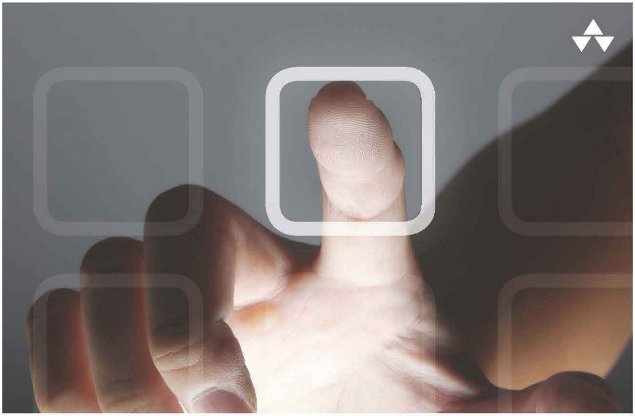


LEARNING

MOBILE APP DEVELOPMENT

A Hands-on Guide to Building Apps with iOS and Android

JAKOB IVERSEN
MICHAEL EIERMAN

Learning Mobile App Development

Learning Mobile App Development

A Hands-on Guide to Building Apps with iOS and Android

Jakob Iversen
Michael Eierman

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2013951436

ISBN-13: 978-0-321-94786-4
ISBN-10: 0-321-94786-X

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing: December 2013

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Barbara Hacha

Indexer
Heather McNeill

Proofreader
Sara Schumacher

Technical Reviewers
Frank McCown
Aileen Pierce
Ray Rischpater
Valerie Shipbaugh

Editorial Assistant
Olivia Basegio

Media Producer
Dan Scherf

Interior Designer
Gary Adair

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

❖

Dedicated to Kim, Katja, Rebecca, and Natasja.

*Dedicated to my wife, Theresa, and daughters,
Lindsey and Kyra.*

❖

Contents

Preface

Welcome to mobile application development!

Developing apps can be fun and is potentially lucrative, but it is also quickly becoming a core skill in the information technology field. Businesses are increasingly looking to mobile apps to enhance their relationships with their customers and improve their internal processes. They need individuals skilled in developing the mobile apps that support these initiatives.

This book is intended to be an introduction to mobile app development. After you successfully complete the book, you will have the basic skills to develop both Android and iPhone/iPad apps. The book takes you from the creation of an app through the publication of the app to its intended audience on both platforms. We (the authors) have been teaching technology for many years at the collegiate level and directly to professionals and strongly believe that the only way to learn a technology is to use it. That is why the book is structured as a series of tutorials that focus on building a complete app on both platforms.

Although the book is an introduction, it does cover many of the unique features of the mobile platforms that make apps a technology offering new capabilities that businesses may use to enrich or augment their operations. The features covered in the book include using the device's capability to determine its location, using hardware sensors and device components in apps, and mapping.

If you have suggestions, bug fixes, corrections, or anything else you'd like to contribute to a future edition, please contact us at jhiversen@gmail.com or michael.eierman@gmail.com. We appreciate any and all feedback that helps make this a better book.

—Jakob Iversen & Michael Eierman, September 2013

What You'll Need

You can begin learning mobile application development with very little investment. However, you will need a few things. The following list covers the basics of what you need for Android programming:

- **Eclipse and the Android SDK**—You can download the SDK from Google (http://developer.android.com/sdk/index.html) as an Android Development Tools (ADT) bundle that includes the Eclipse Integrated Development Environment (IDE), Android development tools, Android SDK tools, Android platform tools, the latest Android SDK, and an emulator. The ADT bundle is for Windows only. If you are going to develop on the Mac, you will have to download Eclipse separately and use the preceding URL to get the various other tools. If you have an existing Eclipse installation, you can use this location to add the Android tools. Appendix A, "Installing Eclipse and Setup for Android Development," has more details on how to install the tools. If your existing Eclipse installation is earlier than the Helios version, we recommend that you update your installation to be perfectly in sync with this book. If you cannot upgrade, you should still be able to work the tutorials. Some of the menu commands may be slightly different and some of the windows may have minor differences, but you should still be able to complete the tutorials.

