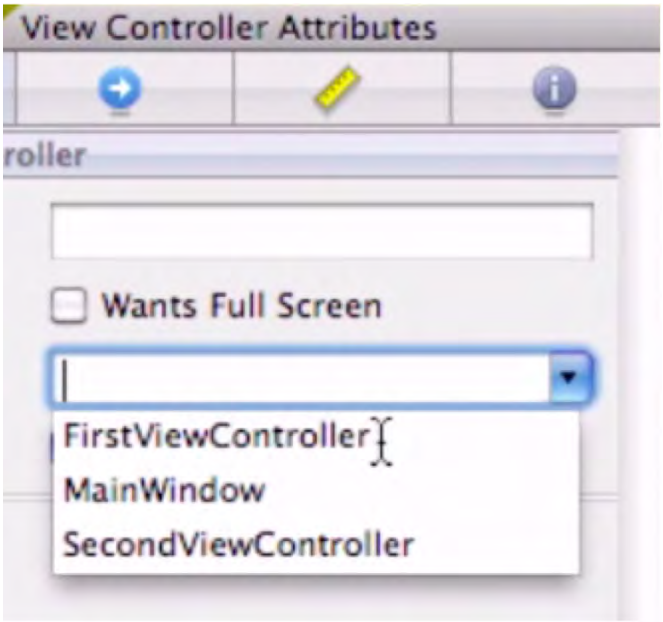


Figure 6–100. *Select the FirstViewController nib.*

Next, we need to connect the `UIViewController`. Go to the View Controller Attributes window and click on the Identity tab. Then, from the drop-down menu, as shown in Figure 6–101, select FirstViewController.

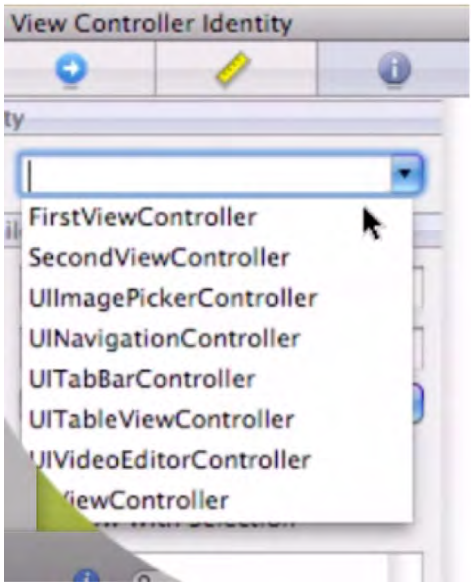


Figure 6–101. *Select FirstViewController from the drop-down menu.*

11. Connecting the second View Controller will be done in the same way. Of course, you will make sure to select the *second*, not the first, View Controller. Select it from the `MainWindow.xib`, as shown in Figure 6–102.

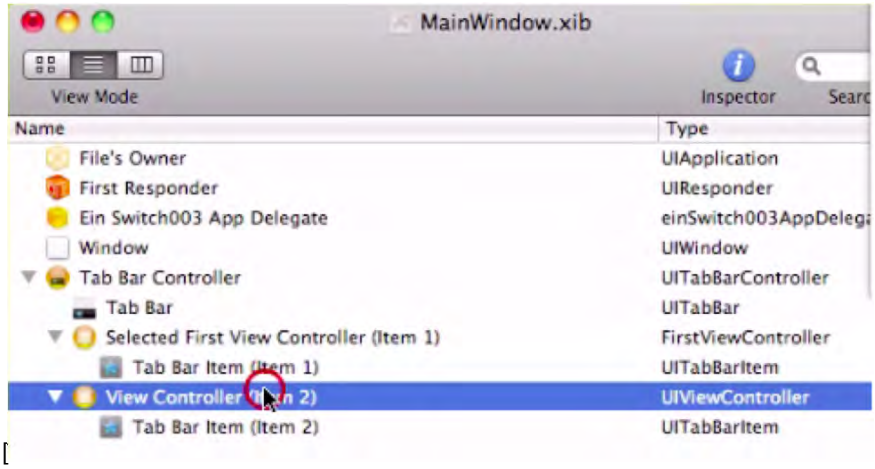


Figure 6–102. *Select the second View Controller, shown as View Controller (item2).*

With the View Controller Attributes tab still open, select SecondViewController from the drop-down menu, as shown in Figure 6–103.

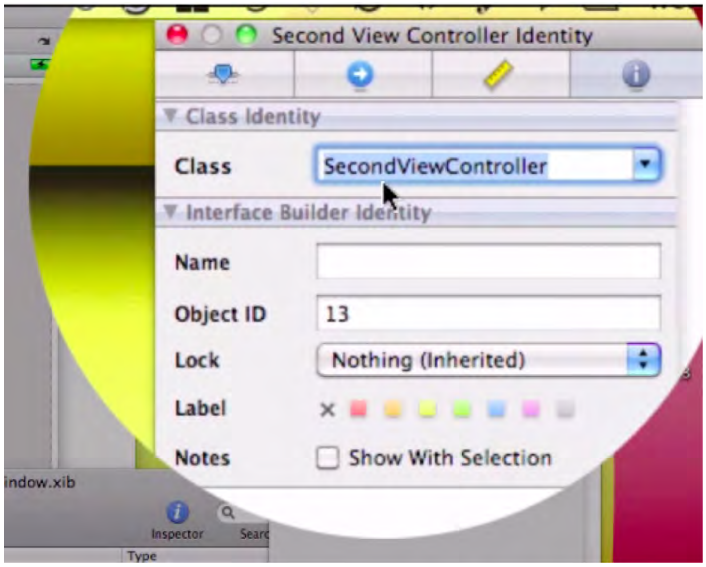


Figure 6–103. *Choose SecondViewController from the Class drop-down menu.*

Our last action, before running the code, is to choose the `FirstViewController` nib and enter ⌘S to save all of your work, as shown in Figure 6–104. Then, enter ⌘Q and go back to Xcode to compile your code.

Wow—*you did it!* Figure 6–105 shows the result.

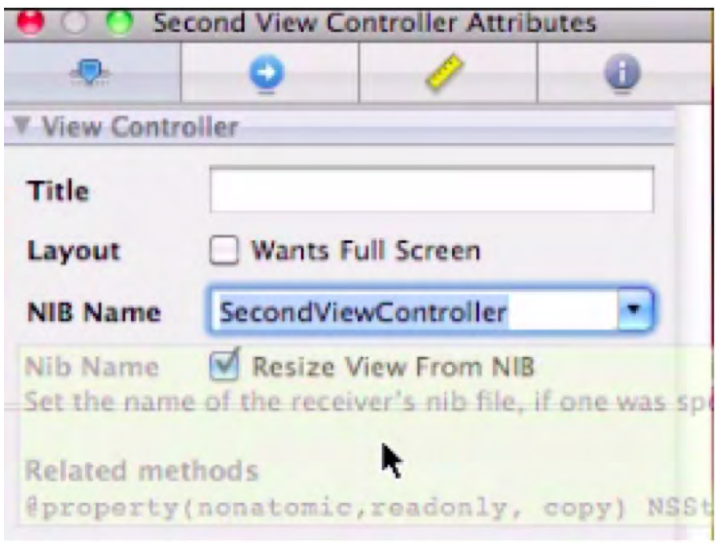


Figure 6–104. *Select the SecondViewController.xib, and save your work by entering ⌘S.*

Figure 6–105. *With your expanded understanding, and your willingness to use a few ready-made tools, the result is SUCCESS! Switch view programming is now within your grasp.*

Digging … Your Brain

Years ago, when I was learning how to program in C as a struggling electrical engineer at Syracuse University, I practiced these steps again and again. It took me literally two weeks of 6–10 hours per day to become fluent in the C programming language; I worked with handmade objects, classes, and methods that would run a larger operating system. It was a semester-end project.

Over the break, I repeated the whole program again, referring to my notes often. I then created a scorecard and made a dash each time I looked at my notes. When I went through the program the third time, I think it took me about 5 hours or so. I forget how many times I redid the code, from scratch, but I was eventually able to code the entire program in just under an hour without making a single reference to my notes. Most of the time, it was my terrible typing that held me up.

Yes, I am recommending that you do the same with this Switch View app. Take all the items off your desktop, except for the four files with which we started. If you are willing to try this, I bet that—by your fifth time through—you will be looking at your notes fewer than 10 times. I am also confident that, *after 10 to 15 practice runs*, you will be able to code the whole thing, as I do, in less than five minutes—without reading any notes!

I know this may sound extreme to some of you, but my main point is that practice does indeed make perfect. This is true in the mental realm as much as it is in the physical one. I know you've heard this before: A mind is a terrible thing to waste.

The more committed you are now, the more assured your future success!

Chapter 7

Dragging, Rotating, and Scaling

In this chapter, we'll be tackling our ninth programming exercise together. This app will be one of your first to include an advanced feature of iPhone and iPad apps: the ability to drag, rotate, and scale objects on the screen with your fingers. This is just one of the unique features of the iPhone and iPad that have contributed to their phenomenal success.

The ability to interact directly with items on the screen—in an easy and intuitive way—is very important for your application. Capitalizing on this integral feature, by accommodating these natural actions, is what we're after in this chapter. We will consider these interactions from the underside of the application—in the same way that mechanics raise a car off the ground hydraulically to gain fuller access to the engine and transmission. After we tune the car up and activate these bells and whistles according to our design, and hand the customer the keys to his shiny new car, he will have no choice but to get in and say "Aaah, this multi-touch transmission is so fine—so *smooth!*" We, as developers and programmers, must know how to generate these capabilities … and experiences—at the source. So how do these components work?

The simple answer is mathematics; the complex answer is trigonometry. Matrix math and transforms make this kind of object and image manipulation possible. Now that you're on the verge of becoming an advanced iPhone and iPad programmer, you're in full geekdom at this point, and you can cope with looking deep inside the machine. Breathe it in. Embrace it!

Let's take a look at some of the guts of the iPhone OS, then, and get started. Our first consideration is "multi-touch." What does that mean? Why is it such a big deal?

> **NOTE:** As I move through the chapter, I sometimes seem to be referring only to the iPhone, but be aware that I include the iPad by implication – just as I include the iPod touch.

Back in the early days of hand-held computing, a person would use a plastic pen-like object called a stylus to interact with the device. This stylus was somewhat bothersome and tended to rebel, occasionally going on strike or getting lost. Runaway styluses were epidemic!

Well, the innovation of the multi-touch screen made this accessory obsolete. Multi-touch rendered the stylus irrelevant, for this super-sensitive surface tracks the touch of one's fingers, which, thankfully, tend to stay close at hand! OK, so that's the origin of the "touch" part of the name. What about the prefix, "multi-"? That is pretty straightforward, too, for humans generally have ten fingers, or digits. Although people tend to use only one at a time for pointing, some clever engineers realized that we could create or represent a host of new actions if we would just consider using some of those other fingers as well.

Thus, *Multi-Touch* was the term given to this platform (now patented), implying that a combination of simultaneous inputs is possible—and maybe also desirable. This immediately expanded the palette of choices, allowing for complex gestures and subtle interfacing. This was a big deal—and it still *is* a big deal!

The ingenuity and flexibility of this sexy, lightweight device—a tool that can be directed so quickly and intuitively with one's fingers—caught on, as you know, even before it hit the shelves. Multi-Touch is only one of the reasons that the iPhone and iPad are so popular, but it is our focus for this chapter.

Some of you are probably thinking, "Nice history lesson - but what's your point?" Basically, that it's OK to be excited about our work, and to acknowledge how cool it is. The hardware for which we are creating our apps is radical and magical, and I want you to harness your excitement and wonder, and live up to this standard. I'm here to help you translate that into something astounding and new—something your users have not imagined yet.

The first step toward this creative challenge is to interpret your users' input and decode their gestures. Of course, to do this elegantly, you need to understand how Multi-Touch input works. In this chapter, we will tell the iPhone and iPad processors what to do when the user pinches, or swipes, or taps the screen. Only upon understanding the input conventions can you begin to speak the touch-based language of your users.

DragRotateAndScale—a View-Based Application

As I mentioned in Chapter 3, the programming exercises in Chapters 7–9 are designed to continue the trend of advancement over the material in the earlier chapters. As with previous chapters, we have made available supplemental videos for these exercises, but you can program all of these apps without referring to the screencasts.

However, because the apps in these final three chapters are considerably more complex than earlier examples, you may want to audit some of them to give yourself some extra perspective. If you do check these out, you will see me, or a teaching assistant, zooming along at a pretty good clip. Don't hesitate to pause, stop, or rewind as many times as you need to in order to become comfortable with the coding sequences.

Something else to be aware of is that in the screencasts we use a lot of boilerplate code. We just paste the ready-made components directly into the apps in progress. But here, in the book, we go line by line through all of the code. So, check out the screencasts … and then set aside a big block of time to go through this app, step by step.

When you're done, try to build it on your own. Then keep on doing it, again and again, until you don't have to make a single reference to the book. You'll find the screencast at `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/009_Drag%20and%20Rotate.htm`

Preliminaries

As you did in Chapter 6, please download and extract images and code for this chapter. If you need refreshing on how to do this, review Figures 6–2 thru 6–4. As usual, start off with a spotlessly clean desktop. Then, open a browser and navigate to `http://rorylewis.com/xCode/009_DragRotateAndScale.zip` and download its contents to your desktop. Then, extract the files onto your desktop.

There will be four text files, one image file, and a folder containing the final working code for DragRotateAndScale (in the event you encounter trouble coding it yourself). The image file is the image I used in the example—of my puppy Shaka. The text files consist of sections of boilerplate code from which I will ask you to copy and paste various pieces. You are welcome to use these files in any of your future programs that involve touches. You'll find Translate.rtf, HelperFunctions.rtf, TranslateRotateScale.rtf, and ViewController.rtf.

Once you have extracted all the files, remember to delete the 009_DragRotateAndScale.zip and 009_DragRotateAndScale folders. Also, file the DragRotateAndScale Xcode to a safe place, for if you leave it on your desktop it will be overwritten and conflict with your exercise code. Monkeys will start writing Shakespeare, and the world will collapse and disappear … all because you did not file it away. After you follow these directions, you will have five files on your desktop.

Starting the DragRotateAndScale App

To start our DragRotateAndScale application, we need to make a new project in Xcode as we've always done. Open Xcode and select the View-based Application template item (as shown in Figure 7–1).

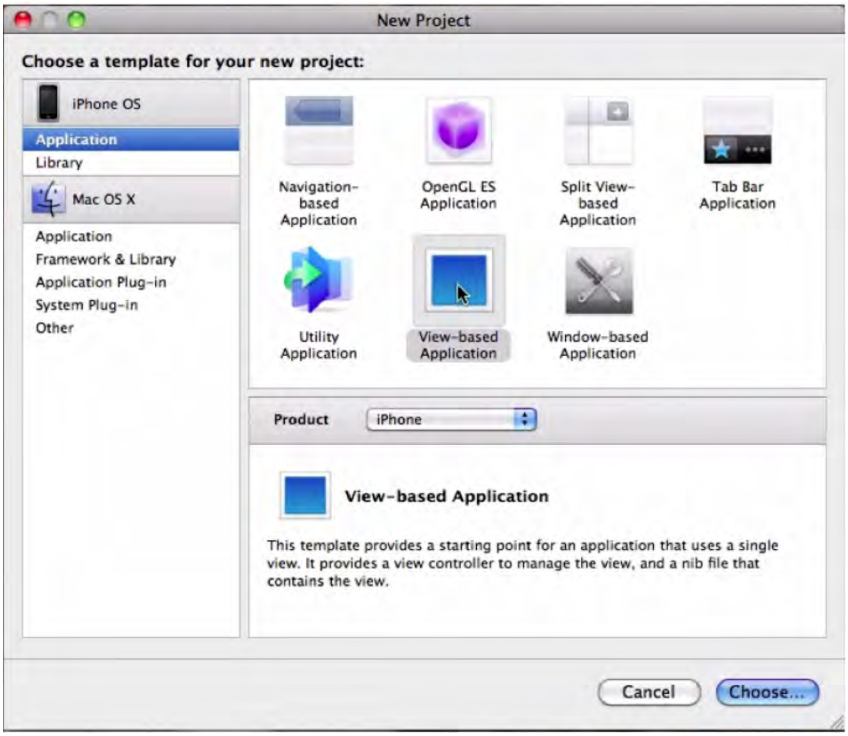


Figure 7–1. *Create a new View-based application to start the DragRotateAndScale project.*

Make sure the Product menu is set to iPhone before continuing. Name the project DragRotateAndScale and click the Save button, as shown in Figure 7–2.

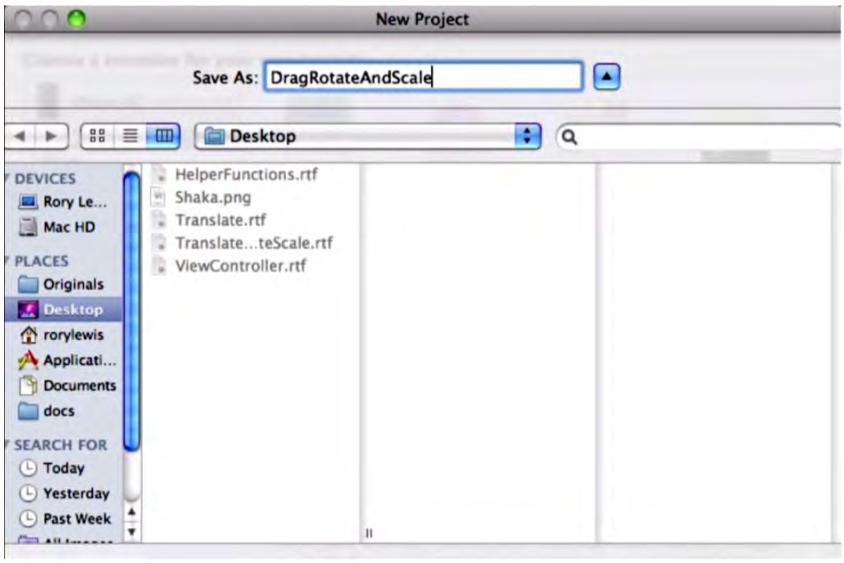


Figure 7–2. *Name and save the project.*

Next, choose and prepare an image file as your main object for this exercise. Choose whatever is appealing, and save it on your desktop. By the way, we have chosen a smaller image than earlier exercises have used (100 × 100 pixels), to allow for manipulating and sizing.

Copy the handy image file into the project's folder, as shown in Figure 7–3. Click the usual boxes and radio buttons to ensure proper management of this image file down the road.

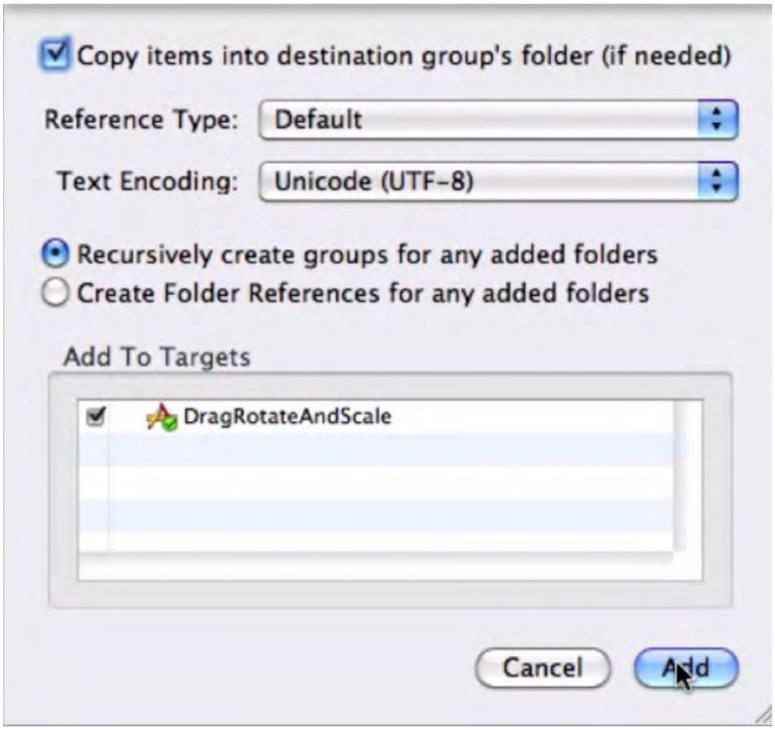


Figure 7–3. *Copy your desired image into the Xcode project. The image used in the example is 100 × 100 pixels.*

Creating a Custom UIImageView Subclass

We are going to add a new file to our project: a UIImageView subclass called TransformView. This class will intercept touch events and change their transforms accordingly. We are making TransformView a UIImageView subclass so that we can assign an image to our instance and then see it rendered on the screen. Technically, a UIView subclass can do what we want, but, for this example, we want to focus on the transforms and not worry about custom drawing code for a UIView subclass.

Creating a custom view isn't strictly necessary for this application, but we will do it anyway—to flex our subclassing muscles. The DragRotateAndScaleViewController could technically do everything we need to make this application work, without the need for a custom UIView subclass, but subclassing makes the code simpler and more robust. Thus, we select a new Objective-C class in the New File window, as shown in Figure 7–4.

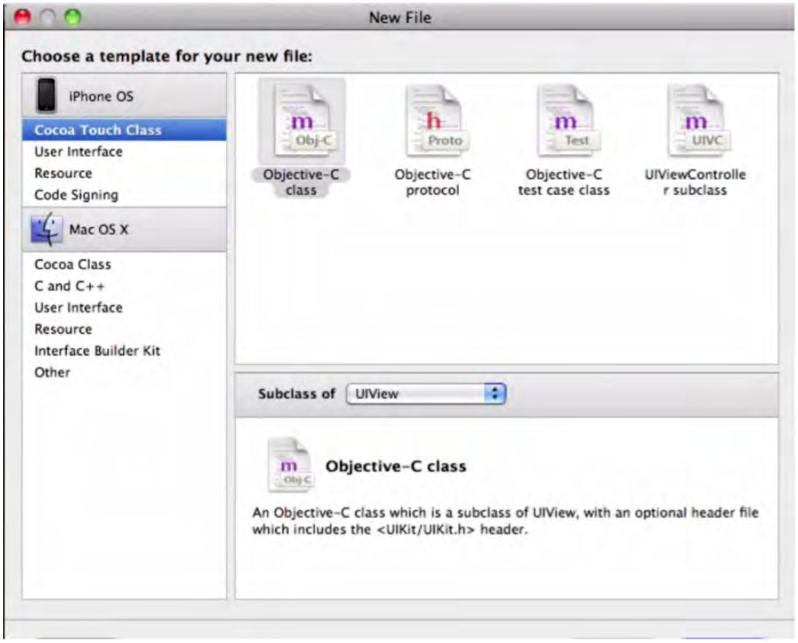


Figure 7–4. *Create a new Objective-C class file.*

In the header file, `TransformView.h`, make sure the superclass—that is, the statement after the colon (:)—is specified as `UIImageView`. We do this so we'll be able to assign an image quickly and easily. Your header file should look like the one in Figure 7–5.

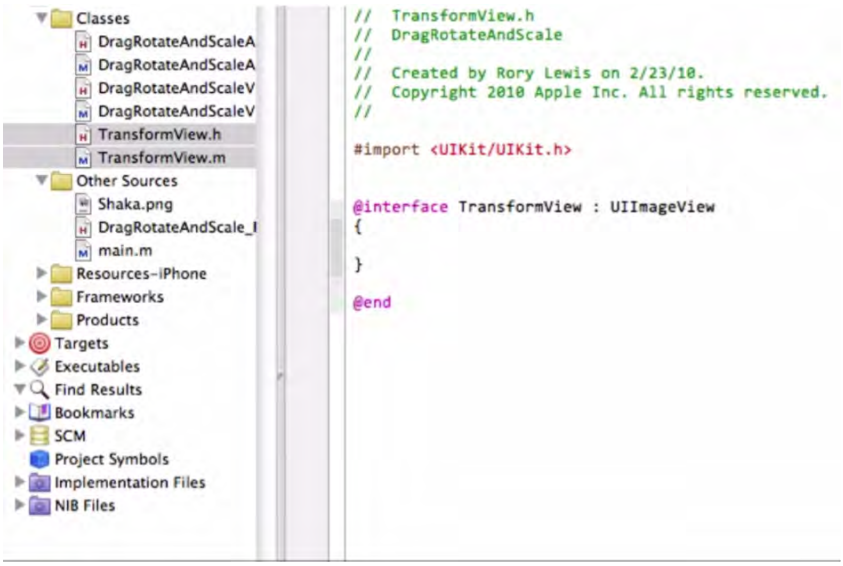


Figure 7–5. *The TransformView class, a subclass of* `UIImageView`

```
//
//  TransformView.h

#import <UIKit/UIKit.h>

@interface TransformView : UIImageView
{

}

@end
```

Overriding – initWithImage in TransformView.m

For the implementation file, `TransformView.m`, we want to handle the creation of our view a little differently than normal. UIImageViews do *not* handle touch input by default, and they therefore reject any touch input they receive. This would not serve us! We want our view to handle not only touch input, but multiple touches.

To do this, we override the method `(id) initWithImage:(UIImage*)image`. Inside of our override, we will insert the lines `[self setUserInteractionEnabled:YES]` and `[self setMultipleTouchEnabled:YES]`. Of course, we end each line with a semicolon.

The first line will allow our view to respond to touch events. The second will allow TransformView to receive multiple touch events, which we'll need if we're going to enable users to pinch and spread their fingers to scale and rotate. Your code should look like Figure 7–6.