- **An Android device**—This is not necessary for purely learning, but if you plan to release your apps to the public, you really should test them on at least one device. The more types of devices, the better—Android on different manufacturers' devices can sometimes behave in different manners.
- **Familiarity with Java**—Android apps are programmed using the Java programming language. You should be able to program in Java. At a minimum you should have programming in some object-based programming language such as C# or C++ so that you can more easily pick up Java.

The following list covers the basics of what you need for iPhone/iPad programming:

- **A Mac running Mac OS X Lion (v 10.8 at a minimum)**—iPhone/iPad programming can be done only on a Mac. That Mac should have a fair amount of disk space available and a significant amount of RAM so you don't have to spend as much time waiting for things to compile and execute.
- **Xcode 5**—Xcode is an IDE provided by Apple available from Apple's iOS Dev Center (http://developer.apple.com/ios). Xcode 5 is free, but you can only run the apps you develop on the simulator provided with Xcode. If you want to distribute your apps, you must sign up as a registered developer ($99/year for individuals, $299/year for corporate developers). If you are a teacher at the university level, your university can sign up for the University Program (http://developer.apple.com/support/iphone/university). This will allow you and your students to test apps on actual devices but does not allow public distribution of the apps you create. If you are a student at a university, check with the computer science or information systems department to see if they have signed up for this program.
- **An iOS device**—As with Android, this is not necessary for learning how to program an iOS app, but it is important for testing apps that you want to release to the public. Additionally, some features of iOS programming cannot be tested on the simulator. Appendix B, "Installing Xcode and Registering Physical Devices" has more details both on installing Xcode and the work needed to be able to test your apps on a physical iOS device.
- **Knowledge of Objective-C 2.0**—iOS apps are programmed in Objective-C. Objective-C is a language that extends the C programming language and is organized like the SmallTalk object-oriented programming language. If you have previous experience with Java or C++ it will ease your transition to Objective-C. Appendix C, "Introduction to Objective-C," contains an introduction to Objective-C that will help you with that transition.

What if I Can't Upgrade My Lab Computers?

Xcode 5 requires OSX 10.8. If your existing Macs cannot be upgraded to 10.8 you should still be able to use this book to learn iOS development. In that case use Xcode 4.6. The sample code provided with this book will not work, but you should be able to develop your own working code by working through the tutorials. Some of the menus and windows will be different, but the tutorial will still work.

Your Roadmap to Android/iOS Development

This book is intended as an introduction to mobile development for both Android and iOS. Although the book provides everything you need to know to begin creating apps on both platforms, it is not intended to be a comprehensive work on the subject. The book assumes programming knowledge. At a minimum you should have taken at least one college-level course in the Java or C programming languages. Mobile development introduces issues and concerns not associated with traditional development, but at its core requires the ability to program. Experience with an IDE is a plus. This book will help you learn the Eclipse and Xcode IDEs but if you have some understanding and experience prior to working through this book, it will ease your learning curve.

As a beginner's book, that should be enough to successfully work through the tutorials. However, to truly master Android and iOS development there is no substitute for designing and implementing your own app. For this you will likely need some reference books. Following is a list of books we have found helpful in our app development efforts. Of course, if all else fails—Google it! And then you'll likely end up with the good folks at StackOverflow.com, which has quickly become a trusted source for answers to programming questions.

- *iOS Programming: The Big Nerd Ranch Guide,* by Joe Conway & Aaron Hillegass (Big Nerd Ranch, 2012)
- *Programming iOS 6,* by Matt Neuburg (O'Reilly, 2013)
- *iPad Enterprise Application Development BluePrints: Design and Build Your Own Enterprise Applications for the iPad*, by Steven F. Daniel (Packt Publishing, 2012)
- *Android Wireless Application Development,* by Lauren Darcy & Shane Conder (Addison-Wesley, 2011)
- *Android Wireless Application Development Volume II: Advanced Topics,* by Lauren Darcy & Shane Conder (Addison-Wesley, 2012)

How This Book Is Organized

This book guides you through the development of mobile applications on both Android and iOS. The book focuses on building a single, complete app on both platforms from beginning to publication. The book is meant for the beginner but goes into enough depth that you could move into developing your own apps upon completion of the book. The philosophy embedded in the book’s approach is that the best way to learn to develop is to develop! Although the book begins with Android development, you could choose to begin with iOS without any problem or setback in understanding. However, we do suggest that you read Chapter 2, “App Design Issues and Considerations,” before beginning either platform. After that, you can choose either Chapters 3–8 on Android or Chapters 9–14 on iOS. You could even switch back and forth between the platforms, reading first the introduction to Android in Chapter 3, then the introduction to iOS in Chapter 9, and then continue switching back and forth between the platforms.

Here’s a brief look at the book’s contents:

- **Part I, “Overview of Mobile App Development”**
    - **Chapter 1, “Why Mobile Apps?”**—Mobile apps are a potentially disruptive technology—technology that changes the way business works. This chapter explores the potential impact of mobile technology and discusses how apps can and do change the way organizations do business.
    - **Chapter 2, “App Design Issues and Considerations”**—Mobile technology has different capabilities and limitations than more traditional computing platforms. This chapter discusses many of the design issues associated with app development.
- **Part II, “Developing the Android App”**
    - **Chapter 3, “Using Eclipse for Android Development”**—Eclipse is an open source development environment commonly used for Android development. Chapter 3 shows how to use Eclipse to build a simple “Hello World” app. The chapter is your first hands-on look at app development.
    - **Chapter 4, “Android Navigation and Interface Design”**—The limited amount of “real estate” on a mobile device typically requires multiple screens to build a complete app. This chapter introduces how you program movement between screens in Android. The chapter explores in depth on how a user interface is coded in Android where the number of screen sizes that your app has to accommodate is relatively large.
    - **Chapter 5, “Persistent Data in Android”**—Business runs on data. An app has to be able to make sure important data is preserved. This chapter explores two types of data persistence methods in Android: the persistence of large and complex data in a relational database using SQLite and simple data persistence through SharedPreferences.
    - **Chapter 6, “Lists in Android: Navigation and Information Display”**—Chapter 6 introduces a structure ubiquitous in mobile computing—the list. Lists display data in a scrollable table format and can be used to “drill down” for more information or to open new screens. This chapter explains how to implement a list in an Android app.

  - **Chapter 7, "Maps and Location in Android"**—Displaying information on a map can be a very effective way to communicate information to an app user. This chapter examines implementing Google Maps in an app and also demonstrates how to capture the device's current location.
  - **Chapter 8, "Access to Hardware and Sensors in Android"**—Mobile devices come equipped with a number of hardware features that can enhance an app's functionality. The code required to access and use these features is discussed in this chapter.

- **Part III, "Developing the iOS App"**
  - **Chapter 9, "Using Xcode for iOS Development"**—Chapter 9 begins the book's discussion of iOS. Xcode is the development environment used to develop iPhone and iPad apps. Xcode and iOS development is introduced by guiding you through the implementation of a simple "Hello World" app.
  - **Chapter 10, "iOS Navigation and Interface Design"**—Just as in Android, interface design and navigation between screens are important concepts to master in mobile development. This chapter guides you through the development of a Storyboard for app navigation and demonstrates how to use Xcode's Interface Builder to implement a user interface.
  - **Chapter 11, "Persistent Data in iOS"**—Many of the same data persistence features available in Android are also present in iOS. One primary difference is that the database feature of iOS is implemented through a wrapper kit called Core Data. Core Data enables the updating and querying of an underlying SQLite database.
  - **Chapter 12, "Tables in iOS: Navigation and Information Display"**—Tables in iOS provide the same type of information presentation format as Lists in Android. Tables display data in a scrollable table format and can be used to "drill down" for more information or to open new screens. Chapter 12 describes how to implement this very important mobile computing concept.
  - **Chapter 13, "Maps and Location in iOS"**—Chapter 13 covers the implementation of maps and capturing device location information on an iOS device. It is analogous to the Android chapter on maps and location.
  - **Chapter 14, "Access to Hardware and Sensors in iOS"**—This chapter demonstrates the techniques used to access hardware features of the device. It covers many of the same sensors and hardware features covered in the Android chapters on the topic.

- **Part IV, "Business Issues"**
  - **Chapter 15, "Monetizing Apps"**—One of the reasons many people consider getting into mobile application development is to make money. Both Android and Apple provide a marketplace for apps that has a wide reach. This chapter discusses various approaches to making money from your apps and briefly discusses organization of your app development business.