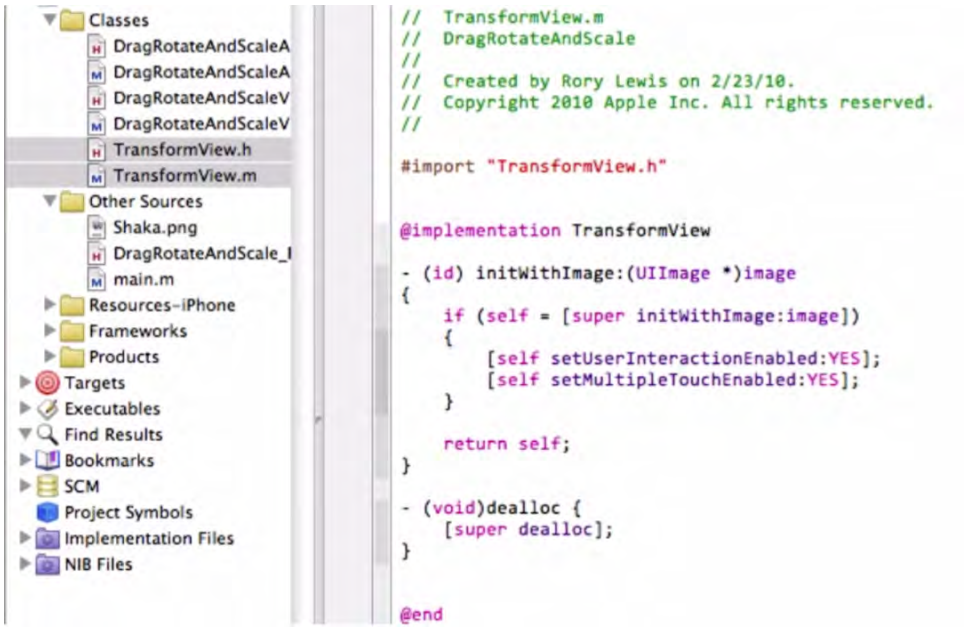


Figure 7–6. *Enable TransformView to process multiple events.*

```
//
//  TransformView.m

#import "TransformView.h"

@implementation TransformView

- (id) initWithImage:(UIImage *)image
{
        if (self = [super initWithImage:image])
        {
                [self setUserInteractionEnabled:YES];
                [self setMultipleTouchEnabled:YES];
        }

        return self;
}

@end
```

Creating Touch-Handling Stubs

Now we want to begin the process of creating the pieces of code that define the extent to which the users' touches affect the images and other interface parameters. To do this, we override various methods whose names literally start with the term "touches." These *touches* methods are called into action whenever a user touches the device, moves a touch, or stops touching the device. The four "reversal" methods we will add are these:

```
- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
- (void) touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event
- (void) touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event
- (void) touchesCancelled:(NSSet*)touches withEvent:(UIEvent*)event
```

> **NOTE:** You can see these methods overridden as *stubs* in Figure 7–7. A stub is a skeleton of a boilerplate set of code made to be easily adaptable to your code as a timesaving device. In our case, pasting these four stubs is an efficient means of programming when a touch begins, moves, ends, or is canceled.

Note that `touchesCancelled:withEvent:` calls, or refers, to `-touchesEnded:withEvent:`. This is because we generally want a *cancelled* touch to behave as though the user had *ended* that touch. "Cancelled" is what we call any user input (in this case a touch) that is called off by performing the identical input; for example, a button is pushed ON, but then pushed again OFF. An "ended" touch is an intentional input that is the cessation, or stopping, of an ongoing touch, or swipe, which proceeds for more than just an instant. This distinction may not always apply, but it's a good approach for our purposes.

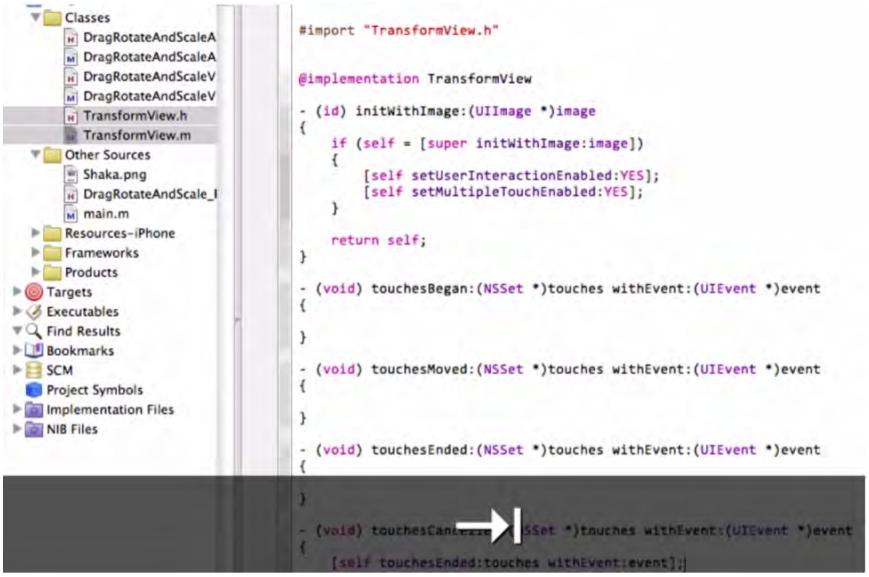


Figure 7–7. *The lines of stub code dealing with overriding defaults are inserted.*

> **NOTE:** We are not even going to *attempt* to have you understand every word of the *touches* code. It's enough to know that whenever you deal with touch-related code, and you are trying to decide which default code to keep and which to delete, you can forget your worrying and use these lines of ready-made and adaptable code known as "stubs." *Do not think too much about this; just use it!*

We may not end up adding details in all of our *touches* methods, but we want to override them all anyway. This is so that we are in complete control—so that we claim administrative ownership of the touches at all times. It is simply good practice.

```
//
//  TransformView.m

#import "TransformView.h"

@implementation TransformView

- (id) initWithImage:(UIImage *)image
{
        if (self = [super initWithImage:image])
        {
                [self setUserInteractionEnabled:YES];
                [self setMultipleTouchEnabled:YES];
        }

        return self;
}

- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
```

```
}

- (void) touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{

}

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{

}

- (void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
        [self touchesEnded:touches withEvent:event];
}
@end
```

Translating in *touchesMoved*

Let's start on the section of our code that will allow for moving the image around—in other words, the code that enables us to drag objects. This means that, for now, we are going to zero in on one line of code: `-touchesMoved:withEvent:`.

For the time being, the other *touches* methods will remain unchanged. Your *touchesMoved* method should look like Figure 7–8.

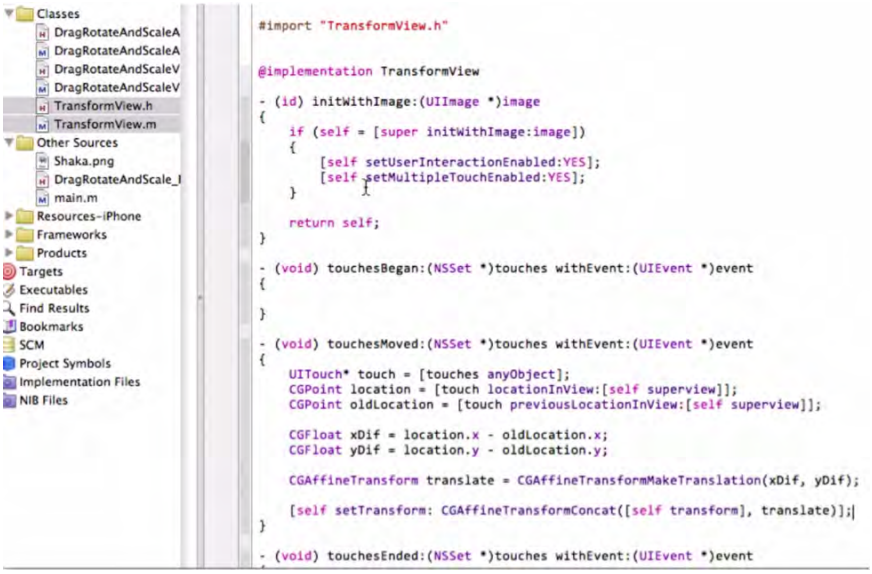


Figure 7–8. *With this added code, the user can now move the TransformView image with one finger.*

```
- (void) touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
                CGPoint newTouch = [[touches anyObject] locationInView:[self~CCC
 superview]];
```

```
                CGPoint lastTouch = [[touches anyObject]
previousLocationInView:[self~CCC
 superview]];

                float xDif = newTouch.x - lastTouch.x;
                float yDif = newTouch.y - lastTouch.y;

                CGAffineTransform translate = CGAffineTransformMakeTranslation(xDif,~CCC
 yDif);
                [self setTransform: CGAffineTransformConcat([self transform],~CCC
 translate)];
}
```

Let's step through this code and see if we can determine what's really happening.

First of all, what is this `CGPoint` business? `CGPoint` is core code that has been programmed for us by the clever people at Apple; it is the part of the program in the *CoreGraphics* module that describes a point in 2D space, with members ***x*** and ***y***. Finger touches by the user result in *CGPoints* when their location is appropriate and requested—by the nature of the application.

This information can yield the distance between the two points using some very simple math. Now, take a look at the call to `[[touches anyObject] locationInView:[self superview]]`. This code grabs a touch object from the NSSet of touches and requests its location in this object's superview. In other words, we are simply asking for the location of the touch in relation to the superview.

This is different than one might expect. Why are we asking about the position in the superview and *not* about the position in the TransformView itself? Because we want to know where to move the TransformView in the superview. Thus, we get the current and previous positions of the touch in the superview.

> **NOTE:** The IPad and iPhone keep track of which view is currently being shown by treating it as a window instance. The instances are arranged in a pyramid order with the top-level view instance called the *content view*, which is the root of all the other views, called *subviews*. The parent of any view to which one is attending, at a given point of time, is referred to as its *superview*.

The next two lines work together to calculate the difference between the old position and the new position on the x- and y-axes.

The following line creates a translation out of the position difference, storing this translation as a temporary *CGAffineTransform*. That's a big term, but it simply means a matrix that stores the change in position for the view. Since the "translation" matrix is relative, we have to add it to our current transform. We do this on the last line of highlighted code, concatenating (merging or combining) the view's "current" transform and the "translation" transform to get a transform that holds our *new* position. Once we have that new transform, we set the view's transform to the new one.

Making Use of *TransformView*

So, our *TransformView* is ready for its first test run. We need to make an instance somewhere and add it as a subview to something else. Our *DragRotateAndScaleViewController* is where this is done. Moving into the *DragRotateAndScaleViewController* header file, all we need to do here is import our `TransformView.h` file, as you can see in Figure 7–9.

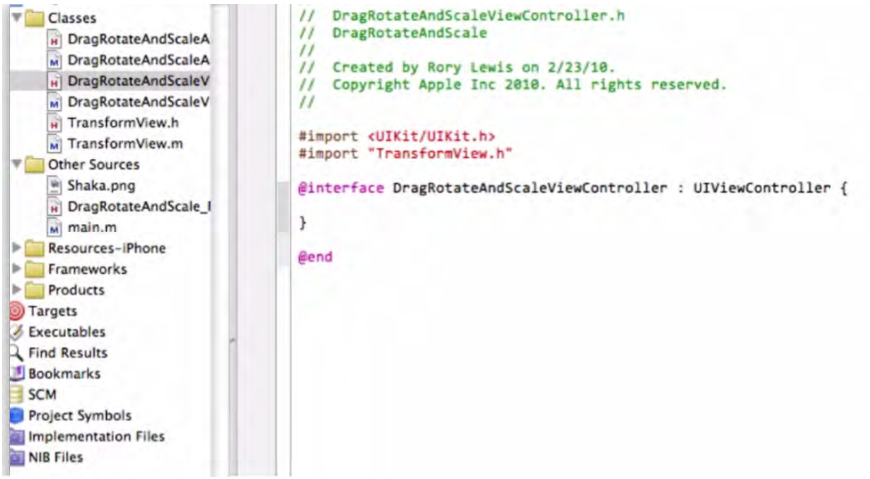


Figure 7–9. *Import the TransformView.h header file.*

```
//
//  DragRotateAndScaleViewController.h

#import <UIKit/UIKit.h>
#import "TransformView.h"

@interface DragRotateAndScaleViewController : UIViewController {

}

@end
```

Creating a TransformView

In the *DragRotateAndScaleViewController* implementation file, we want to create a `TransformView` and make it visible to the user. We want to make sure the view is ready, so we perform this action in the `viewDidLoad` method override. Your code should come out looking like Figure 7–10.

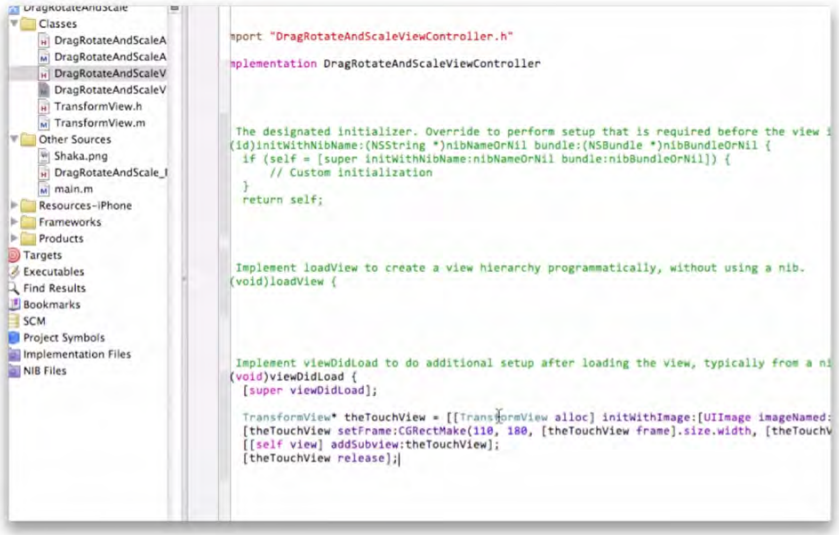


Figure 7–10. *Now we can see and move our TransformView. We're almost there!*

```
- (void)viewDidLoad
{
    [super viewDidLoad];

        TransformView* theTouchView = [[TransformView alloc] initWithImage:[UIImage~CCC
 imageNamed:@"Shaka.png"]];
        [theTouchView setFrame:CGRectMake(110, 180, [theTouchView frame].size.width,~CCC
 [theTouchView frame].size.height)];
        [[self view] addSubview:theTouchView];
        [theTouchView release];
}
```

Let's see how close you are as you analyze these lines and predict what we are creating and defining. In the first bolded line, we begin by creating a new TransformView object, passing it a UIImage object with the name of the image we dragged in at the beginning of the process. This will call the method override we wrote earlier, allowing the TransformView to take touch input.

Next, we set the *frame* of the TransformView in order to position it initially within the view. The numbers were derived from the dimensions of the iPhone and the dimensions of the image, 100 × 100 pixels.

The third line of code adds the TransformView as a subview to the *self view*, so that our TransformView will be drawn.

The last line sends a *release* call—for memory management purposes.

That ought to do it! We should be able to run the code at this point and be able to move the TransformView around—just by touching and dragging. Figure 7–11 shows the first step of testing the TransformView, which is to select the appropriate platform on which to simulate the app.

Figure 7–11. *Make sure the executable is set to the iPhone Simulator.*

Figure 7–12 demonstrates the iPhone Simulator loaded with your single image. I need to be able to move the image – so, touch and drag your photo. Yes, this is a simple test, but Rome wasn't built in a day.

Figure 7–12. *Touch and drag to move your image around!*

Having controlled the image with the simulated fingertip, let's switch over to the other platform. Figure 7–13 shows the frame in which to switch the executable to select the iPad Simulator. You need the TransformView to function there as well as it does on the iPhone.

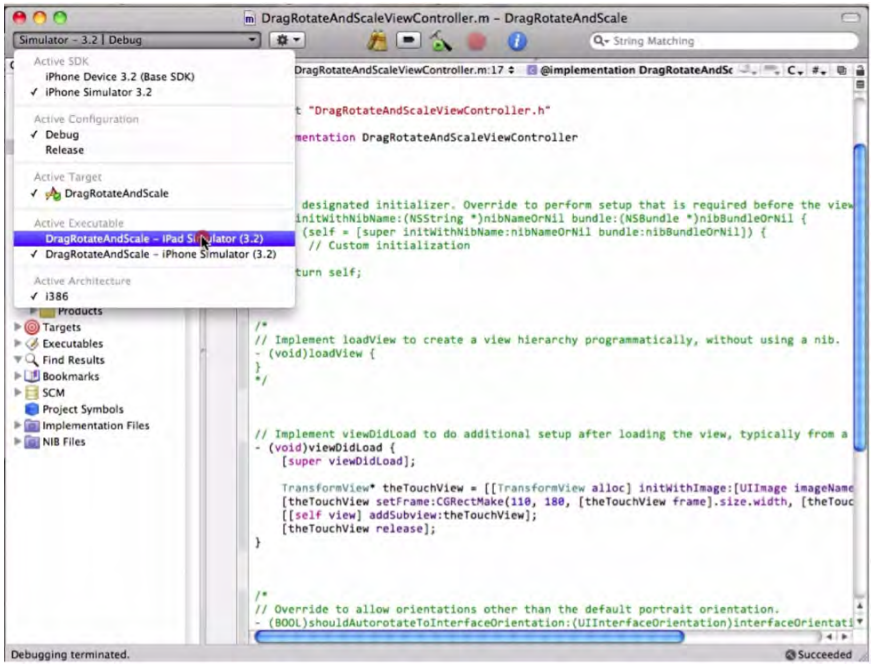


Figure 7–13. *Let's test our code on the iPad Simulator.*

Figure 7–14 depicts the iPad Simulator in a normal view mode, which expands the relative size of the image so that it continues to occupy a proportional amount of the screen. Touch it and drag it to prove the fluency of the iPad platform.

And, because there is also an embedded iPhone view within the iPad Simulator executable, we need to test that as well. Figure 7–15 shows that option, vividly demonstrating the relative sizes of the identical image in these two modes.

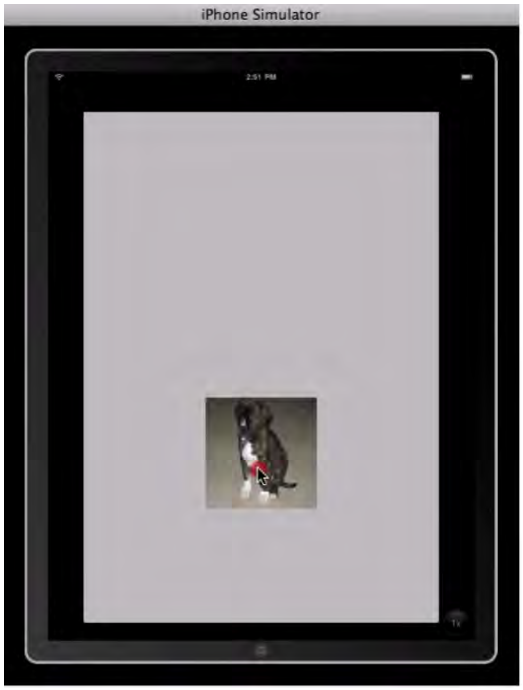


Figure 7–14. *You will see your image moving in a normal iPad view.*

Figure 7–15. *In the embedded iPhone view, we see the image appear to shrink back to its normal size.*

Preparing *TransformView* for Rotation and Scaling

Excellent—we're on a roll! Let's keep going.

Your application looks pretty cool so far, but it can be made even better by allowing the user to zoom and rotate the picture. This requires more complex computation and touch monitoring. As you already know, we must track two concurrent touches and determine their relative positions. To accomplish this, we need to modify our TransformView header file.

In the `TransformView.h` file, we are going to add two *UITouch** fields: `firstTouch` and `secondTouch`. These touch objects will track the distance and angle between the touch points that fall on this view. Additionally, we will add method prototypes for the helper methods that we will be using to calculate the transform changes, as you can see in Figure 7–16.

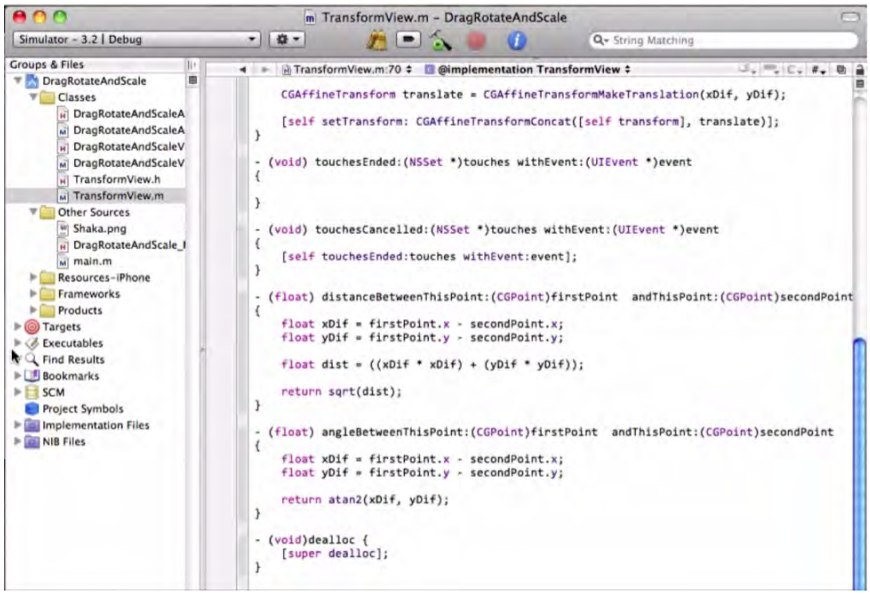


Figure 7–16. *Modify the TransformView.m implementation file to track two touches.*

```
//
//  TransformView.h

#import <UIKit/UIKit.h>

@interface TransformView : UIImageView
{
      UITouch* firstTouch;
      UITouch* secondTouch;
}

- (float) angleBetweenThisPoint:(CGPoint)firstPoint ~CCC
andThisPoint:(CGPoint)secondPoint;
```

```
- (float) distanceBetweenThisPoint:(CGPoint)firstPoint  andThisPoint:~CCC
(CGPoint)secondPoint;

@end
```

We declare the instance variables for the class: two *UITouch* objects that will be used to track the user's touch inputs. At the bottom, you will see the prototypes for the helper methods we will use to change the transform.

You might be asking, "Why even bother with passing CGPoints to the helper functions? Why not use *UITouch** instead?" Because we may decide at some later date to change the way touch input is handled, potentially off-setting or changing the touch positions from their real positions (for whatever reason). That would require us to change our helper code, which is not ideal. Instead, the helper code should always function in the same manner while the calling code changes the input if needed.

Helper Methods

Now that we have our instance variables and method prototypes, we can build out the implementation file. These helper methods could have been declared in a private interface, but that would have been overkill for this type of application.

Inside the `TransformView.m` file, create the helper methods as shown in Figure 7–17.

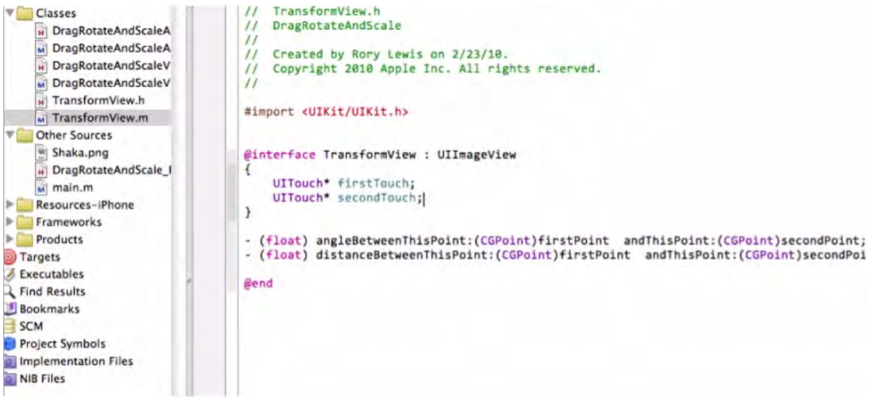


Figure 7–17. *Bring in your "helper methods" from the downloaded helperFunctions.rtf file.*

```
- (float) distanceBetweenThisPoint:(CGPoint)firstPoint andThisPoint:↩
(CGPoint)secondPoint
{
      float xDif = firstPoint.x - secondPoint.x;
      float yDif = firstPoint.y - secondPoint.y;

      float dist = ((xDif * xDif) + (yDif * yDif));

      return sqrt(dist);
}

- (float) angleBetweenThisPoint:(CGPoint)firstPoint↩ andThisPoint:(CGPoint)secondPoint
```

```
{
      float xDif = firstPoint.x - secondPoint.x;
      float yDif = firstPoint.y - secondPoint.y;

      return atan2(xDif, yDif);
}
```

Fortunately, these "helper methods" are relatively straightforward. They simply calculate the distance and the angle in radians, respectively, between the two touches -. These will be used by our *touches* methods to create scaling and rotation transforms that will be applied to the TransformView.

The `-distanceBetweenThisPoint:[*]andThisPoint:` term finds the ***x*** and ***y*** position differences between the two points and utilizes the good old Pythagorean Theorem to calculate the straightline distance between the points.

Similarly, `-angleBetweenThisPoint:[*]andThisPoint:` finds the angle from the first point to the second point in relation to the ***x***-axis, returning the result in radians.

Adding to "-touchesBegan"

We'll start with the easier part of this next section of our code. You will need to add some code to `touchesBegan:withEvent:` in order to track our touches. This looks pretty daunting, but you're beginning to operate at a pretty advanced level … and, for some, this'll be a cakewalk.

We are still in the implementation file, `TransformView.m`, and we are going to rewrite our *touches* methods to handle multiple touches and to utilize our helper functions in order to change the transform.

```
- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
      //Single touch
      if ([touches count] == 1)
      {
        if (!firstTouch)
        {
               firstTouch = [[touches anyObject] retain];
        }
        else if (!secondTouch)
        {
              secondTouch = [[touches anyObject] retain];
        }
      }
      //Multiple touch
      if ([touches count] == 2)
      {
       NSArray* theTouches = [touches allObjects];
       [firstTouch release];
       [secondTouch release];
       firstTouch = nil;
       secondTouch = nil;
```

```
        firstTouch = [[theTouches objectAtIndex:0] retain];
        secondTouch = [[theTouches objectAtIndex:1] retain];
      }
}
```

All right … see if you can tell what's happening here. First, we check to see if there is only one touch—by checking the count of touches. If there is only one touch, we direct the computer to hold onto it—in memory—for later use, in whichever field we aren't already using.

If there are two touches, we grab all of the touches from the set, release our previous touches, and set the fields to the first two touches in the array of touches. This approach gives us the two touch objects we need and it ignores any extraneous touches. Easy, huh?

Copy this code from the TranslateRotateScale.rtf file that you downloaded at the beginning of the chapter and saved to your desktop, as shown in Figure 7–18. Then, paste these lines into the implementation file.

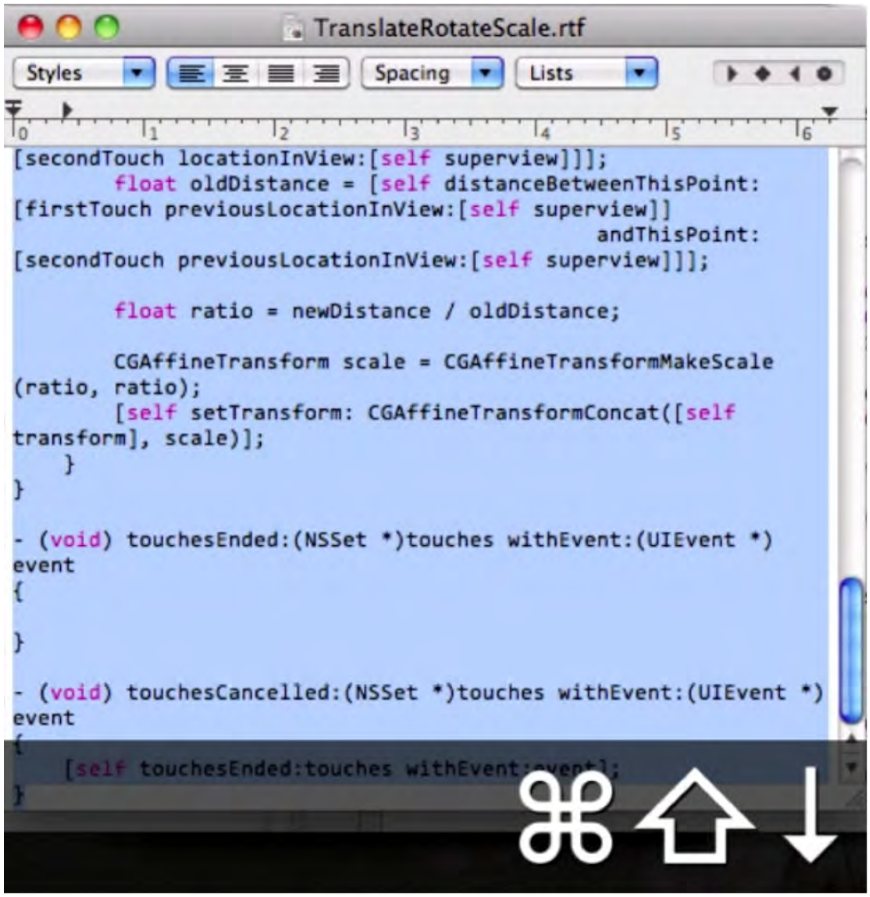


Figure 7–18. *Grab the new touch code from the downloaded TranslateRotateScale.rtf file.*

Modifying *-touchesMoved*

Take a deep breath. *Relax!* The next chunk of code may look intimidating, but it's actually quite simple. We'll walk through it together, and you'll see exactly what's happening.

This code modifies the `-touchesMoved:[*]withEvent:` method, and it also utilizes our helper methods. We use the data from the touches and helper methods to create transforms that are concatenated with the current transform.

```
- (void) touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
      if ([touches count] == 1)
      {
       CGPoint newTouch = [[touches anyObject] locationInView:[self superview]];
       CGPoint lastTouch = [[touches anyObject] previousLocationInView:
       [self superview]];

       float xDif = newTouch.x - lastTouch.x;
       float yDif = newTouch.y - lastTouch.y;

       CGAffineTransform translate = CGAffineTransformMakeTranslation(xDif, yDif);
       [self setTransform: CGAffineTransformConcat([self transform], translate)];
       }
       else if ([touches count] == 2 && firstTouch && secondTouch)
       {
       //Rotate
       float newAngle = [self angleBetweenThisPoint:[firstTouch locationInView:
       [self superview]]

       andThisPoint:[secondTouch locationInView:[self superview]]];
       float oldAngle = [self angleBetweenThisPoint:
       [firstTouch previousLocationInView:[self superview]]

       andThisPoint:[secondTouch previousLocationInView:[self superview]]];

       CGAffineTransform rotation = CGAffineTransformMakeRotation(oldAngle - newAngle);

       [self setTransform: CGAffineTransformConcat([self transform], rotation)];

       //Scale
       float newDistance = [self distanceBetweenThisPoint:
       [firstTouch locationInView:[self superview]]

       andThisPoint:[secondTouch locationInView:[self superview]]];
       float oldDistance = [self distanceBetweenThisPoint:
       [firstTouch previousLocationInView:[self superview]]
       andThisPoint:[secondTouch previousLocationInView:[self superview]]];

       float ratio = newDistance / oldDistance;

       CGAffineTransform scale = CGAffineTransformMakeScale(ratio, ratio);
       [self setTransform: CGAffineTransformConcat([self transform], scale)];
       }
}
```

First, we direct the computer to check whether only one touch moved. If that is the case, then we call forth exactly the same code we had for translation before. It doesn't get much easier than that!

If there are two touches, however, we begin calculating the rotation and scaling transforms. The rotation calculation is started by using our helper method to find the angle between the current touch points. We follow that by finding the angle between the old touch points. A rotation transform is made by finding the difference between the two angles. This creates a relative rotation transform that, just like before, needs to be concatenated with the current transform.

Next, we calculate the appropriate scaling that needs to be done based on the position of the touches. We use our helper method to find the distance between the touch points for both the current touches and the previous touches. We then find the ratio of the new distance to the old distance. This gives us a scaling factor by which we will scale the current transform. Using this factor, we create a relative scale transform and subsequently concatenate it with the current transform.

See? I told you it wasn't that bad! You can show this code to your friends and watch their jaws punch a hole in the floor while you flash them your knowing smile.

Running this code now allows the user to drag the image with one finger or rotate and scale it with a pinch gesture, as shown in Figure 7–19 and Figure 7–20.

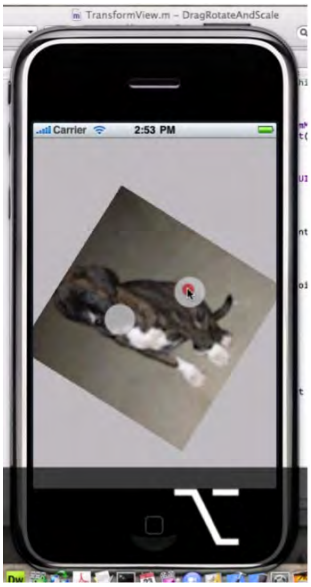


Figure 7–19. *Drag, rotate, and scale your image.*

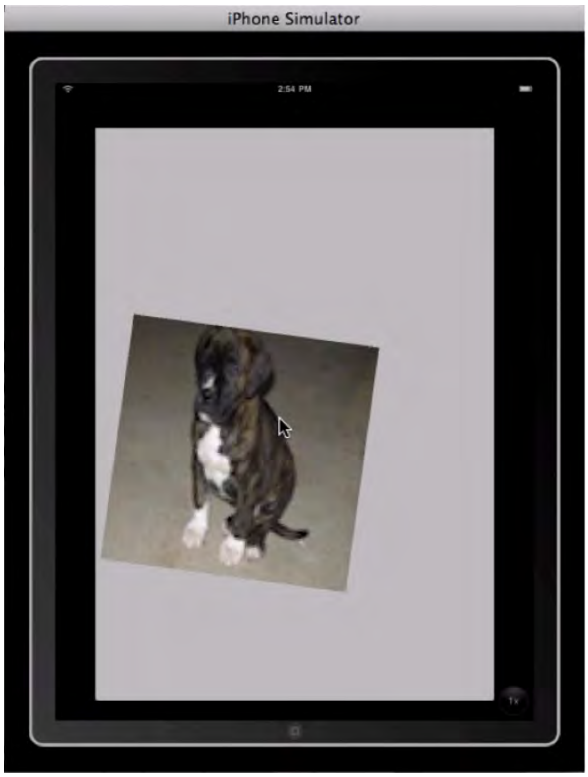


Figure 7–20. *It works smoothly on the iPad!*

You're done! Check out your draggable, rotatable, and scalable custom UIImageView subclass!

So, how is this approach useful? Where should it be used and what are some things to watch for? As we discussed earlier, this is a very intuitive and friendly interface, even for users who are new to the iPhone/iPad. This approach should be considered for any application that has objects that sport lots of detail and benefit from direct user interaction.

NOTE: With the current code, a user can actually shrink an image so small that it cannot be pinched and spread in order to return it to a useful size. Some limit on a scaling factor should be considered for this reason. It is also possible for the user to move an image completely off the screen, so positional constraints are recommended.

Digging the Code

We will now focus on one of the concepts I mentioned earlier only in passing: *event-handling*. The four lines of code that follow deal with related events and the methods by which we want the computer to deal with these events.

```
- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
- (void) touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event
- (void) touchesEnded:(NSSet*)touches withEvent:(UIEvent*)event
- (void) touchesCancelled:(NSSet*)touches withEvent:(UIEvent*)event
```

In order to dig this code, we first need to remind ourselves what methods and arguments are. These four touch methods are called event-handling methods.

Explaining an event-handling method is a bit tricky because it's a very abstract tool – like using a time warp in order to advise Thomas Jefferson about drafting *The Declaration of Independence* on a computer. So before we talk about these four event-handling methods, let's take a close look at how a generic event-handling method works.

Suppose you hear your phone ring. You know that somebody from the outside world is calling you. Do you:

A) pick up the phone and say, "Hello?"

B) pick up the phone and say "Sorry—I need to call you back."

or …

C) let the phone ring through to voicemail?

These are event-handling methods. You have different methods of handling a phone call, and choosing one depends on several factors: who is calling, what you are doing, how tired you are, how hungry you are, and so forth.

Bearing this analogy in mind, the lines we're examining are event-handling methods for handling a touch from the user who is operating your iPhone or iPad app. The programmers at Apple have created event-handling methods that make it easy for you to decide whether you are going to pick up the phone, answer it, and so on.

In our case though, we want to handle four different kinds of events. These touch events are events that are propagated through the responder chain. What's a responder chain? Here's another analogy.

The phone rings, but you don't want to answer it. You ask somebody (brother, sister, mother) to pick up the phone for you. What they do with the phone is totally up to them—because you gave up your chance to answer the phone. That is how the responder chain works: events come in and objects can either handle the events (answer the phone yourself) or continue to pass the event down the responder chain (tell someone else to pick up the phone).

Along these same lines you may see the term f*irst responder* in Interface Builder or some of the Apple documentation. This is the first object in the responder chain, and the first responder always gets the first opportunity with generated events. In the phone

example, you would be the *first responder,* since you were given the opportunity to handle the phone call first. Most controls without targets (such as a button that does not have its target set) send their actions to the first responder by default.

When a touch is placed on the device, the window uses "hit testing" to determine which view the touch was in and then passes the touch event information down the responder chain. If a view's `userInteractionEnabled` property is set to NO, the event will continue down the responder chain until a view is able to handle that event. A touch and its associated view are linked for the lifetime of the touch. This means that, even if the touch moves off the view that received the touch initially, that view is still in charge of that touch and no other views will receive touch information about that touch … no matter where it moves.

The generated events hold information on the touches that triggered them, as well as a timestamp, the event's type, and the event's subtype, all of which can be accessed through corresponding properties.

New to iPhone OS 3.0 are *motion* events that are triggered similarly to touch events. They are targeted events that default to the first responder. The event object for a motion may have its subtype set to UIEventTypeMotionShake, which provides an easy way to detect shake events. The types and subtypes of a UIEvent object provide lots of useful information that can help to determine how an incoming event should be handled.

Cocoa Touch, the view system of the iPhone OS, works according to a hierarchy. That is, views handle drawing themselves and their subviews. When a call like `[[self view] addSubview:aView];` is made, `a view` is made a subview of `self`'s view.

Pretty simple, right? Well, each view has a transform that describes the view's location, rotation, scaling, and other factors relative to that view's superview. This is exactly what we need in order to make our custom view scale, rotate, and move around in its superview, whenever a touch is sensed and identified.

We can change the transform in many ways, but, for the most part, we only need to deal with distances and angles between touches to do everything we need. The structure CGAffineTransform is used to store and manipulate the transforms of views. Now that you're in the world of advanced programming, you should be comfortable using the C-style calls for CGAffineTransform. Take some time to peruse the documentation on CGAffineTransform and take a look at the view programming guides to get an Apple-approved in-depth description of how all this stuff works.

Gesture Support and the iPad

The iPad SDK contains two never-before-seen commands (`3Tap.plist` and `LongPress.plist`) that are nowhere to be found in any iPhone SDK up to 3.1. What do `3Tap` and `LongPress` do? Exactly what their names say. The iPad will recognize three quick taps and an extended long press—different from the one already used for copy and paste.

My prediction is that a primary use of the iPad in future years will be as a textbook medium; students will be able to write notes on their ebooks and display them on their

iPads. Bear in mind that we already allow a swipe to delete messages and emails, just like pressing the Delete key. When a student is writing notes on an iPad etext, maybe a swipe will be a neat way to erase some of the contents, just like using a pencil eraser.

Chapter 8

Table Views, Navigation, and Arrays

To teach or not to teach … arrays, that was the question. In my original plan for this book, I had decided to *not* teach arrays, for a couple of reasons:

1. They are difficult for computer science majors—let alone absolute beginners.
2. They are considered boring.
3. They are even more complex placed in the context of Objective-C.

But I *needed* to teach arrays! Students in my iPhone/iPad class wanted to write Final Projects that utilized tables – which required the use of arrays – and I realized just how essential they were.

To help you decide whether to read this chapter, I want you to consider three things.

1. Why I was *not* going to teach this chapter.
2. Why you *may* want to skip this chapter.
3. Why I decided to teach it after all … and to teach it *my way!*

Why I was *not* going to teach this chapter:

I have had many conversations with colleagues about what takes place when computer science students first come into contact with arrays. My experience is that understanding arrays has nothing to do with being smart. Often I see my brightest and most diligent students struggle and stumble as they plunge into the jungle of arrays. Conversely, I see students who have no business walking the halls of the Computer and Engineering Building grasp the concept as if it were their native language. One student who struggled with the “Hello World” chapters actually nailed arrays!

Why you *may* want to skip this chapter:

One thing I'm clear about is this: you *don't* have to know arrays to be a successful programmer. At a recent conference on programming the iPhone and the iPad, I attended a presentation by a very cool team of programmers about their new apps. One guy was talking about this goofy game in which users throw things into a garbage can.

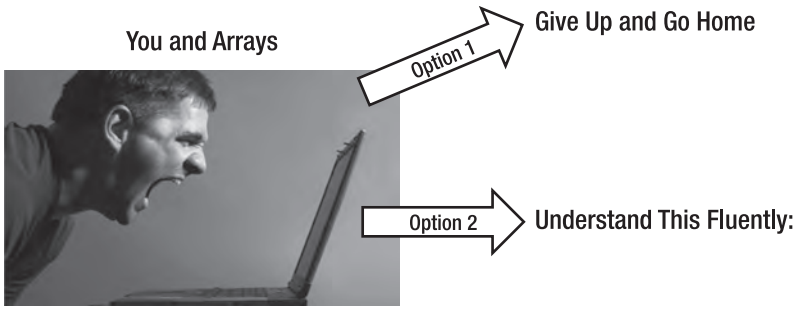
The company consists of two guys in their mid-50s. They have 11 games and make over $20,000 per month. Get this—two years ago they were interior designers who had never programmed before. After they gave their talk, they opened the floor to questions. When asked about the mechanics of how they processed their arrays, they gave this response to a full room of several hundred high-tech geeks:

> *"We don't know anything about arrays. We just use stubs and boilerplate ins and outs—and then we pray!"*

Why I decided to teach it after all … and to teach it *my way!*

In the early stages of this book, I stayed on the fence regarding this topic. I knew that at some point, though, I would have to decide, and my publishers would want to know as well. *To array or not to array!*

From the students' point of view, this fork in the road was tough as well, but because of the requirements of academia, it looked to most of them something like Figure 8–1. For most of my students, Option 1—give up and drop out of engineering—was really no option at all. And if they landed in my class, my objective was, of course, to lead them to Option 2—become committed enough to learn the material.




```
- (NSInteger)tableView:(UITableView*)tableView didSelectRowAtIndexPath:(NSIndexPath*)indexPath
{
  NSString *text = [NSDtring stringWithFormat:@"%@%@",[names objectAtIndex:[indexPath row]],@".png"];
  FooVar1Controller* retController = [FooVar1Controller fooVar1ControllerWithImageNamed:text];
  [[self navigationController] pushViewController:retController animated:YES};
}
```

Figure 8–1. *The traditional options for dealing with arrays: 1) Give up, drop out of engineering, and go home, or 2) get committed and do whatever it takes to learn this torturous material!*

Figure 8–2 illustrates my take on the "Arrays" conundrum. You can see that it consists of *three* options—the added one being a blend of the classical dichotomy:

1. **Avoidance:** "Arrays—who needs 'em! *I'm outta here!* Chapter 9, "MapKit," here I come!"

2. **Lewis's Array Method:** I will present specific boilerplate code, by which you'll learn where and how to insert functioning arrays into your code. In this pragmatic approach, you'll get a basic introduction to what arrays are and learn a few helpful tricks. Because I will have simplified the technical aspects in a major way, you'll end up with *only a taste* about how arrays function. Still, your code will work, and you'll feel really clever!

3. **Learn Arrays Completely:** There's a good reason why Dave Mark and Jeff LaMarche waited until their advanced book, *More iPhone 3 Development : Tackling iPhone SDK 3* (Apress, 2010), to teach arrays. They referred to this area of programming as " … the Devil …"

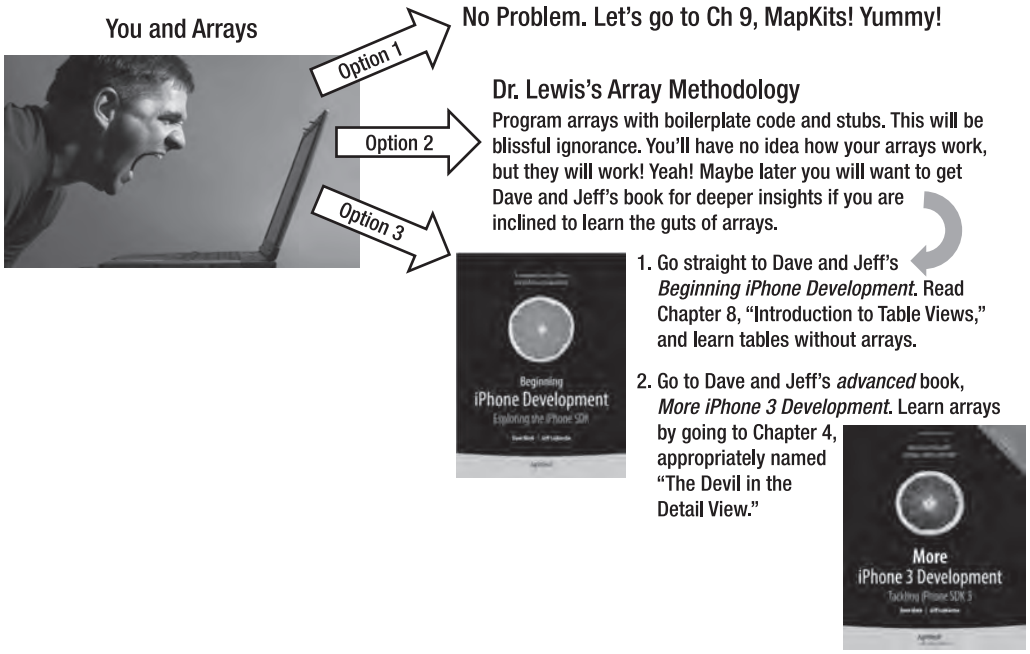


Figure 8–2. *Lewis's take on the issue of arrays: 1) Avoid the issue altogether, 2) Trust in Dr. Lewis's pragmatic approach and learn a few helpful tricks here and there, or 3) Go to Chapter 4 of Dave and Jeff's advanced book and shake hands with the Devil!*

How Shall We Proceed?

If you're still here—and haven't skipped ahead to Chapter 9, or put the book away entirely—then I am assuming you are willing to go along with the program for this next part of our journey. It is important that I communicate two caveats:

- You will, at times, think that I'm *not giving you enough* details about arrays.
- You will, at times, think that I'm giving you *too many* details about arrays.

In other words, please show the same awesome flexibility and patience that you have demonstrated up to this point in our give and take partnership. Trust me! *Fair enough?*

OK—then let's get down to business.

Table Views and Navigation Stacks

To understand how arrays are used in iPhone/iPad apps you need to understand the role of table views and navigation stacks, very powerful and helpful pieces of the iPhone OS. So far, we have talked only a little about tables. These are commonly used to show lists of items, and they allow the user to select and organize those items.

When you need to list things to buy from the grocery store, for instance, what steps do you take to prepare? First, you find a piece of paper, and then you list all of the things you need. A table view acts just like that piece of paper: it organizes the items in a list so that you can easily find anything you need.

Using table views to display lists is only part of the picture, though. A navigation stack lets us move between table views and even between "normal" views.

Let's keep our subject simple for as long as we can. Right now then, I want you to keep in mind five things about table views:

1. A table view is nothing more than a list of stuff, a list of data.
2. In the iPhone/iPad, a table view contains the code for a view object—the thing that displays your table's data on the iPhone's/iPad's screen.
3. The UITableViewCell controls every row in a table view. Don't question this; just accept it.
4. A table view *does not store* your table's data. It stores only *code*—to display the rows that are currently visible on the user's iPad/iPhone screen.
5. Table views involve at least two chunks of data:
    a. Information about which types of data are present and how these are to be configured—by the *UITableViewDelegate* object;

b. Information about how specific data is arranged and displayed—by the *UITableViewDataSource* object protocol.

Similarly, arrays are just lists of stuff—and happens that table views are great at displaying and organizing arrays. But I first need to run by you a thought that vaguely describes how an array works:

> *An array is an ordered collection of objects, starting at zero, which can store any number of objects.*

Imagine a vending machine filled with a wide variety of goodies, numbered A1 to H6 from top left to bottom right. You see a snack that looks too tempting to pass up, so you enter that snack's number and receive your delicious treat. The number used to access your candy, identifying its position in the sequence, is called an *index*. We use indexes (or "indices") to find objects in an array in the same way we would get candy from a vending machine. One important distinction is that an object *remains* in an array—even after you find it using its index. The candy, fortunately, does *not* stay in the vending machine!

Let's imagine that it is your job to report the kinds of candies available in the vending machine. But you can't just look in and count. Instead, the glass has been replaced with a big sheet of opaque plastic, and all the numbers have been removed.

You are therefore playing the role of the *table view*. First, you need to know how many goodies are in the machine. Fortunately, the vending machine already has a read-out that tells you this, and that read out is what will appear on the screen of your iPad or iPhone.

Food: Following the App Store Model

For this example, we are going to use arrays, views, and tables to make a very simple app. It will be built upon the master-detail paradigm and use table views to display delicious dinner entrees! Our approach will be very similar to the way that the App Store presents choices on the iPhone/iPad. We will seek to implement a pattern in which the user starts in a table view that presents categories, moves on to a list of all of the items in a selected category, and finally navigates to a page with detailed information about a specific choice.

We will use our boilerplate code that can be used to form arrays, and we will use stubs in other applications that produce results similar to the App Store interface just described.

Starting the Food App

As usual, we open up Xcode and begin a new project using our shortcut, or by selecting the option from the File drop-down menu, as demonstrated in Figure 8–3. Choose the Navigation-based Application template, as shown in Figure 8–4.