- **Chapter 16, "Publishing Apps"**—After you have developed an app, you'll likely want to make that app available to its intended audience. This chapter discusses publishing apps on Google Play and the App Store, as well as distribution of corporate apps that are not intended for the public at large.

- **Appendixes**
    - **Appendix A, "Installing Eclipse and Setup for Android Development"**—This appendix provides instruction on installing the Eclipse development environment and how to set up Eclipse specifically for Android development.
    - **Appendix B, "Installing Xcode and Registering Physical Devices"**—This appendix provides instruction on installing iOS development environment, Xcode, and describes how to register iOS devices so that they can be used to test your apps.
    - **Appendix C, "Introduction to Objective-C"**—This appendix provides a brief tutorial on the Objective-C language.

About the Sample Code

The sample code for this book is organized by chapter. Chapters 3 and 9 contain a single "Hello World" app in Android and iOS, respectively. Chapters 4 through 8 build a complete Android contact list app, and Chapters 10 through 14 build the same contact list app in iOS. Each chapter folder contains the code for the completed app up to that point. For example, at the end of Chapter 7 the code includes the code developed for chapters 4, 5, 6, and 7. The exception to this single completed app per folder model is in chapters 7 and 13. These chapters demonstrate several approaches to getting location information on the mobile device. Each technique has a folder with the complete app that demonstrates the technique. If a book chapter requires any image resources, you will find those images in the respective chapter.

Getting the Sample Code

You'll find the source code for this book at https://github.com/LearningMobile/BookApps on the open-source GitHub hosting site. There you find a chapter-by-chapter collection of source code that provides working examples of the material covered in this book.

You can download this book's source code using the git version control system. The Github site includes git clients for both Mac and Windows, as well as for Eclipse. Xcode already includes git support.

Contacting the Authors

If you have any comments or questions about this book, please drop us an e-mail message at jhiversen@gmail.com or michael.eierman@gmail.com.

Acknowledgments

Acknowledgments from Jakob Iversen

Thank you goes out to Mindie Boynton at the Business Success Center in Oshkosh for organizing the training seminars that formed the first basis for the tutorials at the core of the book. Thank you also to all the students taking those seminars for keeping the idea alive and providing feedback and catching mistakes in early versions.

Thanks go as well to everyone who worked with us at Pearson: Trina MacDonald, Chris Zahn, and Olivia Basegio, all of whom worked hard to answer our questions and keep us in line. Thank you also to the technical editors, Valerie Shipbaugh for making sure the material was accessible to the target audience and Aileen Pierce for detailed insights in getting the original material updated for iOS 7.

Thank you to my family and friends for providing support and encouragement during long hours of programming and writing. Especially to my wife, Kim, and daughters, Katja, Rebecca, and Natasja, for picking up the slack around the house.

Acknowledgments from Michael Eierman

A big thank you is owed to my friend and business partner George Sorrells. After I showed him an app that I was fooling around with he said, “We should sell that!” That led to a level of work in Android and iOS that gave me the depth of knowledge required to write this book. I’d also like to thank Mindie Boynton at the Business Success Center in Oshkosh for organizing the training seminars that helped us develop the tutorials that are the basis for this book.

Thanks go as well to the good people at Pearson, Trina MacDonald, Chris Zahn, and Olivia Basegio, who worked so hard to get this book in shape. Thank you also to the technical editors, Valerie Shipbaugh, Ray Rischpater, and Frank McCown, for their help in getting many of the inevitable technical errors and oversights eliminated from the text. I would especially like to single out Frank McCown for in-depth reviews that greatly improved the final product.

Finally, thank you to my friends and family. They supported me by providing feedback on the apps I was developing and encouraged me to continue the effort even when things were most frustrating. My wife, Theresa, and daughters, Lindsey and Kyra, deserve extra special thanks for putting up with my constant work on app development and writing this book.

About the Authors

Jakob Iversen, Ph.D. is Associate Professor of Information Systems, Chair of the Interactive Web Management Program, and Director of Information Technology Services at the University of Wisconsin Oshkosh College of Business. His current research interests include software process improvement, agile software development, e-collaboration, and mobile development. Dr. Iversen teaches and consults on web development, mobile development, technology innovation, information systems management, strategy, and software development processes.