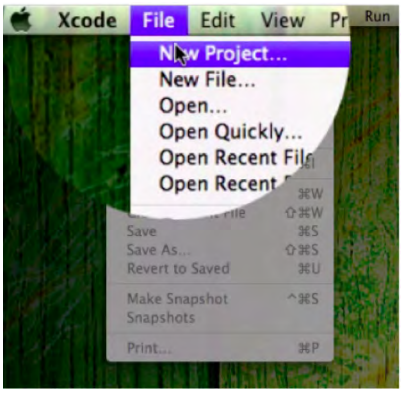


Figure 8–3. *Start a new project in Xcode. You are right – I didn't use the usual shortcut … I just wanted to see if you were awake!*

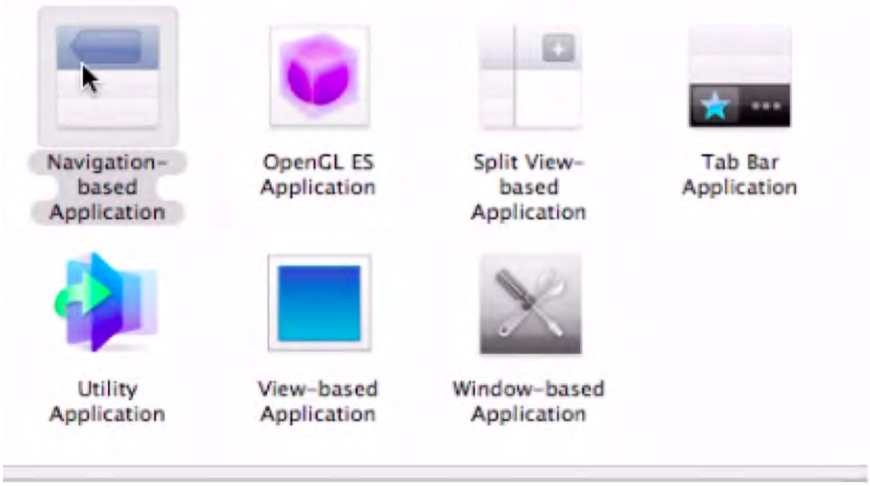


Figure 8–4. *Select a Navigation-based Application.*

Make sure the images we're going to use are in the Xcode project, as in Figure 8–5.

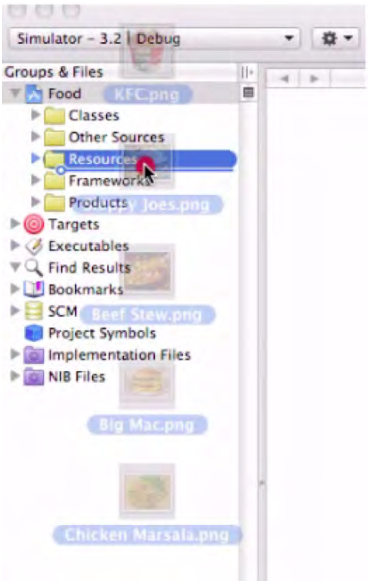


Figure 8–5. *Copy the images into the Xcode project. Don't forget the icon file. Note that these images look like they're being dropped into the Resources folder, but they're actually en route to the Other Sources folder.*

The images can be found on the book's download page at `www.apress.com` website, along with all of the source code for this project. Drag and drop the images into the `Other Sources` folder, and make sure to check the Copy Into box. When your screen looks like Figure 8–6, you're ready to go.

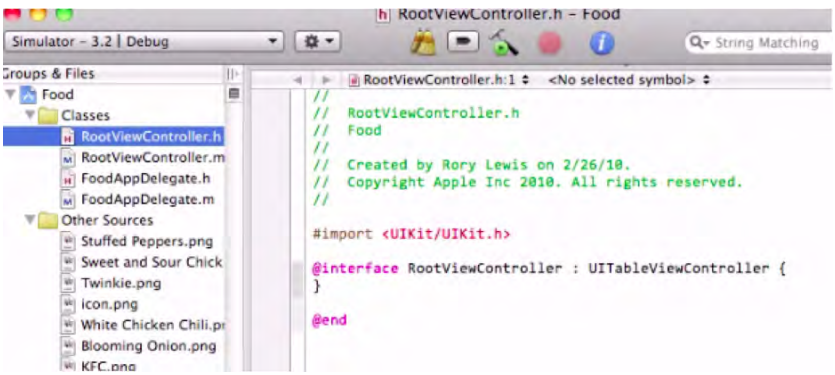


Figure 8–6. *Our project is ready to go.*

Adding the Category Names Array in RootViewController.h

At this point we're still just setting up your table—in the same way that you would set up a list or an Excel chart. We have not yet begun the boilerplate stuff.

In order to populate the list of category names, we need to be able to store them inside the `RootViewController`. So, we need to move into the `RootViewController.h` file and

set up a field to hold the categories. We also need to make sure our `RootViewController` can drill down on the information the user wants by creating a new View Controller. To do this, we add an import line and a new field in the header, as shown in Figure 8–7.

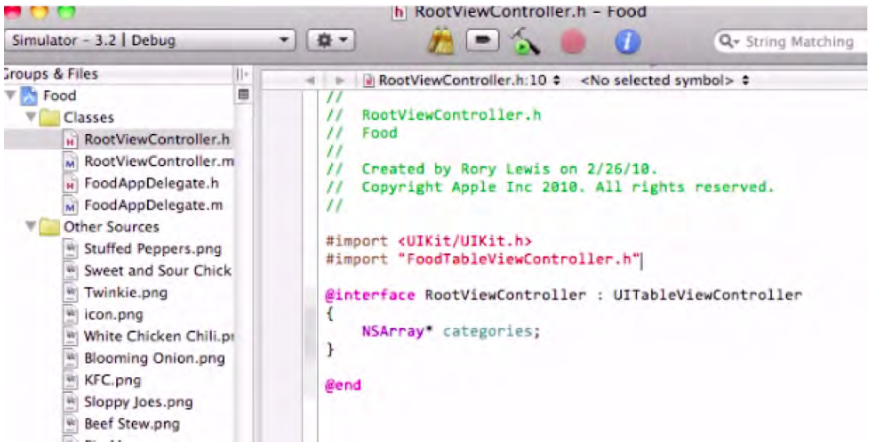


Figure 8–7. *Import the FoodTableViewController.h file and create the categories array.*

Add the line `#import "FoodTableViewController.h"` at the top of the header file. This instruction will import a class that we will create in a later step, but we're putting it in here now since we're already where we need to be.

Next, add a field to the class with the line `NSArray* categories`. This array will hold the category names for use in the table view:

```
//
//  RootViewController.h

#import <UIKit/UIKit.h>
#import "FoodTableViewController.h"

@interface RootViewController : UITableViewController
{
        NSArray* categories;
}

@end
```

Creating the Categories Array in -viewDidLoad

Moving into the `RootViewController` implementation file, we need to set up how the category names are going to be displayed. First, we need to create our category names and hold onto them for later. We will store our array of names in the Categories field that we previously created.

In `RootViewController.m`, we override the `-viewDidLoad` method to set up what we need. First, as always, we call `[super viewDidLoad]` to let the superclass respond to the loaded view as normal. We create the array with all of the names we want and set

"categories" equal to that new array. Note the @ symbols and the *nil* item at the end; those are important!

On the next line, we set the title of the View Controller so that when the navigation controller needs to display a title, "Categories" will be displayed. This title is displayed at the top of the table view in the navigation bar. The code looks like this:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

        categories = [[NSArray alloc] initWithObjects:@"Chicken", @"Beef", @"Pork",↩
 @"Fish", @"Vegetarian", @"Really, Really Healthy Food", nil];
        [self setTitle:@"Categories"];
}
```

Setting Up Table View Data Source Methods

The UITableView class uses delegation and data source objects in order to get data to display and handle input from the user. The methods we are most interested in for the data source, which is already linked to the `RootViewController`, are the following:

```
-(NSInteger)numberOfSectionsInTableView:(UITableView*)tableView

-(NSInteger)tableView:(UITableView*)tableView numberOfRowsInSection:(NSInteger)section

-(UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:↩
(NSIndexPath *)indexPath
```

These are not the only data source methods, but they are the methods we will be using in this exercise. Let's start with `-numberOfSectionsInTableView`. This method returns the number of sections in the *argument* table view. Sections break up the table into chunks, each presenting information based on that section. Make sure you have "`return 1;`" in this method, because that tells the table view that it only has *one* section. This means that everything will be displayed together. Here's the code:

```
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}
```

Next, for `–tableView:numberOfRowsInSection`, I want you to remember something for all your future table view adventures:

> *We return the number of items we want to display.*

In this case we want to display *all* of the items in the `categories` array, and the code for this method is

```
return [categories count]
```

This asks the `categories` array for the number of items it contains and returns *that* number to the table view for the number of items it will display:

```
-(NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [categories count];
}
```

Finally, we need to override `-tableView:cellForRowAtIndexPath:` for our data source methods. This method creates a UITableViewCell, modifies that cell to display the appropriate data, and then returns that cell to the table view for display. We want to change the text of the cell to display the category names we created earlier. To do this, we need to create a cell with a reuse identifier, change it, and then return it.

> **NOTE:** What is a *reuse identifier*? To learn about these little gems, check out the "Digging the Code" section at the end of the chapter.

it regarding the topic of data source methods! Here's the revised code:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier] autorelease];
    }

        NSString* text = [categories objectAtIndex:[indexPath row]];
        [[cell textLabel] setText:text];

    return cell;
}
```

Table View Delegation

Next, we need to set up *delegation* in the `RootViewController` for our table view. Delegation tells the table view what to do when the user taps on an item in the table. We'll use

```
-(void)tableView:(UITableView*)TableViewdidSelectRowAtIndexPath:
(NSIndexPath*)indexPath
```

as our method to accomplish this.

The first line in this method simply creates a pointer to an array, which we initially set to *nil*. We do this so that our later check will fail if the user does not hit a row that we support. A *switch statement* comes afterward; it determines which row the user picked. If it is one of the rows we support, namely rows 0 (the first row), 1 (the second row), …

or 5 (the sixth and last row), we set the array pointer we made before to an array that holds the data we want to display in the next view controller.

We then check to see if the array pointer has been set to something other than *nil*. In this world, "*nil*" will always evaluate as False, while any object will evaluate as True. If the array is valid, we create a new `FoodTableViewController` with the array and push that new controller onto the *navigation stack*. After all of this, we have the table view *deselect* the row that was selected, asking it to animate itself to make the deselection process.

You'll notice that all of the names in the array assignments that we make in this method are identical to the names of the images used in the project. This is no accident. We use the items in the arrays to get those images later.

The code for our delegation method looks like this:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath*)indexPath
{

        NSArray* names = nil;

        switch ([indexPath row])
        {
                case 0:
                        names = [NSArray arrayWithObjects:@"Chicken Marsala",@"White Chicken Chilli", @"Sweet and Sour Chicken", nil];
                        break;
                case 1:
                        names = [NSArray arrayWithObjects:@"Beef Stew",@"Sloppy Joes", @"Stuffed Peppers", nil];
                        break;
                case 5:
                        names = [NSArray arrayWithObjects:@"Big Mac",@"Twinkie", @"KFC", @"Blooming Onion", nil];
                        break;
                default:
                        break;
        }

        if (names)
        {
                FoodTableViewController* ftvc = [FoodTableViewController foodTableViewControllerWithFoodNames:names];
                [[self navigationController] pushViewController:ftvc animated:YES];
        }

        [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Setting up FoodTableViewController

We need to create the next level of detail for our application. The first level was categories of foods, but now we want more detail … in this case, food choices within a category. We do this by creating a new view controller, our `FoodTableViewController`, to

display more specific information (Figure 8–8). When adding the new file, be sure to make `FoodTableViewController` a UITableViewController subclass using the checkbox on the file template Options section, as shown in Figure 8–9.

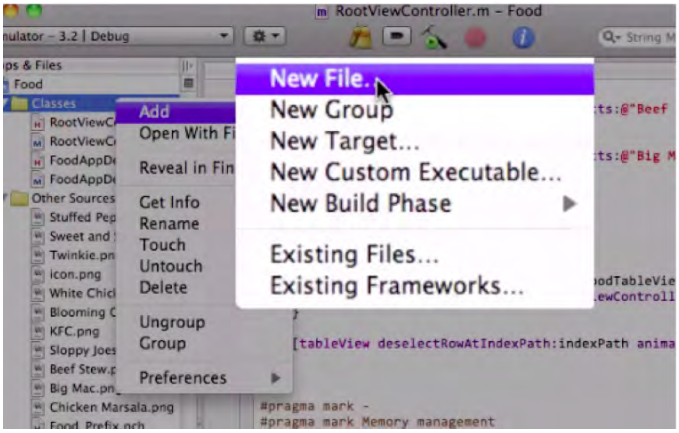


Figure 8–8. *Add a new file to the Food project.*

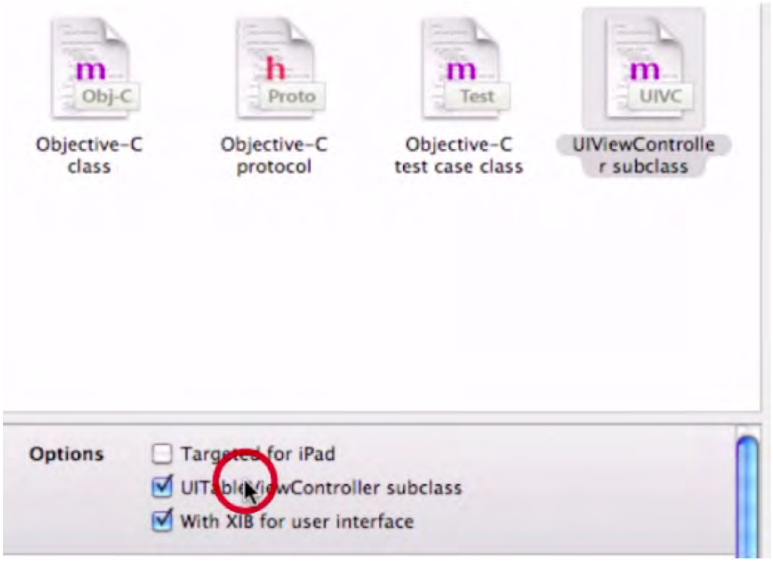


Figure 8–9. *Make the FoodTableViewController a UIViewController subclass by checking the UITableViewController Subclass box.*

Once you have named and saved your new `FoodTableViewController` class, as shown in Figure 8–10, open up the `FoodTableViewController.h` file. Within this file, we're going to add some code that's very similar to the `RootViewController.h` code, but with a few tweaks. First, insert the `#import" FoodTableViewController.h"` line. This will import the `FoodViewController` class once we are underway.

Next, we want to make a new field: `NSArray* names`. This array will hold the food names for this table view controller. This array functions very much the same as the `categories` array from the `RootViewController`. However, we need to make the names array visible to other classes so that the `RootViewController` can pass the data the `FoodTableViewController` needs. To do this, we create a property with the line `@property(nonatomic, retain) NSArray* names`. This allows us to call `setNames:` on an instance of this class. We will also create a "convenience constructor" named `foodTableViewControllerWithFoodNames`.

What in the world is a *convenience constructor*?! Why would we bother making one? Well, a convenience constructor creates an object, usually taking some initialization parameters, and does the work of *memory management* for us. This means that calling the convenience constructor does not require us to release the object when we are done using it. So not only can we create an object with defined starting data, but we also don't have to worry about tracking its lifetime to avoid memory leaks. Additionally, convenience constructors are really easy to make!

> **NOTE:** See "Digging the Code" to learn more about memory management.

You may remember from Chapter 4 that the plus sign, "+," in front of the convenience constructor makes that method a *class method*. If not, just remember that this means we can simply ask the class to create a new instance for us instead of having to `alloc` and `init` the object ourselves.

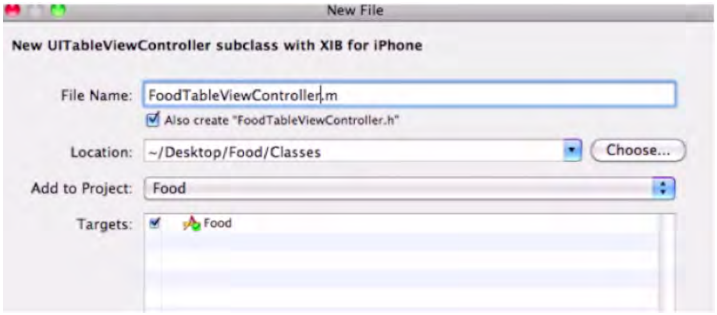


Figure 8–10. *Name and save the new FoodTableViewController class.*

```
//
//  FoodTableViewController.h

#import <UIKit/UIKit.h>
#import "FoodViewController.h"


@interface FoodTableViewController : UITableViewController
{
        NSArray* names;
}

+ (FoodTableViewController*) foodTableViewControllerWithFoodNames:(NSArray*)foodNames;
```

```
@property (nonatomic, retain) NSArray* names;

@end
```

Creating the Convenience Constructor for the FoodTableViewController

Inside `FoodTableViewController.m` we will start by creating the meat of the convenience constructor we declared in the header. But first, we need to add the line `@synthesize names;` so that our property is available to us. Next, we prepare for our convenience constructor by making a C compiler directive (a special way of making something really easy to change and use) that will be called `FoodTableViewControllerNibName`. A few spaces later, we will define this directive to be `@FoodTableViewController`, since that is the name of the `.xib` file we want this view controller to use.

Why are we doing this? We will use this name with our convenience constructor. By creating this directive in the implementation file, we hide how our convenience constructor works and make sure that the right `.xib` file is used to build this view controller. In short, it makes creating a new `FoodTableViewController` easier by not requiring a nib name when called from elsewhere.

After we make our directive, we want to insert our convenience constructor code. You can simply copy and paste the declaration from the header into the implementation file, remove the semicolon at the end, add brackets, and we're ready to start filling it out!

The first thing we do in the convenience constructor is create a new `FoodTableViewController` instance using the `-initWithNibName:bundle` method, passing our *directive name* as the first parameter, and "*nil*" as the second. This keeps the implementation and nib name hidden from prying eyes and makes this method very easy to use.

Before we return the instance, however, we want to set the names of the controller to the names that the user passed to the convenience constructor. We call the property we established in the header to set the names. We then call `return [theController autorelease]` to make sure the caller does not have to worry about memory management for this instance.

That's it. We're *done* with the convenience constructor!

```
/
//  FoodTableViewController.m

#import "FoodTableViewController.h"

#define FoodTableViewControllerNibName
@"FoodTableViewController"

@implementation FoodTableViewController

@synthesize names;
```

```
+ (FoodTableViewController*) foodTableViewControllerWithFoodNames:(NSArray*)foodNames
{
        FoodTableViewController* retController = [[FoodTableViewController alloc]↩
 initWithNibName:FoodTableViewControllerNibName bundle:nil];
        [retController setNames:foodNames];
        return [retController autorelease];
}
```

Data Source and Delegation for the FoodTableViewController

Our table view won't do very much at this point, for we haven't told it what to display. Just as before, we need to set up our data source and delegate methods. And just as before, we want to `return 1;` for `-numberOfSectionsInTableView:` and similarly `return [names count];` for `-tableView:numberOfRowsInSections`. That's all pretty straightforward and functions exactly like the code from `RootViewController`. And as you might expect after all of this, for `-tableView:cellForRowAtIndexPath:` all we have to do is change `categories` to `names` and we are done with data source methods for `FoodTableViewController`:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
    return [names count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:↩
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:↩
CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault↩
 reuseIdentifier:CellIdentifier] autorelease];
    }

        NSString* text = [names objectAtIndex:[indexPath row]];
        [[cell textLabel] setText:text];

    return cell;
}
```

On to delegation! Of course, for delegation we will have to change a lot more to do what we need in `FoodTableViewController`. Inside `-tableView:didSelectRowAtIndexPath:` we need to create the next level of detail and pass it some information. That next level is our `FoodViewController`, which we will create in the next step. For now, though, we will assume it is completed and use a convenience constructor we will create later.

We first create a string that will be used to get the image for the next level of detail. However, we can't just use the name from the `names` array; we have to add the extension before UIImage will load the image for which we are looking.

So, we use the NSString method `+stringWithFormat:` passing `@%@%@`, to return the name from the array, and the extension we want to use, in this case `@".png"`. The `%@` token for the first argument is used to denote that an object goes in that position. Since we are passing NSStrings for both positions, this has the effect of concatenating the strings together, which is precisely what we are after.

WHAT DO THE @%@%@ SYMBOLS MEAN?

The answer to that is out of the scope of this book. If you really want to know the meaning of the %@ symbols, though, consider this. In format strings, `@'%'` character announces a placeholder for a value, with the characters that follow determining the kind of value expected, and how to format it. For example, a format string of `"%d houses"` expects an integer value to be substituted for the format expression `'%d'`. `NSString` supports the format characters defined for the ANSI C function `print(f)`, plus '@' for any object. If the object responds to the `description withLocale: message`, `NSString` sends that message to retrieve the text representation; otherwise, it sends a description message.

Next, we pass the string we made to the convenience constructor for the `FoodViewController` we want to make. Once we have called the convenience constructor we push the new `FoodViewController` onto the navigation stack, and we ask this push to be animated. Finally, as before, we deselect the row and set animated: YES, thereby finishing the delegate methods for `FoodTableViewController`.

```
- (NSInteger)tableView:(UITableView *)tableView didSelectRowAtIndexPath:↩
(NSIndexPath *)indexPath
{
    NSString *text = [NSString stringWithFormat:@"%@%@", [names objectAtIndex:↩
[indexPath row]], @".png"];
    FoodViewController* retController = [FoodTableController↩
 foodViewControllerWithImageNamed:text];
    [[self navigationController] pushViewController:retController animated:YES];
}
```

Creating the *FoodViewController* Class

The `FoodViewController` is still missing. This is the last piece of the application and the class handling the highest level of detail in our navigation-based app. When adding the new file for the `FoodViewController`, be sure to uncheck the "`UITableViewController subclass`" checkbox on the file template chooser, as shown in Figure 8–11. Instead, we are going to use a simple view to show the food when the view controller is loaded.

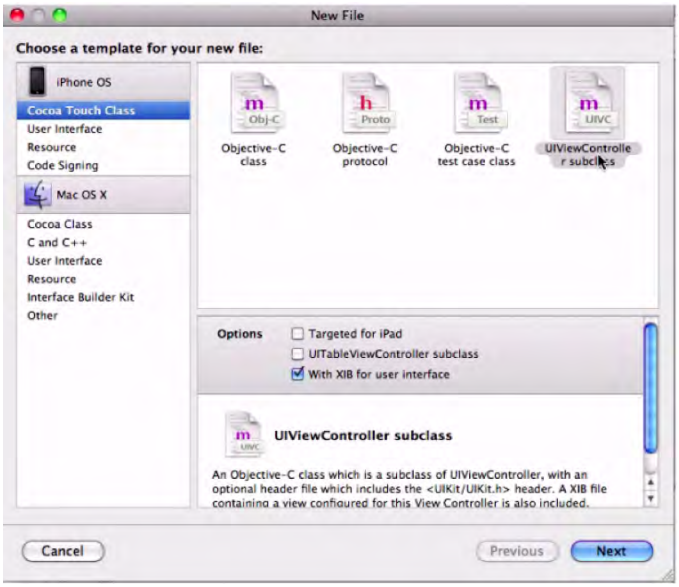


Figure 8–11. *Create the FoodViewController, and make sure the "UITableViewController subclass" box is* ***not*** *checked.*

The *FoodViewController* Header File

Open the `FoodViewController.h` header file. We first want to add a few fields that should be pretty straightforward by now. Add a `UIImageView* field` and an `NSString* field`, and name these fields `imageView` and `imageName`, respectively. The UIImageView will be an `IBOutlet` to which we will link later. This image view will display the image of the food the user has selected. The `NSString` field will hold onto the desired image name until it is needed. We will make properties for both of these fields. Make sure to put "IBOutlet" in front of the UIImageView.

Another task that we must handle is to declare the convenience constructor. This class method takes a single NSString argument, as shown here:

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

@interface FoodViewController : UIViewController
{
        UIImageView* imageView;
        NSString* imageName;
}
+ (FoodViewController*) foodViewControllerWithImageNamed:(NSString*)name;

@property (nonatomic, assign) IBOutlet UIImageView* imageView;
@property (nonatomic, copy) NSString* imageName;
@end
```

The FoodViewController Convenience Constructor

There are only two things we need to do in the `FoodViewController` implementation file: create the convenience constructor, and load an image for when the view loads.

Let's start with the constructor. Just like before, we start by creating a directive with the name of the nib called `FoodViewControllerNibName`,with the appropriate nib name. Next, we need to make sure we have all of our properties synthesized. Copying and pasting the method signature should get us started on our convenience constructor.

The first line in the convenience constructor creates the `FoodViewController` instance that we will return, passing the directive to `-initWithNibName:bundle`. The image name is then set through a "property method call," a sub-routine that helps manage resources efficiently. Finally, the newly created instance is sent an *autorelease* message, and the result from that message is returned.

```
//
//  FoodViewController.m

#import "FoodViewController.h"

#define FoodViewControllerNibName
@"FoodViewController"

@implementation FoodViewController

@synthesize imageView;
@synthesize imageName;

+ (FoodViewController*) foodViewControllerWithImageNamed:(NSString*)name
{
        FoodViewController* retController = [[FoodViewController alloc]↩
 initWithNibName:FoodViewControllerNibName bundle:nil];
        [retController setImageName:name];
        return [retController autorelease];
}
```

Setting Up FoodViewController, -viewDidLoad, and the (.xib)

The last bit of code we need is to override `-viewDidLoad` in `FoodViewController.m`. All we need to do is add a few lines of code. The image for the UIImageView outlet that was created in the header needs a photo to display. So we simply create a UIImage with the class method call of `+imageNamed`. This creates an image with the data from the file with the passed name. Setting this image on the `UIImageView` will make it visible. We also don't need `imageName` anymore, now that the image has been loaded, so we release it.

```
- (void) viewDidLoad
{
        [imageView setImage:[UIImage imageNamed:imageName]];
        [imageName release];
}
```

Finally, open the `FoodViewController.xib` file, as shown in Figure 8–12, for we have some familiar moves to arrange our view.

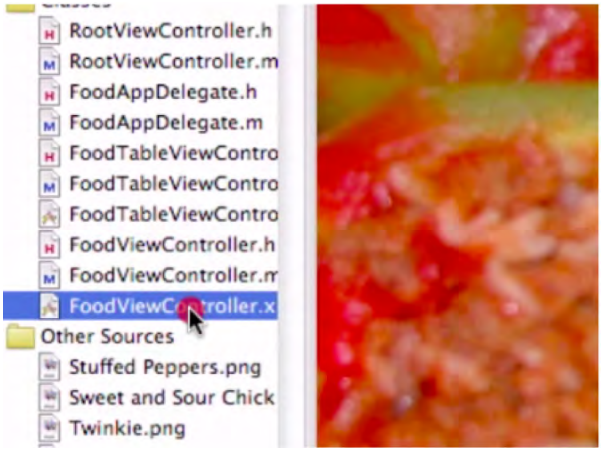


Figure 8–12. *Open the FoodViewController.xib file.*

Drag a UIImageView onto the View, as indicated in Figure 8–13, and resize it to occupy the entire space. Right-click and drag from the File's Owner icon to the new UIImageView, as in Figure 8–14. Then, connect the imageView outlet, as shown in Figure 8–15.

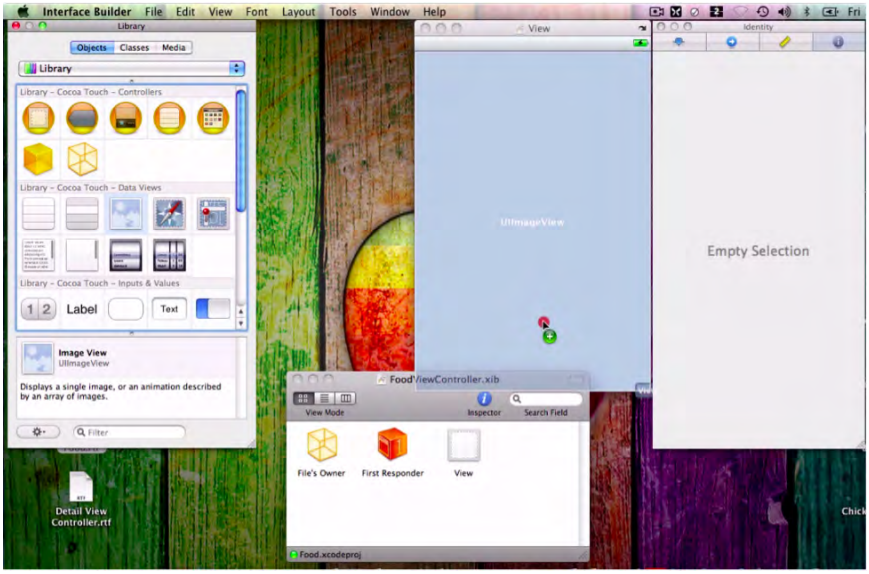


Figure 8–13. *Place a UIImageView in the view, and make sure it is sized correctly.*

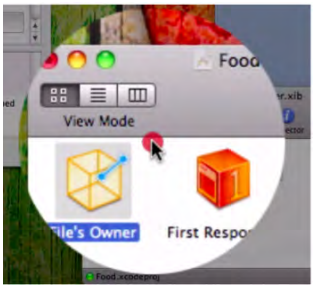


Figure 8–14. *Right-click and drag from the File's Owner icon …*

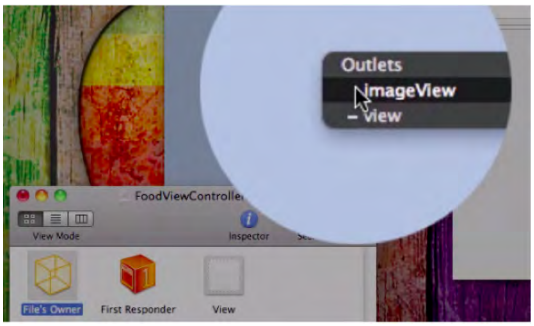


Figure 8–15. *… over to the UIImageView and select the imageView option in the Outlets drop-down menu.*

Icon File

Our food-browsing app is done! But what good is an app with no *icon*? The icon draws the user to run your app, to buy your app. It is the user's first and last impression of your app. With this in mind, let's set an icon for this project.

The icon, a `.png` image of 52 × 52 pixels, has already been added to our arsenal of images in the "Resources" folder. So, open the Food `Info.plist` file and set the Icon File entry to the name you wish to use as an icon. Refer to Figure 8–16.

By the way, the extension, usually `.png`, does not need to be included in the name.

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 8–16. *In the Info.plist, set the icon file's name.*

Testing the App

Congratulations! You are finished with Chapter 8's exercise!

So let's test-drive it. Running the "Food" application allows the user to select categories of dishes, to scan specific dishes in a category, and to view a picture of the dish, in all its mouth-watering glory. Figures 8–17 through 8–20 show the sequence from the menus to viewing a selected entree.

With the power of UIKit, specifically `UINavigationController`, the drill-down navigation interface is provided with virtually no code on our part. The folks at Apple built UIKit with a lot of helpful classes and pieces of code that significantly reduce the amount of work required to produce a powerful and polished application in a short period of time.

Nice going!

Figure 8–17. *A pretty fair menu of choices represented at the Category level.*

Figure 8–18. *Within the Chicken category, we see several entrees.*

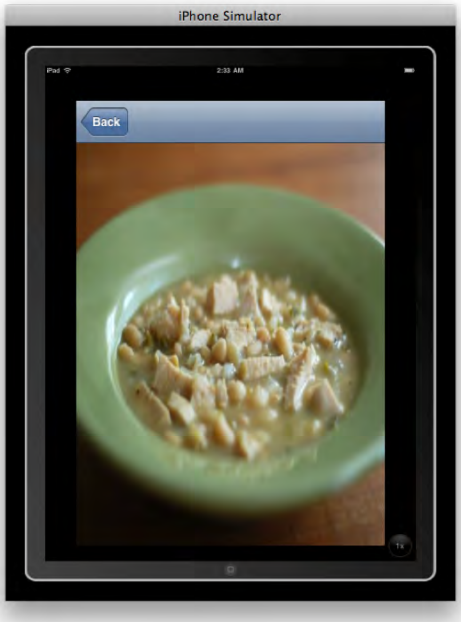


Figure 8–19. *Here is a detailed image of one selection in the iPad Simulator "2x" magnification … Mmm, chicken!*

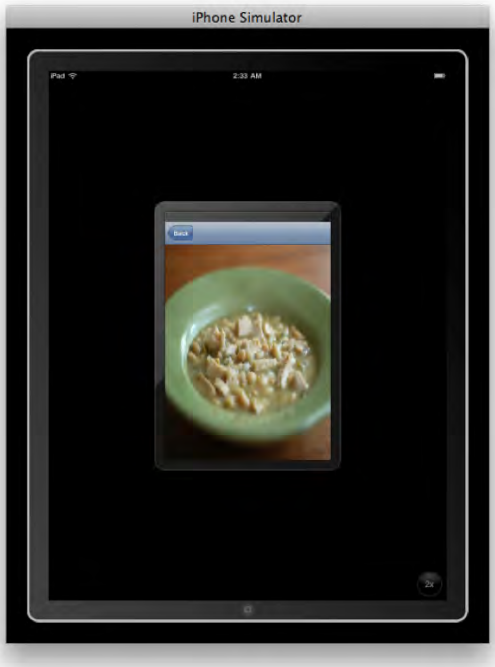


Figure 8–20. *Here is the same image seen after clicking the "1x" button at the lower-right in the iPad Simulator mode.*

Digging the Code

Two general concepts caught my fancy as good destinations for our end-of-chapter detour this time around. The first topic, memory management, is obviously a central and important aspect of computers and programming, and I hope you find this piece edifying. The second topic, reuse identifiers, is at first obscure and odd, but I think you'll see that it's a sort of table space recycling.

Memory Management

Why do we want to release the `imageName` object when we are done using it? How many objects can we make? What do I do if I don't want an object anymore? *Memory management* on the iPhone and iPad is a critical and challenging issue. The iPhone is a very small device compared to your average desktop computer. This makes the available memory on an iPhone comparably small, and we must take every step we can to make sure we use as little memory as possible. The iPad has considerably more memory than the iPhone, but memory management is still crucial. There are three concepts in the arena of memory management that I want you to wrap your brain around: *retain*, *release*, and *autorelease*.

Imagine if you will a piece of paper in a busy restaurant. One of the chefs walks over to the paper and jots something down, something he needs to know. Before he leaves, however, he adds this line to the top of the note: "Don't throw this away. I need it!" Several other chefs and kitchen staff walk by and also write things down on this piece of paper, and they add their own requests to keep the important piece of paper. Through the course of the day and evening, these busy people return and, one by one, scratch off their notes and requests. Finally, the last chef comes by and scratches off her particular line of notes. She notices that no one claims to need the paper anymore, and she proceeds to throw the paper away.

This is exactly how *retain* and *release* work. Retain tells an object: "Hey, stick around, I still need you for something." Release tells an object: "I don't need you anymore." The retain and release mechanisms create a count that is increased and decreased, respectively, depending on an object's ongoing usefulness. This official count is called the *retain count*. It keeps track of how many things need the object to stick around. In the example above, the retain count would be the total number of signatures on the paper. When an object's retain count reaches zero—meaning that no one needs that object anymore—the object is *deallocated*; that is, thrown away.

Those are the basics of memory management on the iPhone and iPad. Is that all? Well, not quite. We also need to understand what kinds of conventions are in place for memory management in Objective-C. Fortunately, these are pretty easy and, if you follow them, they will never steer you wrong. The first rule is that methods containing any of the following terms are "owning" references: *alloc*, *init*, *copy*, *new*. Methods containing any of those words return an object with a retain count of +1. This means that YOU (that's right, you!) are responsible for *calling release* on that object when you are done with it, to reduce the object's *retain count* back to zero. Any method that does not contain any of those words is an *autoreleased* object, so you do not need to track its retain count or worry about it. In fact, you should never attempt to release an autoreleased object returned from such a method unless you first send it a retain message.

Autorelease works by releasing objects at some time in the future. Let's say you just called *alloc/init* on an object, but you aren't sure how long you need to hold onto this object. As I just explained, these commands are *owning references* and, therefore, it is your job to signal their release. Because you often aren't sure of how long to hold on, you instead send this object an *autorelease* message. This makes sure that release is *eventually* called on this object, but you can still use it for a short period of time. Exactly how long depends on many things. If you know you need the object for an extended period, you should really retain it instead.

We are not out of the woods quite yet. There are some other conventions about modifying an object's retain count that we haven't yet covered. Most containers (all Cocoa containers, at any rate) retain their contents. Any methods with the words "add," "remove," or "set" typically change the retain count of the objects that are sent to them. For example, the `-addObject:` method of `NSMutableArray` retains the object passed as an argument. This means that if we don't need the object around anymore, we want to send that object a release method to relinquish ownership. If you've been keeping up,

you will know that the object added to the array will still stick around since it started with a 1 retain count, was added to the array for a retain count of 2, and then we released it, returning its retain count to 1.

See? Memory management isn't that bad! A few simple rules and you're golden. Remember: `alloc/init/copy/new` in methods return *owning references* and require a release when you are done using the object. Methods *without* `alloc/init/copy/new` return *autoreleased* objects, and this means they will be discarded sometime in the future. If you need to use an autoreleased object for a long time, make sure to retain (and subsequently release) that object. On that note, remember to retain autoreleased objects when you are creating your own methods.

Reuse Identifiers

A *reuse identifier* lets the table view reuse old cells without having to make more. It's a clever technique used by efficiency-minded programmers that makes table views fast and easy to use. And, yes, it is digital recycling!

Once we have a viable cell, we need to change its text label. We grab the text that will be used from our "categories" array, since that array holds the text we want to display. We call `objectAtIndex:` to access our array, passing it `[indexPath row]` for the index. This gets the object that we need for the row in the table (that the table view is trying to display). We then set the cell's Text Label text to *the text we got* from the "categories" array, and then we return the cell to be displayed.

Yes, I know, it's a whirlwind of variables, exchanges, counts, and releases. You'll catch on, though. Remember: this is just a quick tour—a bit of dessert after a tasty chapter!

Chapter 9

MapKit

I have been looking forward to writing this chapter on MapKit since the time I first conceived this book. This is the last chapter, and our journey together is almost over. It is fitting that we finish with a bang, and I am confident that this subject will not disappoint you.

During the course of this chapter, you will see that some of the coolest and most successful apps are based upon what we call the MapKit framework. The biggest reason I saved the best for last is that this topic requires as much foundation as possible in order to not overwhelm the student. Teaching this course to a lecture hall full of eager, and mostly novice, programmers, I have learned the hard way that when I succumbed to the students' enthusiasm and tried to teach MapKit midway through the semester, I led the entire class into a brick wall.

Even though MapKit provides us the means to write powerful and vivid apps, it also demands that we be quite aware of what methods, classes, and frameworks are. Originally, the scope of this book didn't include covering all of those concepts, but there was no way I could leave out MapKit!

So, before we begin, we need to sit back and look at a few things. MapKit, as a toolbox, is a very challenging set of utilities and devices, but I will show you some basics and teach you how to use them to successfully and creatively navigate the example in this chapter.

We will first talk about *frameworks* and *classes*. Next, we will see what MapKit can already do without your having to program anything. Then we will jump in at the deep end to see what other programmers have done using MapKit, and glean what we can from them. After honing your understanding of methods, and once you have acquired a decent grasp of frameworks, classes, and other Apple goodies contained in MapKit, we'll tackle—*gently*—the exercise.

In the latter half of the chapter, I will serve you an extended dessert in the "Digging My Students' MapKit Code" section. Rather than finishing with an eclectic mix of technical references, I will instead present three of my students' efforts in MapKit-related projects. I am hopeful that when you see what these representative students were able to accomplish, shortly after they passed my class, you will feel even more inspired to set your course for the next challenge.

My objective, then, is to get each of you to a place where you can say: *I have programmed a basic iPad MapKit app, and I understand how to move forward with confidence into more advanced territory*.

A Little about Frameworks

When Steve Jobs was fired from Apple, he formed a business called NeXT. His company produced beautiful, black, streamlined computers in the early '90s that made me drool with envy, for a few of my professors owned a NeXT computer. The most profound aspect of this outfit was not that they cranked out these black, streamlined boxes, but rather that they utilized a language called Objective-C. Jobs found that, even though it was difficult to program in this complex language, the code it produced was able to "talk to" the microprocessor elegantly. So what's this got to do with MapKit?!

What NeXT did was create *frameworks* of complex Objective-C code that you can look at as tools that a carpenter might have in his toolbox. When we use MapKit, we are actually bringing in to our own code a framework of map-related tools—just as a carpenter may have one set of tools for cabinetry and another set, specially made for intricate furniture. These tools will differ greatly from the kinds of tools that a roofing carpenter may use.

To this end, we will bring into Xcode two frameworks that we have not used before. It will be almost as if you had been learning techniques as a flooring and cabinetry carpenter in Chapters 1-8, but today we are going to the hardware store to get outfitted for our next gig: audio-video installations in walls and ceilings. Thus, before we head on to the next program, we are going to have to go buy two brand-new tools. One of our new tools, the CoreLocation framework, shows us where we are geographically. The other tool, MapKit, enables us to interact with maps in a multiplicity of ways.

As you know, the way users interact with the iPad and iPhone is completely different from anything seen before. Before the advent of these slick devices, 99% of all interactions we had with computers were based upon the mouse and keyboard. As you have been learning, though, from the examples that you've programmed, we have used unique methods and classes to jump between screens and to sense when a user is pinching, tapping, or scrolling on the screen. To this already formidable set of tools, we are now going to add CoreLocation and MapKit frameworks.

Most of the programming we have explored has been relatively transparent. In this chapter, it won't be so transparent. For example, we will have to really maintain our focus in order to keep track of how MapKit knows where we are—on a map. We'll look into how it follows our finger interactions and how it knows where we are in terms of the various screens and views associated with maps.

One of the central areas of iPad/iPhone app development is *event handling*. This is the part that confused most of my students when we got to this lesson, and I will do my best to keep you from wandering off into the briar patch. If I do a decent job of introducing the concepts of frameworks and classes to you, then you will not be burdened by having to think too much about event handling. You can get an idea of the

scope of this topic by considering that while part of your app is keeping track of interacting with a map and with a GPS satellite, another portion of your code has to always be looking at when the user is going to direct the program to a new event.

Important Things to Know

There are three important things to know about the foundation of map-related applications in the iPad and iPhone arena. Apps rely on three important tools: MapKit, CoreLocation, and the `MKAnnotationView` class reference. As I have indicated, we are not going to involve ourselves with how these sophisticated tools work so much as to practice the art of deciding *when* to reach for *which* tool in your newly expanded toolbox.

Among other things, these tools allow us to display maps in our applications, to use annotations, to work with something called Geocoding (which works with longitude and latitude), and to interact with our location (via CoreLocation).

When we want to interact effortlessly with Google Maps, we will use the Apple-provided MapKit framework. When we want to get our location or do cool things using GPS-satellite technology (with Google Maps), we will use the CoreLocation framework. Finally, when we want to place pins on a map, create references, draw chevron marks, or insert an image of your dog showing where he is on a map, we will call these *annotations* and, thus, use `MKAnnotationView`.

Preinstalled MapKit Apps

So that you can take maximum advantage of the new ideas presented in this chapter, and be prepared to stretch and expand into a new level of creativity, we will first go on a little tour of existing apps, preinstalled on the iPad and iPhone. It is important that you become familiar with these so that you can more easily add bells and whistles to your own creations—on top of these ready-made "map apps," as described at `Apple.com`.

Find Locations

The Find Locations app (see Figure 9–1), preinstalled on iPhone 3GS and iPad, finds your location quickly and accurately via GPS, Wi-Fi, and cellular towers. Drop a pin to mark your location or share it with others via email or MMS.

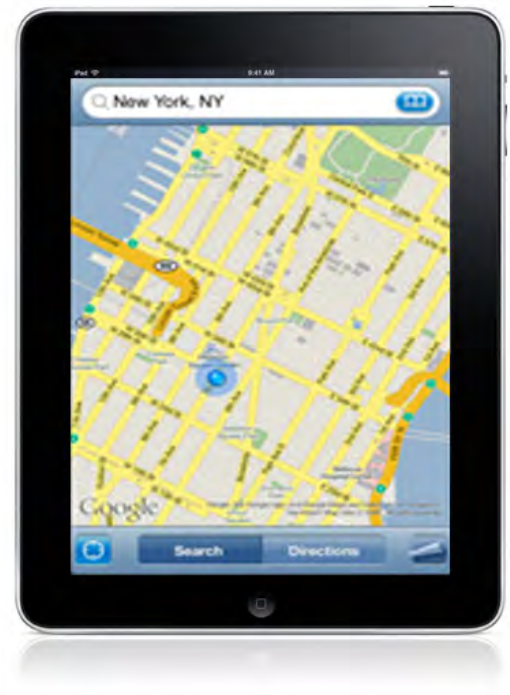


Figure 9–1. *Find Locations—a powerful zooming map function on the iPhone/iPad.*

Get Directions

Shown in Figure 9–2, the preinstalled Get Directions app lets you view a list of turn-by-turn directions or follow a highlighted map route and track your progress with GPS. You specify whether you'd like walking or driving directions, or see what time the next train or bus leaves with public transit directions.

Figure 9–2. *Get Directions – use this in conjunction with, or in lieu of, the visual map (with highlighted route).*

See Which Way You're Facing

In the preinstalled See Which Way You're Facing app, shown in Figure 9–3, a built-in digital compass rotates maps so that they always match the direction you're facing. You can also use the compass on its own.

Figure 9–3. *See which way you're facing – shows your orientation with built-in compass (on Model 3GS).*

See Traffic

The preinstalled iPhone app See Traffic, illustrated in Figure 9–4, shows you live traffic information, indicating traffic speed along your route in easy-to-read green, red, and yellow highlights.

Figure 9–4. *See Traffic—one of many possibilities when running 'Maps' on iPhone/iPad.*

Search for a Location

In the Search For A Location mode, shown in Figure 9–5, you can find locations by address or by keyword. For example, search for “coffee” to see every cafe near you. When you find what you’re looking for, tap the phone number to call (on the iPhone), tap the web address to open the website in Safari, or add it to Contacts for future reference.

Figure 9–5. *The Search For A Location mode is a quick and powerful capability that brings specific destination information to your fingertips.*

Change Your View

The preinstalled Change Your View app, shown in Figure 9–6, lets you switch between map view, satellite view, hybrid view, and street view. You can double-tap or pinch to zoom in and out.

Figure 9–6. *The Change Your View mode is a standard feature of 'Maps' on iPhone/iPad.*

Cool and Popular MapKit Apps to Inspire You

I found that it really helped my students when, after showing them the prebuilt apps, we spent some time to review some super-cool third-party MapKit apps … to inspire them and get their brains storming. So, imagine you are sitting with us and taking this brief tour as well. Here are nine MapKit apps that caught my eye, some of which I use regularly.

- **MapMyRide:** This is a MapKit app I use all the time. I simply turn it on and start riding around on my bike. It tracks my speed, time, and mileage, as well as the elevation I ride. It then takes into account my age, gender, and body weight, and it tells me how many calories I burned. [On a good day, I can almost burn off two doughnuts!] The point is that this application calculates all these things while I'm just riding along huffing and puffing! When I get home, I can see the route on my computer. It does most of its work by using and manipulating preinstalled MapKit apps.
- **QuikMaps:** This is a do-it-yourself map app that allows you to doodle on the map. It integrates with a number of places, including your website, Google Earth, or even your GPS.
- **360 Cities—The World In Virtual Reality:** This shows 360-degree panoramas of over 50 world cities and 6000 panoramas. It is the perfect technology for real estate agents, tour guides, and adventurers.
- **Cool Maps—7 Wonders of the World:** This shows the seven wonders of the ancient world, and the seven wonders of the modern world, including natural wonders, underwater wonders, strange wonders, and local wonders. I am impressed with how slick the programmers have made the touch and feel of the app.
- **Blipstar:** This app converts Internet business URL addresses to their corresponding brick–and-mortar stores, presented on a cool map.
- **Twitter Spy:** This app lets people see where the person who is tweeting them is currently located. Yep—wacky and crazy, but true.
- **Geo IP Tool:** This app displays the longitude and latitude information of businesses on the Web, and then shows you the best ways to get there.
- **Map Tunneling Tool:** This one is just funny and clever. Imagine where you would come out if you began digging a hole straight down—from wherever. *Is the answer always China?*
- **Tall Eye:** This app shows you where you will go if you walk directly, in a straight line, around the earth, starting at one point and staying on a specific bearing all the way around.

MapKit_01: A View-Based Application

For your final exercise, you are going to begin with some boilerplate code that suits our basic requirements, and you will then modify it from there. I will tour you through some of the same building blocks and files that you've seen throughout this book, and I will be challenging you to see what areas of the code are pretty much the same as what you've encountered and what areas are different – given the nature of this application.

The ability to recognize patterns and to see structures just under the surface is a powerful aptitude that we all have, but we programmers cultivate ours to a heightened degree. So, be on the lookout for phrases, statements, suffixes, prefixes – grammatical hooks, as it were – that you can build on, modify, and/or revise. Play a little game: see if you can anticipate some of the moves I will have you take.

Possible Prepping for the App

We are going to consider a wide variety of components that we may want to build in to our app. Before that, though, I want to make sure we are all on the same page with terminology. We programmers need to recall some basic earth science and geography so that our code will be as effective as possible.

When we direct the computer to animate a pin dropping down, with annotations, onto a specific location, giving "longitude" and "latitude," we need to know what these terms really mean. *Lines of latitude* are the imaginary lines that circle the globe "horizontally," running east to west (or west to east). These invisible lines are measured in degrees, minutes, and seconds, north or south of the Equator. The Equator is the elliptical locus of points on the Earth's surface midway between the poles, which themselves are *real* points physically—defined by the Earth's rotation on its axis. Lines of latitude are often referred to as parallels. The North Pole is 90 degrees north latitude; the South Pole is 90 degrees south latitude.

Lines of longitude, on the other hand, which are often called meridians, are imaginary "vertical" lines (ellipses) always crossing through the North and South Poles. They are also measured in degrees, minutes, and seconds, east or west of the Prime Meridian, an arbitrary standard that runs through Greenwich, England. Unlike the Equator, which goes all the way around the world—360 degrees, the Prime Meridian (0 degrees longitude) is a semi-circle (semi-ellipse), extending from the North Pole to the South Pole; the other half of the arc is called the International Date Line, and it is defined as 180 degrees east and/or 180 degrees west longitude.