Michael Eierman, Ph.D is a Professor of Information Systems and Chair of the Information Systems Department at the University of Wisconsin Oshkosh College of Business. Dr. Eierman has worked in the information systems field for nearly 30 years as a programmer, analyst, and consultant, but primarily as a teacher. From the very first class taken in college at the suggestion of an advisor, information systems have been his passion. His research has taken many directions over his years as a professor but is currently focused on the impact of collaborative and mobile technology. Dr. Eierman is also co-owner and manager of Ei-Sor Development, LLC—a provider of Android and iOS apps designed for the outdoorsman.

3

Using Eclipse for Android Development

This chapter is an introduction to building a complete Android app. The chapter includes creating a new app project, exploring the components of an Android app, setting up the emulator to run and test apps, and building a variation of the traditional Hello World app. This and the following chapters in this part assume that you have access to Eclipse and that it is set up for Android development. If this is not the case, refer to Appendix A, "Installing Eclipse and Setup for Android Development" before continuing.

Starting a New Project

Eclipse is a powerful, open source, integrated development environment (IDE) that facilitates the creation of desktop, mobile, and web applications. Eclipse is a highly versatile and adaptable tool. Many types of applications and programming languages can be used by adding different "plug-ins." For example, plug-ins are available for a very large number of programming languages as diverse as COBOL, PHP, Java, Ruby, and C++, to name a few. Additionally, plug-ins provide the capability to develop for different platforms, such as Android, Blackberry, and Windows. Many of the tools in the Eclipse IDE will be explained through the act of developing an Android app.

Android is a mobile operating system designed for smartphones and tablets. The operating system is very powerful, enabling access to a diverse set of hardware resources on a smartphone or tablet. Android is provided by Google and is continually updated, improved, and extended. This makes the development of apps for Android smartphones and tablets both exciting and challenging. As with Eclipse, the many features of the Android environment are best explained through the act of developing an app.

Setting Up the Workspace

Eclipse uses the concept of a workspace for organizing projects. Because Eclipse can be used to develop many types of applications, this is very useful. A workspace, in reality, is just a folder on some drive on your computer. The folder contains the application's code and resources, code libraries used by the application (or references to them), and metadata that is used to keep track of environment information for the workspace.

To begin, run Eclipse. The Workspace Launcher dialog window opens, asking which workspace you want to use. The default workspace (or last used) is displayed in the dialog window's text box. Most IDEs are designed with the idea that developers are going to be working on the same machine each time they work on a project. This can cause problems in the education environment where students do not have the ability to work on the same machine and/or store their work on the machine they are currently working on. If you are using your own machine, you can skip to the next section; your workspace was created when you installed Eclipse and is ready to go. However, if you are working in an environment where you cannot use the same machine each time, you need to set up a workspace on either a flash drive or on a network drive. Determine which of these options is best for your situation and perform the following steps:

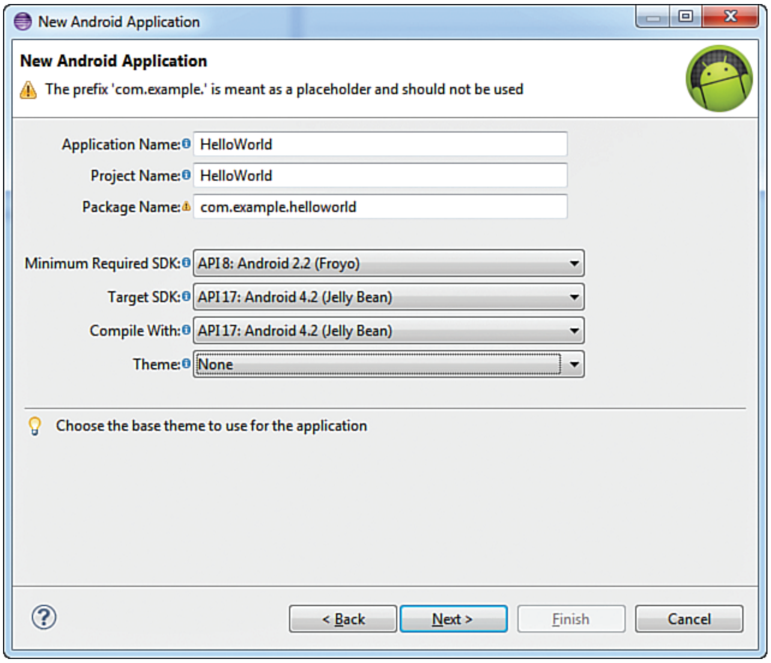
1. Create a folder in your selected location named **workspace**.
2. Go back to the Workspace Launcher and browse to your new folder. Click OK.