For our Chapter 9 app, the example I have used to demonstrate the "pin drop" on location is my office at the University of Colorado at Colorado Springs. You, of course, can use any location you choose. You may want to use your own address, or a well-known landmark. To do this, you must get the latitude and longitude values of that location,—most likely from Google Maps or a direct GPS reading. There are many sites on the Internet, too, where you can find these coordinates; Figure 9–7 illustrates one of them, `www.batchgeocode.com`.

Figure 9–7. *http://www.batchgeocode.com/lookup is one of many Internet sites where one can enter an address and receive its longitudinal and latitudinal coordinates.*

Here's a thought … let's start at the end of our process and think backwards for a minute. Go ahead and jump forward in this chapter for a sneak peek at what the app will look like—what results it will return if all goes well. In Figure 9–25, you will see a picture of a hybrid map showing a red pin that's sitting on top of a building. That's the Engineering Building at the University of Colorado at Colorado Springs, and the pin is located right above my office. The next picture has what we call an *annotation*, which is the text. "Dr. Rory Lewis" is the title, and "University of Colorado at Colorado Springs" is the subtitle.

Later in the tutorial, you will see that we need to be careful about what is the title and what is the subtitle. We control the color of the pin, and we decide on the style of animation—how the pin drops onto the map image.

In Figure 9–27, we zoom way out and see a super high-level view—from space. This allows us to see huge regions, but we obviously lose detail. This shows, too, the difference between the user's current location (blue dot) and a highlighted location (red pin). Interestingly, the iPad Simulator assumes the user is in Cupertino, CA—the location of Apple's headquarters.

This is a good place for me to remind you of the title of this book: *iPhone and iPad Apps for Absolute Beginners*. Take a deep breath! Even if I were meeting your greatest expectations of teaching you the most you've ever been taught, and even if you were meeting your greatest expectations of yourself—learning so much complexity in such a short time, you would still *not* become an expert in this challenging area of MapKit code! My humble goal here is not fluency, but reasonable familiarity and a sense of what lies ahead.

If that sounds right, let's get on with it.

Preliminaries

As in previous chapters, please download and extract images and code for this chapter. Navigate to `http://www.rorylewis.com/xCode/011_MapKit_01.zip` and download its contents. Then, extract the files onto your beautifully clean desktop.

The zip file contains several folders of content: `MapKit_01.xcodeproj`, `Classes`, and `build`, and some individual files: `MapKit_01ViewController.xib`, `MapKit_01-Info.plist`, `MainWindow.xib`, `MapKit_01_Prefix.pch`, and `main.m`.

Once you have extracted all the files, remember to delete the `011_MapKit_01.zip` and `MapKit_01` folders. We don't want any of your files to be overwritten and conflict with your exercise code.

To view the screencast of this chapter's exercise, go to `http://www.rorylewis.com/docs/02_iPad_iPhone/06_iphone_Movies/011_MapKit.htm`.

A New View-Based Template

1. Open Xcode and enter ⌘⇧N, as shown in Figure 9–8, and click on the View-based Application template. Call it `MapKit_01` and save it to your desktop. A folder bearing that name appears on the desktop.
2. The first thing we need to do is add two frameworks: CoreLocation and MapKit. Right-click on the `Frameworks` folder, click Add and then Existing Frameworks, and then, in the drop-down menu that appears, select both CoreLocation and MapKit Framework, as shown in Figure 9–9.

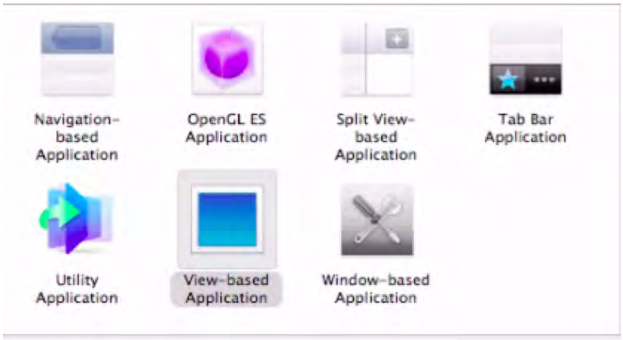


Figure 9–8. *Select the View-based Application icon, and save your new file to your desktop.*

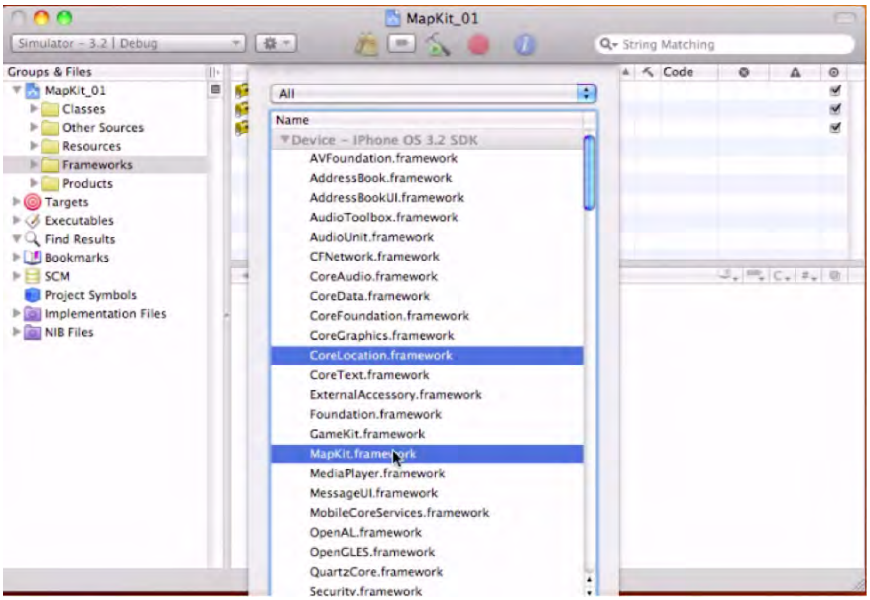


Figure 9–9. *Select the CoreLocation and MapKit frameworks.*

Adding the Annotation File

3. The next thing we need is a means to control our annotation. For that, we will create a UIViewController subclass that will control all the characteristics we want to display on this annotation. Click the `Classes` folder and enter ⌘N as shown in Figure 9–10. Because this controller will be in charge of controlling annotations for your position, name it "myPos."

   Make sure that your options are *not* checked. These include Targeted for iPad, UITableViewController Subclass, and XIB Interface. This will automatically create a `myPos` header file and implementation file in your `Classes` folder once you hit the Next button or Return.

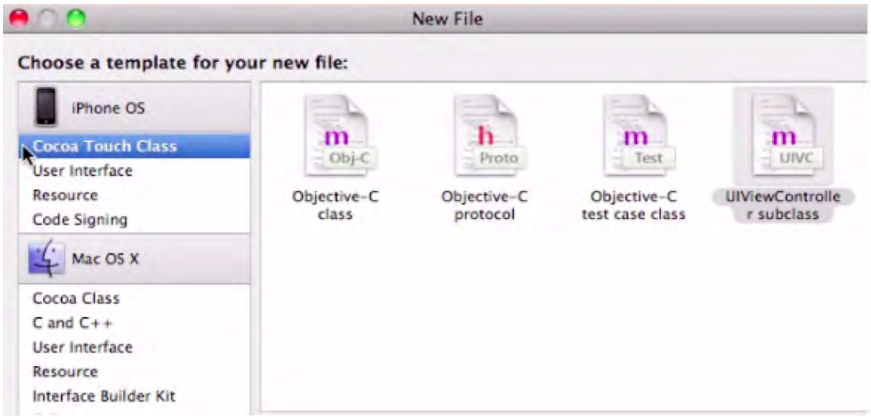


Figure 9–10. *Create a new UIViewController subclass, and name it "myPos."*

It's Already Working!

Believe it or not, we already have enough to show a map working in our iPad or iPhone. As I mentioned in the introduction to this chapter, Apple programmers have included an enormous amount of complex Objective-C code in the two frameworks that we imported.

4. To examine the details, let's open up your nib file by going into the `Resources` folder and opening up the `MapKit_01ViewController.xib` file. Click on it as shown in Figure 9–11.

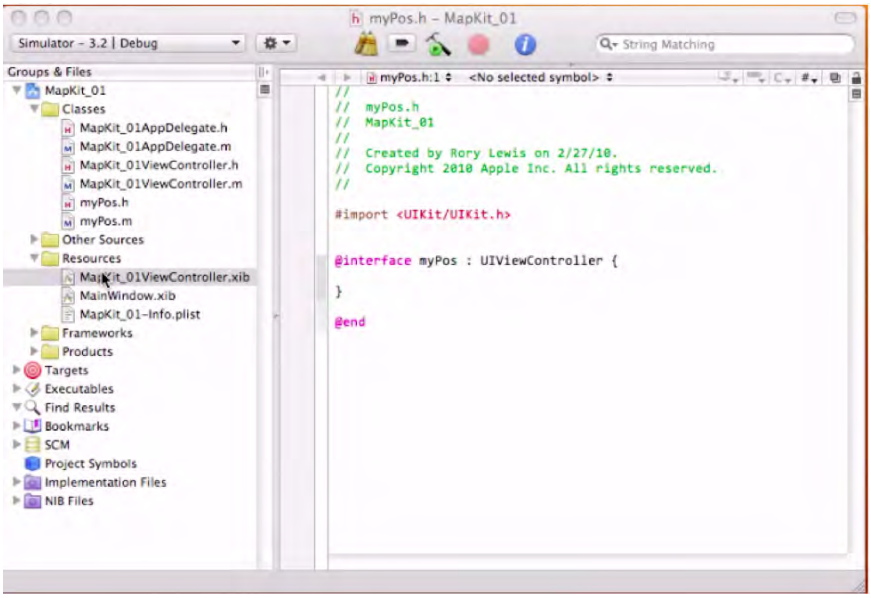


Figure 9–11. *Open up the MapKit_01ViewController.xib file to see how the frameworks can already implement a working map.*

Check It Out—the iPad Simulator

5. After opening up Interface Builder, drag a Map View object from the Library and drop it onto the View. Then go to your Inspector and click Show User Location. With this done, click ⌘S to save, and then go back to your `Classes` folder in Xcode. Click in any file and hit ⌘Return, and the iPad Simulator will appear, showing a map of the world. Shortly thereafter, a blue marker will drop on Cupertino, CA, as a representation of your "current" location.

NOTE: Because you are not operating on an actual iPad or iPhone, the simulator pretends that your Geocode Location is that of Apple's headquarters in Cupertino, CA, the heart of Silicon Valley. Once your app—at this stage of development—is put on a real iPad or iPhone, the GPS system in your device will use the `CoreLocation` framework you imported in Step 2 to get your actual location.

Your screen might look slightly different initially because, by default, it will show the iPhone screen embedded in your iPad simulator. To make the screen in your iPad look exactly like my screen in Figure 9–12, simply click the Enlarge button at the bottom-right corner of the inset screen of your iPad.

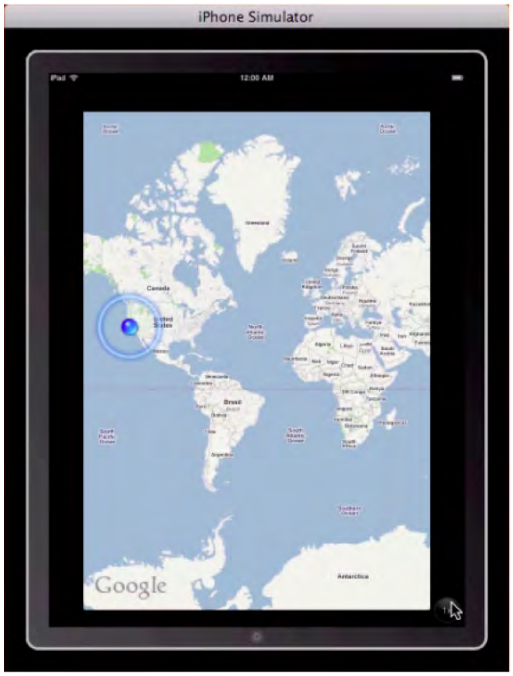


Figure 9–12. *You already have a working map appearing in the iPad simulator.*

Make It Look a Little Bit Better

Even though we have a map appearing in our iPad that shows that our CoreLocation framework is working, we want to make it look a lot better than this. Specifically, we would like it to show the location of a point of our choosing, and we're going to want to have a pin drop down too, and include an annotation in which a title and subtitle explain something about our location. This information will appear in a little black box when the user clicks on the pinhead, of whatever color we choose.

6. We will start by declaring our classes, methods, and outlets in our `ViewController.h` file and then implementing them in our `ViewController.m` file. At that point, we will make some adjustments to the `myPos` and `AppDelegate` files.

7. Open up your header file by clicking on `MapKit_01ViewController.h` as shown in Figure 9–13. The first thing we need to do is tell our app that we have imported the MapKit framework; we do this by inserting `#import <MapKit/MapKit.h>` right under the line `#import <UIKit/UIKit.h>`. The next thing we will do is tell the header file that we will be using the `MKMapViewDelegate` protocol. This protocol defines a set of optional methods that our app will use to receive map update records.

8. Now we also need to add an outlet with a pointer to the `MKMapView` class. We do that by typing in `IBOutlet MKMapView *mapView`, which declares an object of type `MKMapView`. The last thing we need to do is define the @property, by entering

```
@property (nonatomic, retain) IBOutlet MKMapView *mapView;

@end
```

Your code, at this point, should look as follows:

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

@class MyPos;

@interface MapKit_01ViewController : UIViewController <MKMapViewDelegate>
{
        IBOutlet MKMapView *mapView;
}

@property (nonatomic, retain) IBOutlet MKMapView *mapView;

@end
```

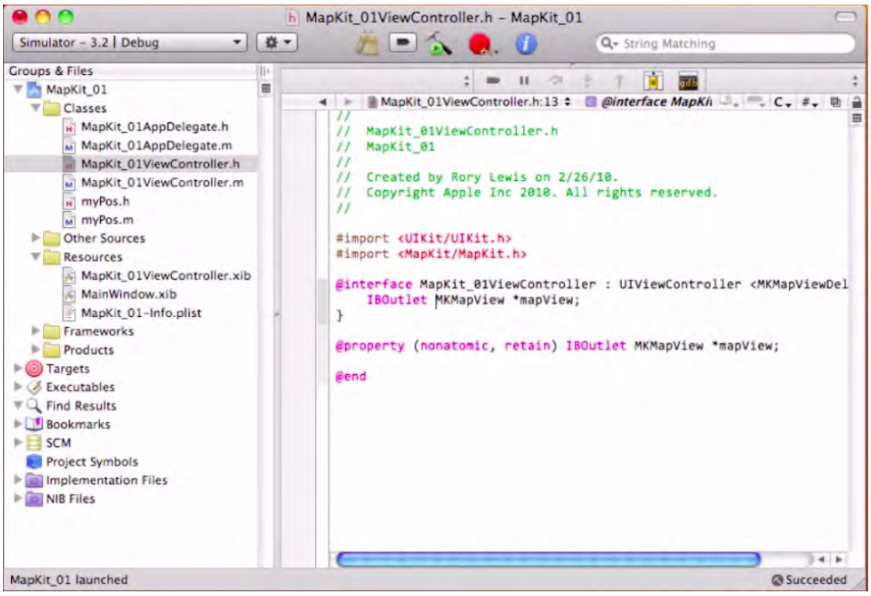


Figure 9–13. *Coding our ViewController header file.*

Dealing with the Implementation

As mentioned in the introduction to this chapter, controlling and working with the MapKit and CoreLocation frameworks is not a trivial matter. Daunting as these areas can be, I could not leave them out of this book. We proceed on the basis that you have learned by now to look for familiar patterns, integrate what you can, and just follow directions when things get a bit dicey!

> **NOTE:** What we are going to do in our implementation file is synthesize our property and release it in the `dealloc` method.

9. As shown in Figure 9–14, we now copy the `dealloc` statement from the bottom of the implementation file, `MapKit_01ViewController.m`. We are going to paste it at the top of the file, immediately below the synthesize statement you just wrote. Then add a release for the mapView property, as shown in Figure 9–15.

   The reason I asked you to put it right at the top is that I now want you to delete everything below it, down to the `viewDidLoad` method. See Figure 9–16.

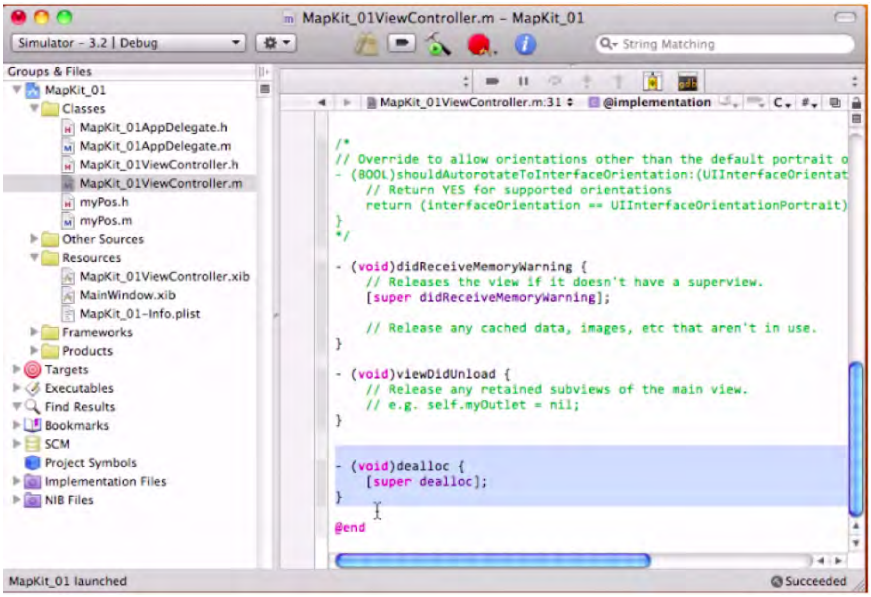


Figure 9–14. *Copy the dealloc statement from the bottom of the implementation file.*

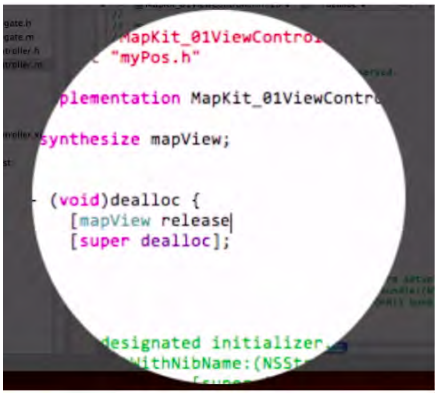


Figure 9–15. *Paste it at the top of the file, just below the synthesize mapView statement. Also, add a release for the mapView property.*

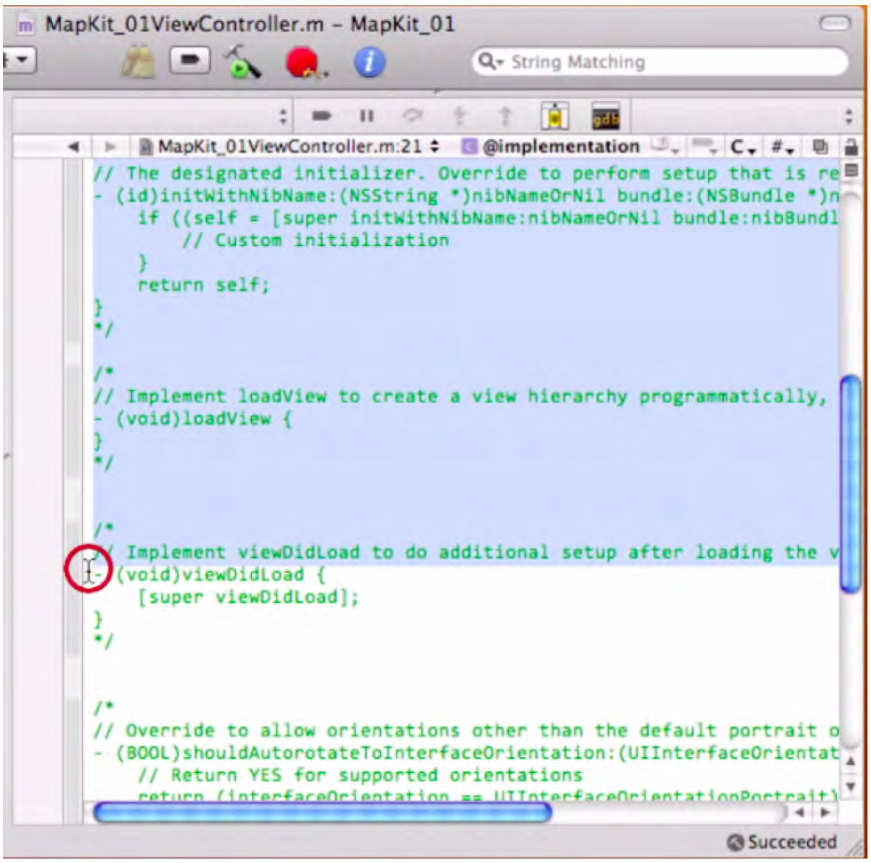


Figure 9–16. *The next step is deleting boilerplate implementation code all the way down to the (void)viewDidLoad { line.*

Let's think about this … We need to add the `viewDidLoad` method, within which we will set the map type and enable zooming, scrolling, and so on. In our case, we will set the map type to a Hybrid map. If you prefer, though, you may choose to use a Satellite map or a Street map.

At this point, we will bring in the location I requested of you in the prep section—some place of interest or personal significance. If you don't have a preference, you can go ahead and use mine, as shown in the code that follows.

10. The first thing we do is set all the coordinate regions to zeros. Then we enter coordinates of our place of interest—which, for me, is my office at the University of Colorado at Colorado Springs. I enter `Region.center.latitude = 38.893432;` (the positive value denotes north of the Equator) and `region.center.longitude = -104.800161;` (the negative sign denotes west of the Prime Meridian). Related to these parameters, we need to set the latitude and longitude `Delta = 0.01f`. If your math or physics is rusty, recall that "delta" refers to the change, or difference, between two values. Finally, for the `viewDidLoad`, I have chosen to animate the pin when it drops by using the code `[mapView setRegion:region animated:YES]`. Refer to Figure 9–17.

11. The next action is to set the view controller class as the *delegate*, the role that will handle the interaction between the frameworks of our mapView. We do this with `[mapView setDelegate:self]`.

Regarding the dropped pin and the attached label, we need to make the Annotation Object the holder of the information of our coordinates. Our Annotation View is the type of view associated with the Annotation Object. Our Annotation Object needs to comply with all the rules we will set forth in our MKAnnotation Protocol. In order to create this Annotation Object, we must define a new class, which we did when we created the `myPos` classes.

12. We now need to instantiate this myPos object and add it to our map. To do this, we add the delegate function that will display the annotations on to our map. We start by having myPos name a pointer we'll call "ann." Next, we set the title, and in my case I chose to use my name. So, we get `ann.title = @"Dr. Rory Lewis"`. We handle the subtitle similarly: `ann.subtitle = @"University of Colorado at Colorado Springs"`. We also want the pin to drop in the center of the map: `ann.coordinate = region.center`. Reference all of the above with `[mapView addAnnotation:ann]`.

At this point, we will take advantage of a boilerplate method of code that most MapKit maps use.

> **NOTE:** We seldom change these chunks of code, and by the time you read this book, this next set of code may be part of a new function or a new class. The reason is that when people start using the same piece of code over and over, referring to it as "boilerplate code," that's about the time Apple decides to make a new class or function out of it, and sets it to a specific name.

For now, I have included this boilerplate code in the `Downloads` folder for this app. This code does two things:

- It creates a *delegate method* that manages our annotation during zooming and scrolling. In other words, it keeps track of where we are—even when the user scrolls, zooms in, or zooms out of our map.
- It creates a *static identifier*, which controls our "queue meaning." If it can't *dequeue* our annotation, it will allocate one that we choose. I have also included code that changes the pin color to red. Also, I have allowed *callout views*.

Make sure your `- (BOOL)shouldAutorotateToInterfaceOrientation` is activated along with your `- (void)didReceiveMemoryWarning`. Your code should now look like this:

```
#import "MapKit_01ViewController.h"
#import "MyPos.h"

@implementation MapKit_01ViewController

@synthesize mapView;

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

- (void)viewDidLoad
{
    [super viewDidLoad];

        [mapView setMapType:MKMapTypeStandard];
        [mapView setZoomEnabled:YES];
        [mapView setScrollEnabled:YES];

        MKCoordinateRegion region = { {0.0, 0.0 }, { 0.0, 0.0 } };
        region.center.latitude = 38.893432 ;
        region.center.longitude = -104.800161;
        region.span.longitudeDelta = 0.01f;
        region.span.latitudeDelta = 0.01f;
        [mapView setRegion:region animated:YES];

        [mapView setDelegate:self];

        MyPos *ann = [[MyPos alloc] init];
        ann.title = @"Dr. Rory Lewis";
        ann.subtitle = @"University of Colorado at Colorado Springs";
        ann.coordinate = region.center;
        [mapView addAnnotation:ann];

}

- (MKAnnotationView *)mapView:(MKMapView *)mV viewForAnnotation:↩
(id <MKAnnotation>)annotation
{
        MKPinAnnotationView *pinView = nil;
        if(annotation != mapView.userLocation)
        {
```

```
                static NSString *defaultPinID = @"com.invasivecode.pin";
                pinView = (MKPinAnnotationView *)[mapView↩
 dequeueReusableAnnotationViewWithIdentifier:defaultPinID];
                if ( pinView == nil )
                        pinView = [[[MKPinAnnotationView alloc]↩
 initWithAnnotation:annotation reuseIdentifier:defaultPinID] autorelease];

                pinView.pinColor = MKPinAnnotationColorRed;
                pinView.canShowCallout = YES;
                pinView.animatesDrop = YES;
        }
        else
        {
                [mapView.userLocation setTitle:@"I am here"];
        }

    return pinView;
}

- (BOOL)shouldAutorotateToInterfaceOrientation:↩
(UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

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

- (void)viewDidUnload
{

}

@end
```

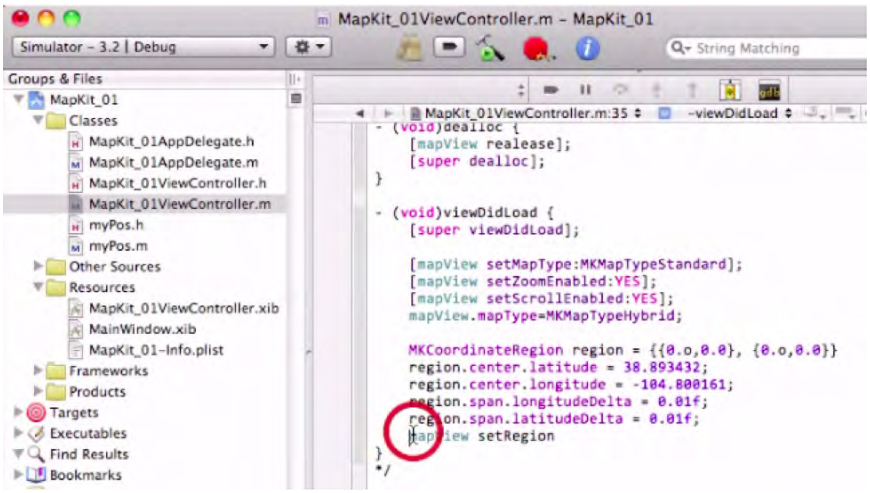


Figure 9–17. *Adding coordinates, parameters, and our region setting.*

Coding the myPos.h File

13. The first thing we do after opening the `myPos.h` file is replace the import line `#import <UIKit/UIKit.h>` with `#import <Foundation/Foundation.h>`. We do this because the foundation framework pulls in all of our foundation framework classes. The next thing we do is to add MapKit by entering `#import <MapKit/MKAnnotation.h>`.

14. We set our CLLocation Class Reference to incorporate the geographical coordinates and altitude of our device, as seen in Figure 9–18. We do that with this line:

```
CLLocationCoordinate2D coordinate; NSString *title; NSString *subtitle
```

15. Finally, we create "@property" statements for the location, title, and subtitle, as shown in the code that follows. Once you have made these additions, save your work. See Figure 9–19.