   Often in a situation where you change the workspace to a location not on the machine that Eclipse is installed on, Eclipse will not be able to find the Android SDK. If it cannot find the SDK, a dialog window opens. If this happens, you will have to tell Eclipse where the files are located by performing the next steps.
3. Click Open Preferences on the dialog window and browse to the sdk folder. This is usually located in the .android folder. Click Apply.

   The available Android versions should be displayed in the window.
4. Click OK to close the dialog window. Your workspace is now ready to begin Android development.

Creating the Project

The traditional beginning tutorial for many different languages and development platforms is "Hello World." Your first Android app will be a slightly modified "Hello World" app. In Eclipse, all Android apps are created within a project. To create your first app, you will have to create your first project. Creating a new project requires stepping through a series of windows and making choices to configure your app. To get started, from Eclipse's main menu choose File > New > Android Application Project. You should see the New Android Application dialog window, as shown in Figure 3.1.


New Android Application

New Android Application

The prefix 'com.example.' is meant as a placeholder and should not be used

Application Name: HelloWorld

Project Name: HelloWorld

Package Name: com.example.helloworld

Minimum Required SDK: API 8: Android 2.2 (Froyo)

Target SDK: API 17: Android 4.2 (Jelly Bean)

Compile With: API 17: Android 4.2 (Jelly Bean)

Theme: None

Choose the base theme to use for the application

Figure 3.1 Initial new Android application window configured for "Hello World."

Fill out the screen as shown. The application name is displayed on the phone's screen as the name of the app. You can use spaces if you want. As you type the name, the project name and package name will be completed. There are no spaces allowed in these items. The wizard will remove them as you type. Don't put them back in either of these fields. The package name is important. For this initial project you don't need to change the default. However, if you are building an app for sale, in place of "example" you should put your company name. This identifier will be used in the Play Store to link your apps to the services they use and connect all your apps.

Next, click the Minimum Required SDK drop-down. A list of potential Android SDKs are listed. SDK stands for Software Development Kit, and it is a set of tools and code libraries used to write software for a specific platform. Each release of the Android OS is associated with an SDK so that programmers can write code for that platform. An application programming interface (API) is a set of routines that allow a program (app) to access the resources of the operating system to provide functionality to the user. The minimum required SDK determines what phones and other Android devices will be able to install your app. (Phones and tablets using Android operating systems earlier than this selection will not even see your app in the Play Store.) This selection will also determine the features you can program into your app. The recommended minimum is the default: *Froyo API 8*. An app that has this minimum will be accessible to more than 90% of the devices "in the wild."

The Target SDK should usually be set to the latest version of the Android operating system. At the writing of this book, that version is the Jelly Bean (API 17). After you release an app, you should periodically update these values and recompile your app as new versions of Android are released. At times, new versions of the operating system can affect the performance of your app, so it is best to keep the app up to date. The Compile With target should also be the latest SDK.