```
#import <Foundation/Foundation.h>
#import <MapKit/MKAnnotation.h>

@interface MyPos : NSObject <MKAnnotation>
{
        CLLocationCoordinate2D coordinate;
        NSString *title;
        NSString *subtitle;
}

@property (nonatomic, assign) CLLocationCoordinate2D coordinate;
@property (nonatomic, copy) NSString *title;
@property (nonatomic, copy) NSString *subtitle;

@end
```

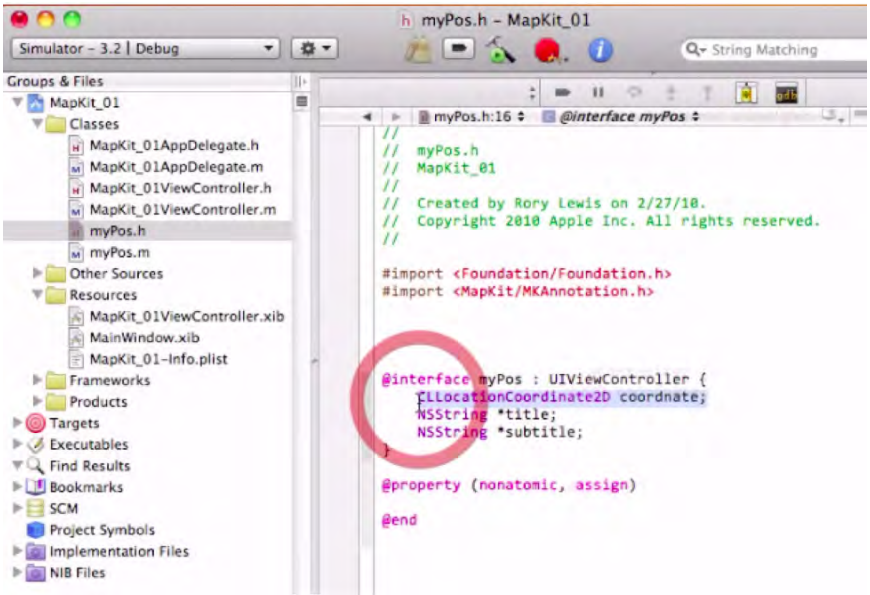


Figure 9–18. *Incorporate the geographical coordinates and altitude of our device.*

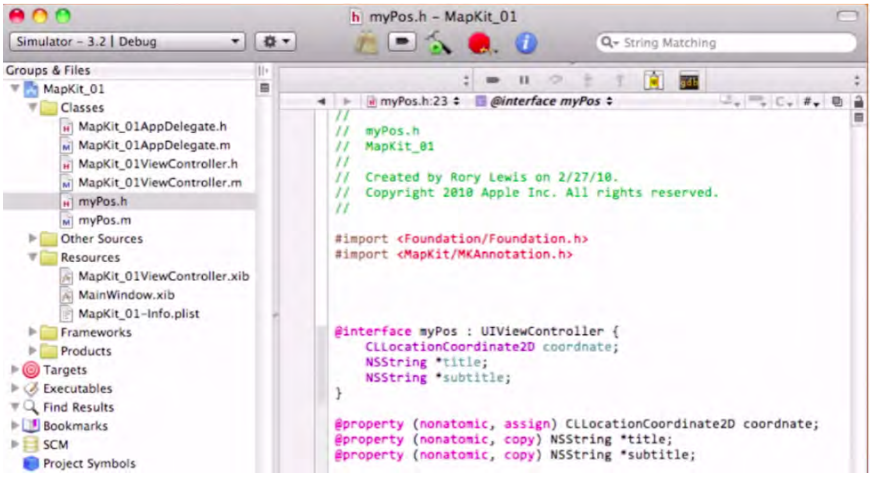


Figure 9–19. *Your "myPos" header file should look like this before you save it.*

The myPos.m File

16. Here we simply synthesize our coordinate, title, and subtitle with an `@synthesize` statement that includes `coordinate`, `title`, and `subtitle`. Then, release the `title` and `subtitle` in our `dealloc` (deallocation) statement. Once done, your file should look similar to Figure 9–20. Save your work on this file.

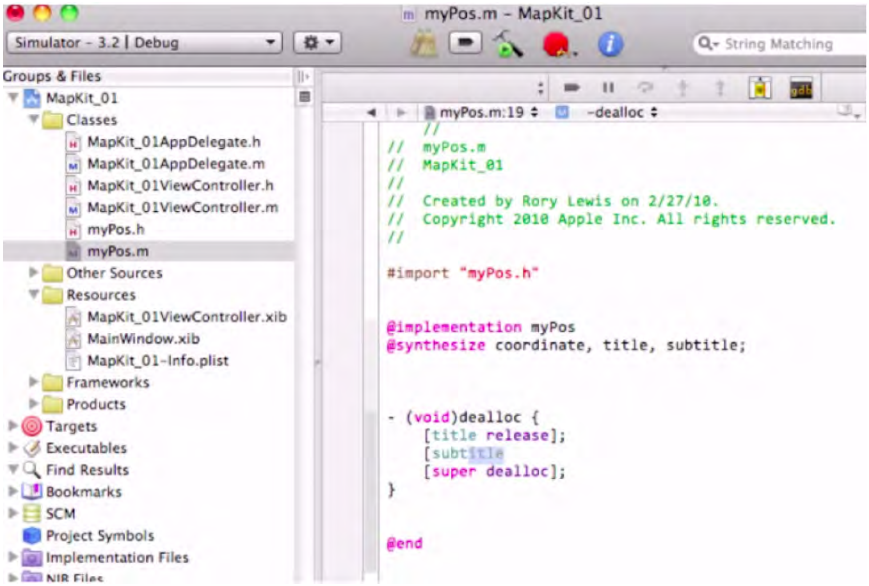


Figure 9–20. *This is how your myPos implementation file looks before you save it.*

The AppDelegate Files

As you can see by studying Figure 9–21, we don't have to do anything with the AppDelegate header file. Everything we need is already there. We will, however, add a little bit to the AppDelegate implementation file.

17. Open `MapKit_01AppDelegate.m`. First, we need to override the point for customization after our app launches. We do this with `[windowaddSubview:viewController.view]` and `[window makeKeyAndVisible]`. Make sure that your `dealloc` statement deallocates both the `viewController` and the `window`.

> **NOTE:** For some reason, when I set my app to default to the iPad platform, it did not dealloc as expected. I am operating on beta version, so this might be a minor error. Nevertheless, it is always a great idea to check on this parameter. See Figure 9–22.

18. Save your work, and then go to the `Resources` folder.

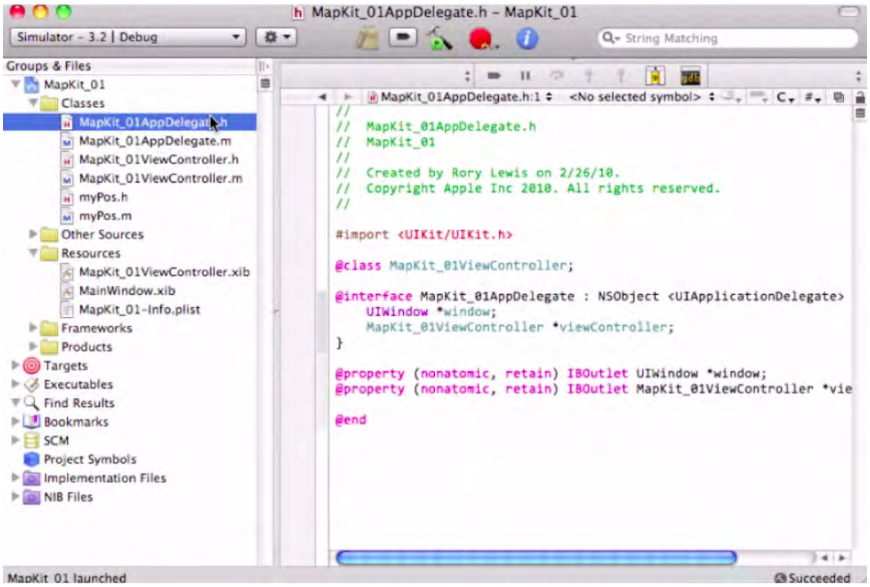


Figure 9–21. *The AppDelegate header file is perfect. Don't change anything!*

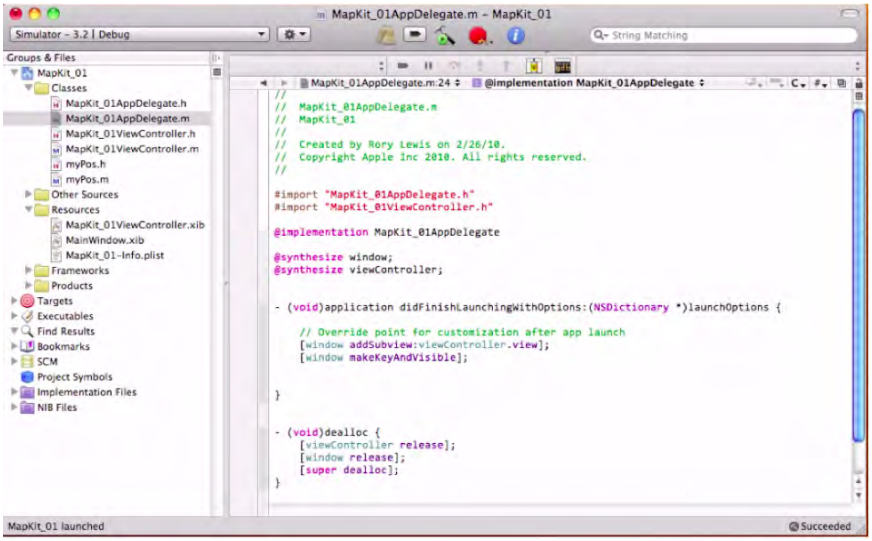


Figure 9–22. *The AppDelegate implementation file. Get in the habit of checking your dealloc statements.*

Connect MapView with MKMapView

19. As shown in Figure 9–23, go to the `Resources` folder to open up your nib file. In Interface Builder, connect the mapView of your File's Owner to your MKMapView, as shown in Figure 9–24.

20. Then, go back to Xcode and compile. The iPad simulation will appear, and a pin will drop down onto the top of my office—or the different location you chose. The pin will be red, and if you click on the pinhead, it will display the annotation. As shown in Figures 9–25 through 9–27, you can go back and forth between the iPad view and the iPhone view.

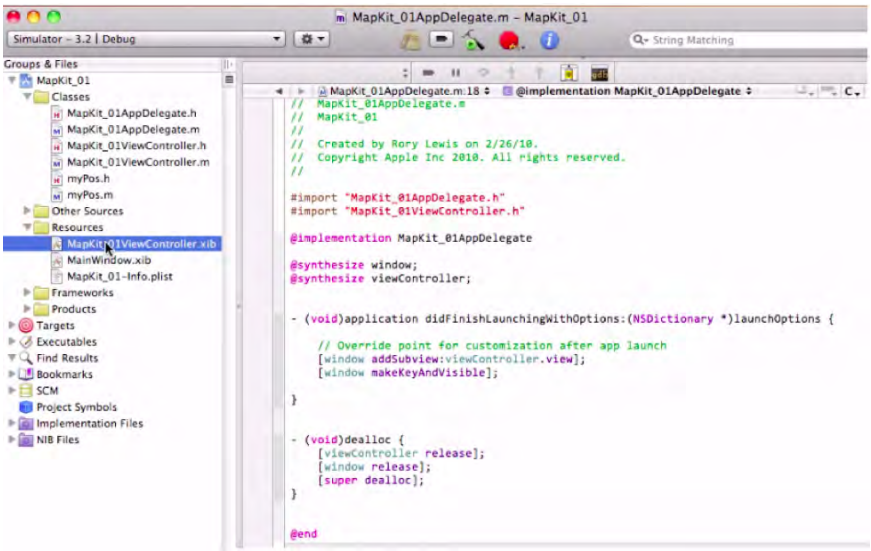


Figure 9–23. *In the Resources folder, click on the MapKit_01ViewController.xib file so you can make the connections and get this app running!*

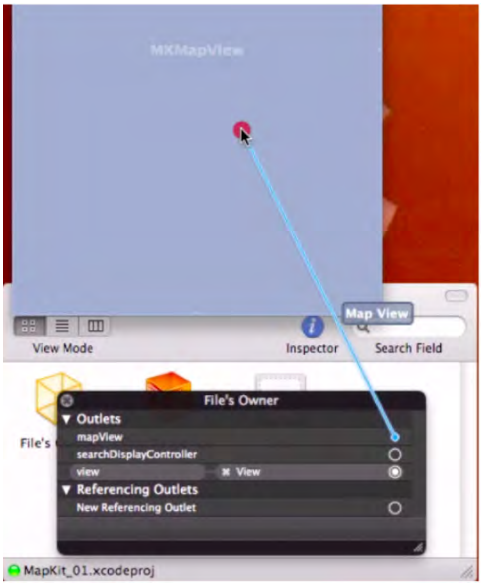


Figure 9–24. *Connect the mapView of your File's Owner to your MKMapView.*

Congratulations! Once again, you have successfully implemented an app of fair complexity, regardless of the fact that you started with a body of code that you modified. As you compare your own Simulator to the images ahead, bask in the glow of accomplishment.

Then, perhaps after a brief rest, I hope you will venture forward to see if some student examples in the "Digging My Students' MapKit Code" section whet your appetite for further development and challenge.

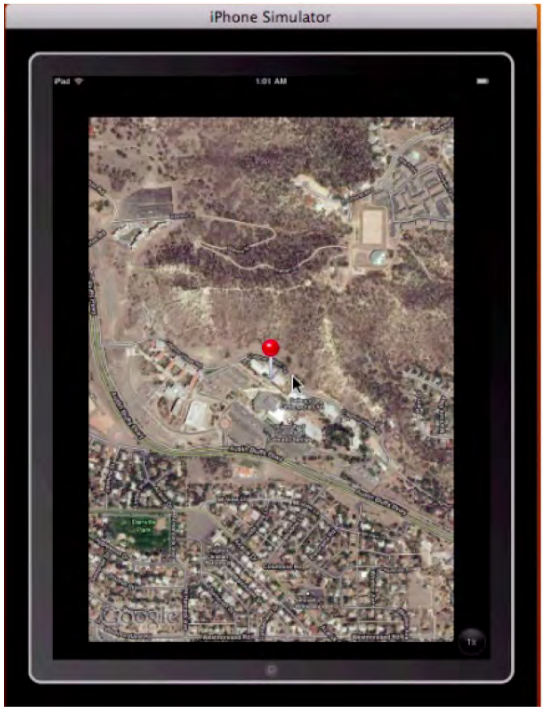


Figure 9–25. *The pin immediately comes down onto our set coordinates.*

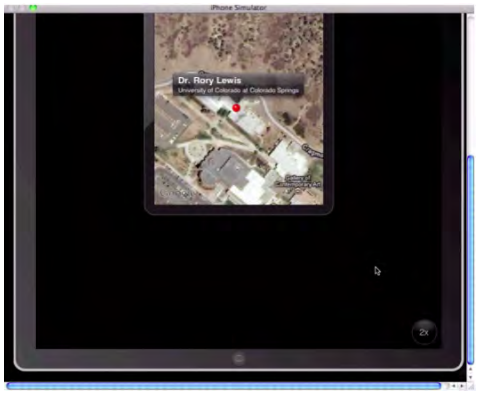


Figure 9–26. *Clicking on the Pin, our annotation opens up. Here we have the view of the iPad Simulator set to the embedded iPhone view.*

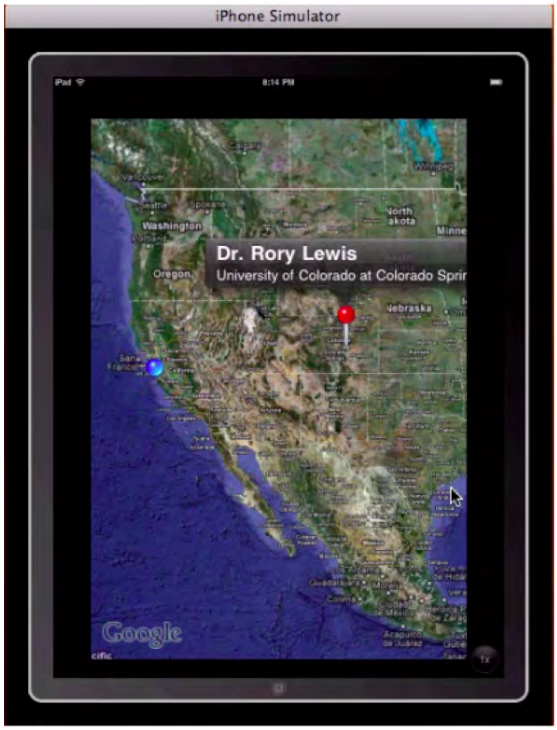


Figure 9–27. *Zooming out we see how far away we are from our "current location," which, on the iPad simulator, defaults to Apple's headquarters in Cupertino, CA.*

Digging My Students' MapKit Code

When people come up to me and say, "Hey, Dr. Lewis, I have this really *great idea* for a new app ...," it is amazing how often it involves using the MapKit framework. We have seen how fun and sexy this stuff is, and now you have likely gathered that delving into the code can turn into quite a complexity.

As a final buffet of tasty, high-calorie, high-tech fun and flash, I am going to share some final project scenarios with you. I certainly hope you actively follow along here, but I also want to honor the fact that you're done. You already succeeded in making it to the end of Chapter 9. So remember, this section here is like one of those "bonus feature" DVDs that Hollywood loves to include—at no extra cost. *Relax and enjoy!*

Parsing to MapKit from the Internet

A little background: I presented my class the MapKit session very much as I laid out the first example of this chapter. Then we moved into one of the coolest things there is with MapKit—the ability to parse, or read live info from, the ether. This feature allows users to "see" the info on their map. I'll explain this to a degree before I present three student final projects.

One of the most intriguing things we can do with MapKit is get real live information from the Internet and configure it in a way that makes the Google Map on the user's iPhone come alive with live information (weather, traffic, geographical phenomena, taxis, planes, and so on). For example, one of the most popular apps for the San Francisco Bay Area is a program demonstrated in *iPhone Cool Projects* (Wolfgang Ante, et al., Apress, 2009) (see Figure 9–28) called "Routesy Bay Area San Francisco Muni and BART," written by Steven Peterson.

Peterson parses all the data from the BART (Bay Area Rapid Transit, `http://www.bart.gov/`) web server that keeps track of how close to schedule all its trains are, where they are, and their speeds. The app parses all this data and makes it useful and relevant to users *at their specific locations* in the San Francisco Bay Area. In Figure 9–29, you will see their app's red icon, and then several iPhone images. The left one shows all the places a user can catch buses and trains. The middle picture uses the same code we used in our example with *core location* to show a user's current location with a blue icon, and where a requested station is. The right image reports to the user the relevant information on the best train given the context, the timing, and so on. The app provides data for the next three trains that will be arriving at the train station nearest the user.

In essence, the MapKit code on the iPhone is, among other things, a *parsing* utility. It retrieves live information from a server that most people don't even know exists, and it puts a stream of data to a novel and useful purpose.

Because of the immediate and practical results that users of Peterson's app, and others like it, can reap, I figured this would be a perfect theme to round out this book. I'll first go over some of my "Parsing with MapKits" lecture notes, and then I will show you several solid final projects created by my students on that basis.

With my students' blessing, the code for their projects can be downloaded from my website, as shown below. This gives you the opportunity to have the code on your Mac while I point out how you can modify it, learn the key features from it, or just put it on your iPad and show the guys at the bar these cool apps. *Enjoy!*

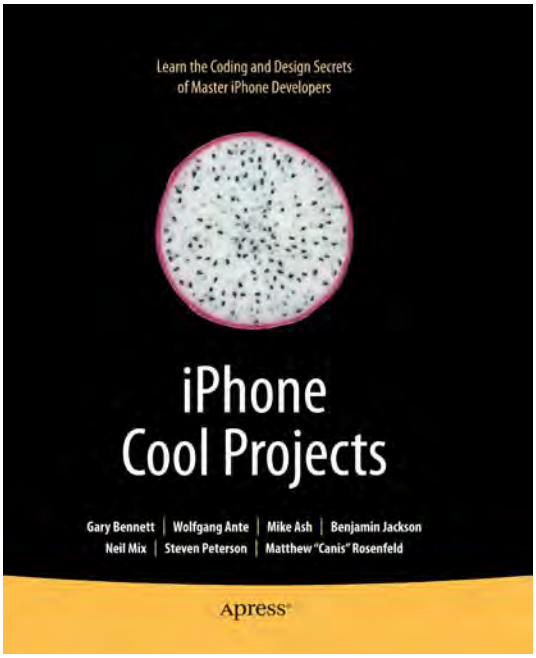


Figure 9–28. *Apress's iPhone Cool Projects.*

Figure 9–29. App icon and examples of three action screens—parsing app: "Routesy Bay Area San Francisco Muni and BART."

The code for these three student Final Projects is located as follows:

- Stephen A. Moraco (Son):
  `http://www.rorylewis.com/xCode/011b_TrafficCam.zip`
- Stephen M. Moraco (Father):
  `http://www.rorylewis.com/xCode/011a_APRSkit.zip`
- Satish Rege: `http://www.rorylewis.com/xCode/011c_MyTraffic.zip`

MapKit Parsing

Remember that this is digging deep into the code at a level that is outside the scope of the book. However, all the instructions that follow can be seen in my students' code, which you are welcome to download. For now, just read along and see if you can follow their pattern of parsing, creating delegate objects, and so forth.

Before we look at their actual apps, consider a hypothetical scenario: Imagine that there is a Grateful Dead Server that broadcasts an update on every Deadhead's geographical location—at least those who allow themselves to be visible on the grid. This hypothetical app allows a (serious) fan of the Grateful Dead to locate all the other Deadheads nearby at any time. These fans can meet and share bootlegs, hang out, and generally relate on a plane that other Grateful Dead disciples can appreciate.

Starting Point: If we were to create such an app, just as in the "Routesy" example, we would allow users to see where they are by bringing up the Attributes Inspector and turning on a Shows User Location switch. We would create a controller called *DeadHeadsView* that creates an instance of a parser we'll call `Gratefuldead`. Then, we would make it set itself as the delegate so it receives the feedback and calls a `getGratefuldead` data method.

Getting Data from Web: As our parser sifts through the XML on the Grateful Dead Server, we would want it to grab `Gratefuldead` element data and create an instance of each *Gratefuldead* object. So, for each instance it creates, it calls back to us with an `addGratefuldead` method. We would need to implement our `Gratefuldead` and *Parser* methods on our `deadHeadsViewcontroller`. We might find that it's easier to think of our `GratefuldeadParser.h` this way:

```
+ (id)GratefuldeadParser; // this creates it
- (void)getGratefuldeadData; // this activates it
```

Add Methods to View Controller: Before adding implementation methods on our `DeadHeadsView` controller, we would need to implement the protocol with `GratefuldeadParser` Delegate and import its header file `#import <GratefuldeadParser.h>`. At this point, we'd be finished with the header, and we'd move to the implementation file.

First, we'd copy the two implementation methods from `GratefuldeadParser.h` and paste these two methods after the `@synthesize` statement:

```
@implementation DeadHeadsViewCOntroller

@synthesize deadView

- (void)getGratefuldeadData:(Gratefuldead *)Gratefuldead;
-(void)parserFinished
```

Test the Parser Feed: To test the Grateful Dead Server, we would see if we could log some messages. Let's separate the two methods, delete the semicolons, add brackets, and then enter "log" as shown:

```
- (void)getGratefuldeadData☹Gratefuldead *)Gratefuldead {
NSLog(@"Hippie Message");
```

```
}

-(void)parserFinished{
NSLog(@"located a Dead Head at %@", Gratefuldead.place");
}
```

Start the Parser Method: Having implemented our delegate methods, we would need to do three things:

1. Code the parser method. Put it into a method we could call `(void)viewWillAppear`. This would get called on by a view controller when its view is about to be displayed. If we were to do it this way, note that we would always want to call in our `- (void)viewWillAppear` method.
2. Create an instance of our parser that we would call *`GratefuldeadParser`*. With this, we'd get `GratefuldeadParser *parser = [GratefuldeadParser gratefuldeadParser]`. We want to make ourselves the delegate, which means that, now, *`GratefuldeadParser parser.delegate = self`*.
3. Two actions in this step: first, tell the parser to get the Gratefuldead data:

```
[parser getGratefuldeadData];
```

   Second, handle its import:

```
#import "GratefuldeadParser.h"
```

   Then, when the `- (void)viewWillAppear` is invoked, it would create an instance of *`GratefuldeadParser`*. As it receives the locations of all the Deadheads, it shows us where they are!

Do you recall how we made sure that the user of the app would appear on the map as a blue dot? I want you to think of the blue dot as just an annotation view. When it is added to the `deadView`, it essentially asks its delegate for the location of itself.

> **NOTE:** If we return anything other than *nil*, then our annotation view, instead of the blue one, will be used and then return that view.

So, looking at this, we return nil when the annotation does not equal the user's current location.

```
- (MKAnnotationView *)deadView:(MKDeadView *)deadView
                 viewForAnnotation:(id <MKAnnotation>)annotation {
MKAnnotationView *view = nil;
return view;
```

But here's the thing; we do *not* want to return nil for our `Gratefuldead` locations. Conversely, we want to do cool things when our annotation is not equal to the `deadView userLocation` property, which itself is an annotation:

```
if(annotation != deadView.userLocation) {

                // THIS IS WHERE WE DO OUR COOL STUFF
               // BECAUSE IT'S A DEADHEAD, NOT THE USER
 }
```

At this point we use the *`dequeueReusableAnnotationViewWithIdentifier`*, delegate method which is available for reuse the instant they are off screen. It has a handy way of storing your annotations in a separate data structure and then automatically adding and removing them from your map as the user's events require it. Note that `dequeueReusableAnnotationViewWithIdentifier` is about getting the reusable annotation view from the map, and it has nothing to do with adding or removing annotations:

```
GratefuldeadAnnotation *eqAnn = (GratefuldeadAnnotation*)annotation;
view = [self.deadView  dequeueReusableAnnotationViewWithIdentifier:@"GratefuldeadLoc"];
       if(nil == view) {
        view = [[[MKPinAnnotationView alloc] initWithAnnotation:eqAnn
                                                 reuseIdentifier:@"GratefuldeadLoc"]
autorelease];
}
```

The annotation view goes and looks in its reuse queue to see if there are any views that can be reused `if (nil == view) { …` If there are none, it returns nil - which means we need to create a new one *`view = [[[MKPinAnnotationView alloc] initWithAnnotation:eqAnn`*.

There are many creative ways to make your annotations appear and be animated with chevrons, bells and whistles, Grateful Dead beads, and so on. You can check out what's out there to make the iCandy portion of your annotations, just as you wish.

In this regard, at this point of writing your code the most important step is to review your code for errors using your NSLog debugger; this will determine whether it connects to a server of your choice. Once that is done, it becomes an issue of parsing the XML. Then, the last step is to shop for iCandy for the annotations.

Three MapKit Final Projects: *CS-201 iPhone Apps, Objective-C*

Following are three apps that draw heavily on parsing information from the Internet. The first two come from a father and son, both named Stephen Moraco, which completely confused me in the lecture hall, and the third is from Satish Rege. They were kind enough to write unedited bios as to why they took the class, and they included exact lecture notes and apps shown in this book, and what they got out of the course.

Biographical Info—Examples 1 & 2

Stephen A. Moraco (Son)

Stephen M. Moraco (Father)

Steve A. (Figure 9–30) is in his senior year in high school. He has been concurrently enrolled at UCCS and has taken courses for dual credit (both high school and college). I, Stephen M. (Figure 9–31), am a professional software engineer working for Agilent Technologies, Inc. Both of us have iPhones and have an interest in learning to write applications for the iPhone. The UCCS course caught our attention as a way we could learn this together. In fact, we really enjoyed Dr. Lewis' CS201 classes, in which we toured the iPhone SDK and practiced writing a number of applications. The discussions in class and then between the two of us as we were driving home always had us excited about things we could do with the iPhone. Our final projects came from these discussions. Dr. Lewis, thank you for offering this course. It provided, in our case, a wonderful time of shared learning. We couldn't have had a more enjoyable time.

Figure 9–30. *Stephen A. Moraco (Son)*

Figure 9–31. *Stephen M. Moraco (Father)*

Final Project—Example 1

Stephen M. Moraco's app is one that is close to his heart. Being an amateur radio hobbyist, he decided to parse Bob Bruning's WB4APR site, where Bob had developed an Automatic Position Reporting System (APRS). Very much like the example I gave in class, locating Deadheads, Stephen, the father, made an app that can locate all the Amateur Radio Operators that are within a user-specified distance from where they are at the time. I will not go over all of Stephen's code because you can download it and go over it carefully. The portions I think you should take note of are as follows: His `APRSmapViewController` header file sets out the road map with 3 IBOutlets, 1 IBAction, and a ViewController:

```
@property (nonatomic, retain) IBOutlet MKMapView *mapView;
@property (nonatomic, retain) APRSwebViewController *webBrowserController;
@property (nonatomic, retain) IBOutlet UISegmentedControl *ctlMapTypeChooser;
@property (nonatomic, retain) IBOutlet UIActivityIndicatorView *aiActivityInd;

-(IBAction)selectMapMode:(id)sender;
```

In the APRSkit_MoracoDadAppDelegate implementation file, he uses the following code to give the user a chance to log in, the results of which are shown in Figure 9–32. The particulars of this step are seen in the - *(void)applicationDidFinishLaunching* method, which also houses the distance (radius) from the user that the system will search for matches:

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

     NSLog(@"MapAPRS_MoracoDadAppDelegate:applicationDidFinishLaunching - ENTRY");
    // Override point for customization after app launch

     [window addSubview:[navigationController view]];
           [window makeKeyAndVisible];

     // preload our applcation defaults
     NSUserDefaults *upSettings = [NSUserDefaults standardUserDefaults];
     NSString *strDefaultCallsign = [upSettings stringForKey:kCallSignKey];
     if(strDefaultCallsign == nil)
     {
          strDefaultCallsign = strEmptyString;
     }
     self.callSign = strDefaultCallsign;
     //[strDefaultCallsign release];

     NSString *strDefaultSitePassword = [upSettings stringForKey:kSitePasswordKey];
     if(strDefaultSitePassword == nil)
     {
          strDefaultSitePassword = strEmptyString;
     }
     self.sitePassword = strDefaultSitePassword;

     NSString *strDefaultDistanceInMiles = [upSettings
stringForKey:kDistanceInMilesKey];
     if(strDefaultDistanceInMiles == nil)
     {
          strDefaultDistanceInMiles = @"30";
     }
     self.distanceInMiles = strDefaultDistanceInMiles;
     //[strDefaultSitePassword release];
     // INCORRECT DECR [upSettings release];
}
```

Figure 9–32. *CS-201 Final Project—Stephen M. Moraco's APRS set-up screen where users enter their Amateur Radio call signs and passwords.*

One of the first things Stephen did when he went to the website was make a list of all the attributes in the *XML* feed. The following list shows what he saw.

Column-1 was the call sign, CALLSIGN

Column-2 was the URL to Message traffic if available

Column-3 was the URL to Weather page if available

Column-4 was the Latitude

Column-5 was the Longitude

Column-6 was the distance from me (in miles)

Column-7 was the DD:HH:MM:SS of last report

To account for this data, he made eight pointers in his `APRSstationParser.m` file. Note that he has an extra one for possible unknown columns.

```
NSString *kCallSignCol          = @"Callsign";
NSString *kMsgURLCol            = @"MsgURL";
NSString *kWxURLCol             = @"WxURL";
NSString *kLatitudeCol          = @"Lat";
NSString *kLongitudeCol         = @"Long";
NSString *kDistanceCol          = @"Distance";
```

```
NSString *kLastReportCol        = @"LastReport";
NSString *kUnknownCol           = @"???";// re didn't recognize column #
```

Then, in the same file, he made `case` statements:

```
    case 1:
              m_strColumnName = kCallSignCol;
              break;
         case 2:
              m_strColumnName = kMsgURLCol;
              break;
         case 3:
              m_strColumnName = kWxURLCol;
              break;
         case 4:
              m_strColumnName = kLatitudeCol;
              break;
         case 5:
              m_strColumnName = kLongitudeCol;
              break;
         case 6:
              m_strColumnName = kDistanceCol;
              break;
         case 7:
              m_strColumnName = kLastReportCol;
              break;
         default:
              m_strColumnName = kUnknownCol;
              break;
```

Also, in the `APRSkit_MoracoDadAppDelegate` implementation file, the *`-(void)recenterMap`* method scans all annotations to determine geographical center and, just as we did in this chapter's exercise, to calculate the region of map to display. Stephen does likewise after his three `if` statements. An image of the pins dropping is shown in Figure 9–33.

```
- (void)recenterMap {
     NSLog(@" - APRSpinViewController:recenterMap - ENTRY");
          NSArray *coordinates = [self.mapView
valueForKeyPath:@"annotations.coordinate"];
     CLLocationCoordinate2D maxCoord = {-90.0f, -180.0f};
     CLLocationCoordinate2D minCoord = {90.0f, 180.0f};
     for(NSValue *value in coordinates) {
          CLLocationCoordinate2D coord = {0.0f, 0.0f};
          [value getValue:&coord];
               if(coord.longitude > maxCoord.longitude) {
                    maxCoord.longitude = coord.longitude;
               }
               if(coord.latitude > maxCoord.latitude) {
                    maxCoord.latitude = coord.latitude;
               }
               if(coord.longitude < minCoord.longitude) {
                    minCoord.longitude = coord.longitude;
               }
               if(coord.latitude < minCoord.latitude) {
                    minCoord.latitude = coord.latitude;
               }
```

```
    }
```

It should be noted that in the `APRSstation` class, Stephen represents the details parsed from the APRS, which sets the location of the pins.

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

@interface APRSstation : NSObject {
    NSString   *m_strCallsign;
    NSDate     *m_dtLastReport;
    NSNumber   *m_nDistanceInMiles;
    NSString   *m_strMsgURL;
    NSString   *m_strWxURL;
    NSString   *m_strTimeSinceLastReport;
    CLLocation *m_locPosition;
    int         m_nInstanceNbr;
}

@property(nonatomic, copy) NSString *callSign;
@property(nonatomic, copy) NSNumber *distanceInMiles;
@property(nonatomic, retain) NSDate *lastReport;
@property(nonatomic, copy) NSString *timeSinceLastReport;
@property(nonatomic, copy) NSString *msgURL;
@property(nonatomic, copy) NSString *wxURL;
@property(nonatomic, retain) CLLocation *position;

@end
```

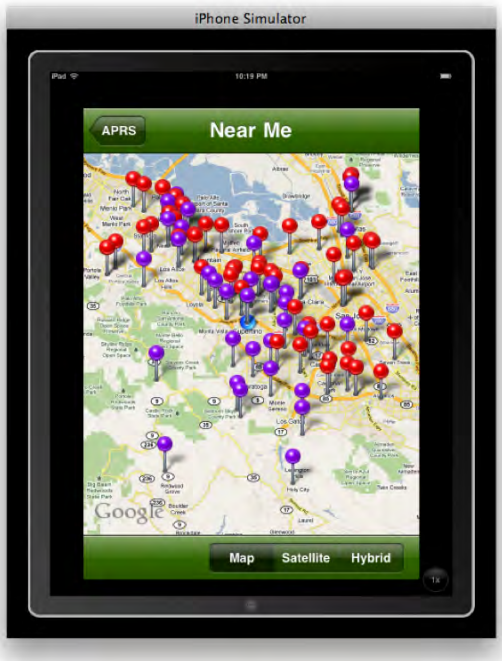


Figure 9–33. *CS-201 Final Project—Stephen M. Moraco's Animated pins drop down within the specified radius of the user's location. Here on the iPad simulator, the pins drop in the surrounding areas of Apple Headquarters.*

Another really cool thing Stephen did was to distinguish between those amateur radio stations that have their own websites and those that do not. For the ones that have web sites, on the annotation view he includes a chevron which, when clicked, yields the web page. See Figures 9–34 and 9–35. This code is seen directly under the switch cases in the `APRSstationParser.m` file.

Figure 9–34. *CS-201 Final Project—Stephen M. Moraco's app provides Annotations to appear when one clicks on a pin and where a linked website is on the APRS server, a blue chevron appears where one may click to go to the amateur radio station's website. In this case, amateur radio station KJ6EXD-7* does *have a website.*

Figure 9–35. *CS-201 Final Project—Stephen M. Moraco's App showing the KJ6EXD-7 website embedded in the iPad.*

In the `APRSmapViewController` implementation file, Stephen includes, among other things, a bare-bones methodology to switch between map, satellite, and hybrid views. An example of this is seen when we show the closest radio station to the user, which, in simulator mode is Apple Headquarters. See Figure 9–36, where the view is in Hybrid mode.

```
-(IBAction)selectMapMode:(id)sender
{
     UISegmentedControl *scChooser = (UISegmentedControl *)sender;
     int nMapStyleIdx = [scChooser selectedSegmentIndex];
     NSLog(@"APRSmapViewController:selectMapMode - New Style=%d",nMapStyleIdx);

     switch (nMapStyleIdx) {
          case 0:
               self.mapView.mapType = MKMapTypeStandard;
               break;
          case 1:
               self.mapView.mapType = MKMapTypeSatellite;
               break;
          case 2:
               self.mapView.mapType = MKMapTypeHybrid;
               break;
          default:
               NSLog(@"APRSmapViewController:selectMapMode - Unknown Selection?!");
               break;
     }
}
```

Figure 9–36. *CS-201 Final Project—Stephen M. Moraco's App showing the closest amateur radio station to Apple Headquarters in the Hybrid map view*

Finally, as a nice finishing touch that I always encourage students to complete, Stephen included a nice About page in the `AboutView` nib. See Figure 9–37.

Figure 9–37. *CS-201 Final Project—Stephen M. Moraco's App showing his "About Page" -- totally cool!*

NOTE: In order to run the code, you will need to have a password and username. You have two options here: 1) Acquire your own or 2) Download any of these 3 apps, which are essentially the same.

```
http://itunes.apple.com/us/app/pocketpacket/id336500866?mt=8
http://itunes.apple.com/us/app/ibcnu/id314134969?mt=8
http://itunes.apple.com/us/app/aprs/id341511796?mt=8
```

Final Project—Example 2

Stephen A. Moraco is a gifted high school student who attended my class. His app parses the National Weather Cam network at `http://www.mhartman-wx.com/wcn/`. This can be seen in the `TrafficCamParser` implementation file *`static NSString *strURL =http://www.mhartman-wx.com/wcn/wcn_db.txt`*. See Figure 9–38.

Figure 9–38. *CS-201 Final Project—Stephen A. Moraco's App starts off with hundreds of pins plummeting from the sky as they fill up a specified area around the user's "current" location at Apple Headquarters.*

He found that he needed to use an adapter to filter out bad meta tags in the `<HEAD></HEAD>` sections. There was so much extraneous matter on the server that it was crashing the code. To take care of this, he had to make rules to replace "^" with `</field><field>`, replace `<br>'s` with blank space, replace `"<font size=\"-1\">(" and ")<br>"` with `</field><field>`, start and end with `<CAM><field> and </field></CAM>`

and remove tags, remove nonbreaking spaces. I've added numbering to help you see the start of each line, as the word wrap confuses me, too!

```
1.  NSString *strNoParaQueryResults = [strQueryResults
    stringByReplacingOccurrencesOfString:@"<font size=\"-1\">("
    withString:@"</field><field>"];
2.  strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@")<br>" withString:@"</field><field>"];
3.  strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"</font>" withString:@""];
4.  strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@" " withString:@""];
5.  strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"></a>" withString:@"></img></a>"];
6.  strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"width=150" withString:@"width=\"150\""];
7.  strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"height=100" withString:@"height=\"100\""];
8.  strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"width=100" withString:@"width=\"100\""];
9.  strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"height=150" withString:@"height=\"150\""];
10. strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"border=0" withString:@"border=\"0\""];
11. strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"\"\"" withString:@"\""];
12. strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@".jpg " withString:@".jpg\" "];
13. strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"&" withString:@"and"];
14. strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"<font size=\"-1\">" withString:@""];
15. strNoParaQueryResults = [strNoParaQueryResults
    stringByReplacingOccurrencesOfString:@"</b<" withString:@"</b><"];
```

The TrafficCamAnnotation.h header files used is straightforward and simple, using the *+ (id)annotationWithCam:(TrafficCam *)Cam*; and - (id)initWithCam:(TrafficCam *)Cam; pointers as described earlier for my hypothetical GratefuldeadParser.h. In this case, *+ (id)annotationWithCam:(TrafficCam *)Cam*; creates parsed file and - (id)initWithCam:(TrafficCam *)Cam; initializes it. The result of all this hard work, taking care of the non-useful code, can be seen in clean annotation. See Figure 9–39.

```
#import <MapKit/MapKit.h>
#import <CoreLocation/CoreLocation.h>

@class TrafficCam;

@interface TrafficCamAnnotation : NSObject <MKAnnotation> {
     CLLocationCoordinate2D Coordinate;
     NSString *Title;
     NSString *Subtitle;
     TrafficCam *Cam;
```

```
}

@property(nonatomic, assign) CLLocationCoordinate2D coordinate;
@property(nonatomic, retain) NSString *title;
@property(nonatomic, retain) NSString *subtitle;
@property(nonatomic, retain) TrafficCam *cam;

+ (id)annotationWithCam:(TrafficCam *)Cam;
- (id)initWithCam:(TrafficCam *)Cam;

@end
```

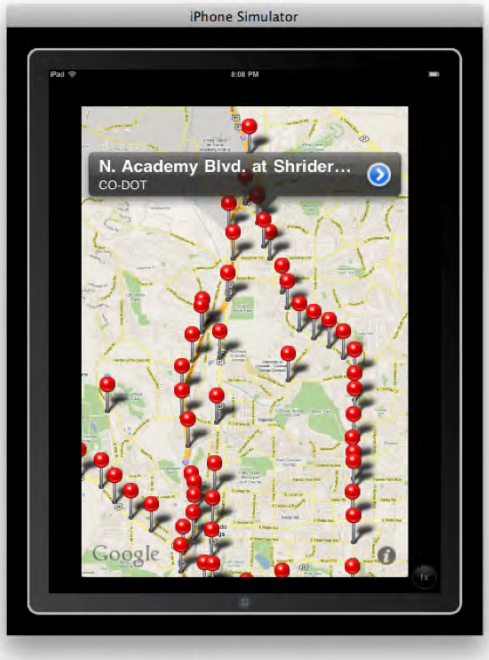


Figure 9–39. *CS-201 Final Project—Stephen A. Moraco's App zoomed into the Colorado Springs area. The annotation of North Academy at Shrider appears because the author clicked on that intersection.*

Stephen also found he could not automatically use the camera video views. Working around this ended up being a non-trivial task in `TrafficCamSettingsViewController.m`. One example was to allow orientations other than the default portrait orientation:

```
BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

He had to arrange this code in order to have beautifully spaced video cam images fitting nicely in the screen, as illustrated in Figure 9–40.

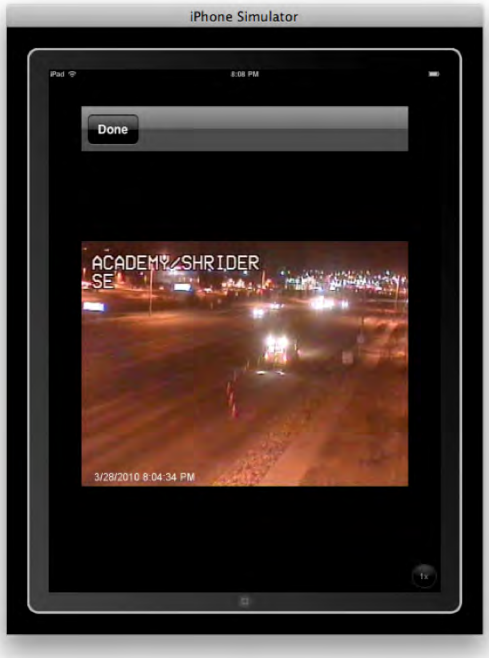


Figure 9–40. *CS-201 Final Project—Stephen A. Moraco's App zoomed into the Colorado Springs area. The annotation of North Academy at Shrider Appears because the author clicked on that intersection.*

Biographical Info—Example 3, Satish Rege

Why do I want to be an iPhone developer? Simple—the iPhone imparts the computing, the communicating, and the multimedia experience of a large computing system in the palm of your hand. It provides rich resources and user interface primitives to express creative capabilities in a synergistic way. These iPhone properties attracted me to want to learn iPhone development tools to express my own ideas. Rory's course was an excellent introduction that covered a multitude of iPhone capabilities and made them easy to master.

Figure 9–41. *Satish Rege*

Final Project—Example 3

Satish (Figure 9–41) was always able to come up with eloquent and simple code for all his homework assignments. When I would grade the weekly assignments, Satish had the knack of always being able to put into 20 lines of code what others would often take three times as much to do the same thing. For his final project, Satish's app allows one to look up ahead at the traffic at intersections to come, and, if there is a traffic jam at one intersection, to recommend another.

At least in theory, that is how it works. Satish saves a lot of heartache by starting at one location he knew would be a tough intersection: I-25Northbound. He focused on controller implementation files and then he rotates back and forth from there depending on your location in Colorado Springs. He has 27 cases for the 27 cameras in Colorado Springs. Simple, elegant, … beautiful.

Figure 9–42 shows the list. Figures 9–43 and 9–44 show two examples of the traffic views.

```
//Choose the camera depending on your co-ordinate

     switch (cameraCordinate) {
          case 1:
               url = [NSURL
URLWithString:@"http://www.springsgov.com/trafficeng/bImage.ASP?camID=17"];     //Camera
- S Academy/ I-25 North
               break;
          case 2:
               url = [NSURL
URLWithString:@"http://www.springsgov.com/trafficeng/bImage.ASP?camID=18"];     //Camera
- HWY 85/87/I-25 N
               break;

>>>>>>
>>>>>>
>>>>>>
          case 26:
               url = [NSURL
URLWithString:@"http://www.springsgov.com/trafficeng/bImage.ASP?camID=33"];     //
Camera - Monument/ I-25 N
               break;
          case 27:
               url = [NSURL
URLWithString:@"http://www.springsgov.com/trafficeng/bImage.ASP?camID=49"];     //Camera
- CountyLine/ I-25 SE
               break;
```

Figure 9–42. *CS-201 Final Project—Rege's app selects the traffic lights closest to the user as he or she is driving down the street.*

Figure 9–43. *CS-201 Final Project—Rege's traffic monitoring app showing the embedded camera view.*

Figure 9–44. *CS-201 Final Project—Satish Rege's app showing another embedded camera view.*

Zoom Out … Seeing the Big Picture

It's important to know where we came from, where we are now, and where we are going next. Not to get too metaphysical on you, but this chapter is a bit of a metaphor for our lives. Where were you 5 years ago? Last year? One day before you bought this book? Where do you intend to be 6 months from now?

That's why this subject is so popular. People love to know where they are! People love to know, love to be shown, how to get from "here" to "there."

You know how men stereotypically refuse to stop and ask for directions? I know *I* do—because I should just *know* where I'm going. When GPS came on the scene, I was impressed. But when Apple included one, by way of 'Maps,' in my first iPhone, I was totally blown away. All of a sudden, I had the ability to consult the oracle *and* maintain my male ego at the same time!

That's power, and that's authority … and that's the revolution you've joined. Now that you have completed this book, and successfully navigated through these exercises—some easier, some tougher—you are well on your way in the world of programming.

As I stated earlier, my goals for you in this chapter were humble. As in any really challenging and worthwhile pursuit, practice makes perfect. If you are exhausted, but

still excited about these ideas and possibilities, then I count that as full success—for you and for myself.

Some of you are maybe thinking about topics that we did not cover in this book: the accelerometer, cameras/videos, peer-to-peer protocol, RSS feeds, mail clients/POP servers, etc. If these areas interest you, my hope is that your mind is already racing off in these new directions. That means you *do* know where you are, and that you know where you *want to go*. Life is good!

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X