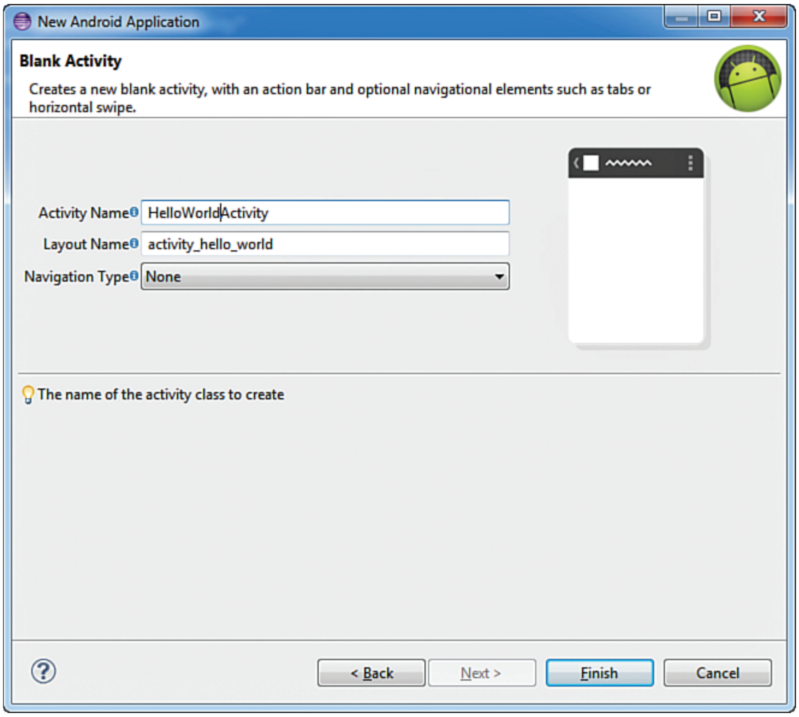
Themes are a useful way to ensure a consistent look for your app. However, because this is an introduction you will not be using them in this book. Click the drop-down and select None as your theme.

After you have verified that your selections match those in Figure 3.1, click the Next button and the Configure Project window will be displayed. You should accept the defaults on this screen. After you learn the app creation process, you may want to modify the default settings to better match your requirements. However, by using the defaults, some work is done for you that can easily be changed later as needed. Click the Next button to display the Configure Launcher Icon window.

The Configure Launcher Icon window allows you to associate an icon with your app that will be displayed on the phone's screen along with the app name. Notice the different sizes of the icons. If you are providing an icon for your app, you will have to supply several sizes of the same picture. This is because Android apps can run on any Android device that meets the app's SDK requirements. However, these devices can have different screen resolutions and different screen sizes. By supplying different icon sizes, the app will pick the one that best matches the device it is running on. This helps ensure that your app will show up as you design it, regardless of the characteristics of the device it is running on. Suggested sizes for app icons are 32×32, 48×48, 72×72, 96×96, and 144×144 pixels for low to extra high density screens. Accept the default icon for this app by clicking the Next button.

The Create Activity window is the next step in configuring your project. An Activity is a core component of any Android application. Activities are typically associated with a visible screen. Most of the core functionality of an app is provided by an activity and its associated screen (called a *layout*). Click among the different activity options. Notice that when you have selected some of them, the Next button is disabled. The choices are limited by your choice of minimum and target SDK. Eclipse won't let you use features that will not work on the devices you targeted. In this case, because you selected API 8 as the minimum SDK that your app would be allowed to run on, some activity types are not available, even though they are available in the target SDK you selected.

From the list of possible activities, choose Blank Activity and click the Next button. The Blank Activity window is displayed (Figure 3.2). This allows us to configure the first Activity in our app. With this screen we can change the name of the activities we create. In the Activity Name text box, delete MainActivity and type HelloWorldActivity. Notice below Activity Name is Layout Name. As you typed in the activity name, the text in this box changed to reflect the text you entered. A layout is an XML file that provides the user interface for the activity. Layouts are discussed in detail later. For now, just remember that every activity has an associated layout file.


New Android Application

Blank Activity

Creates a new blank activity, with an action bar and optional navigational elements such as tabs or horizontal swipe.

Activity Name: HelloWorldActivity

Layout Name: activity_hello_world

Navigation Type: None

The name of the activity class to create

Figure 3.2 Blank Activity window with default selections.

The final item on this page is Navigation Type. Select it and click among the options. Notice that just like the Create Activity window, you are not allowed to use some navigation types. Again this is based on the SDK choices you made earlier. Select None as your Navigation Type and click Finish. Your app project is created! Depending on the capability of your computer, it may take some time to create the project. When Eclipse has finished creating your project, your Eclipse environment should look like Figure 3.3.

Components of the IDE

Many of the items in the IDE will be explained as needed. For now you will examine just a few. The top center section is the *Editor*. Much of the development work is done here, including the UI design and writing code. It should currently be displaying the layout for the HelloWorldActivity in Graphical Layout mode. You can switch between graphical layout and the XML code that generates the layout with the tabs below the layout. One tab will always say Graphical Layout. The other will be the filename of the layout. In this case it is activity_hello-world.xml.

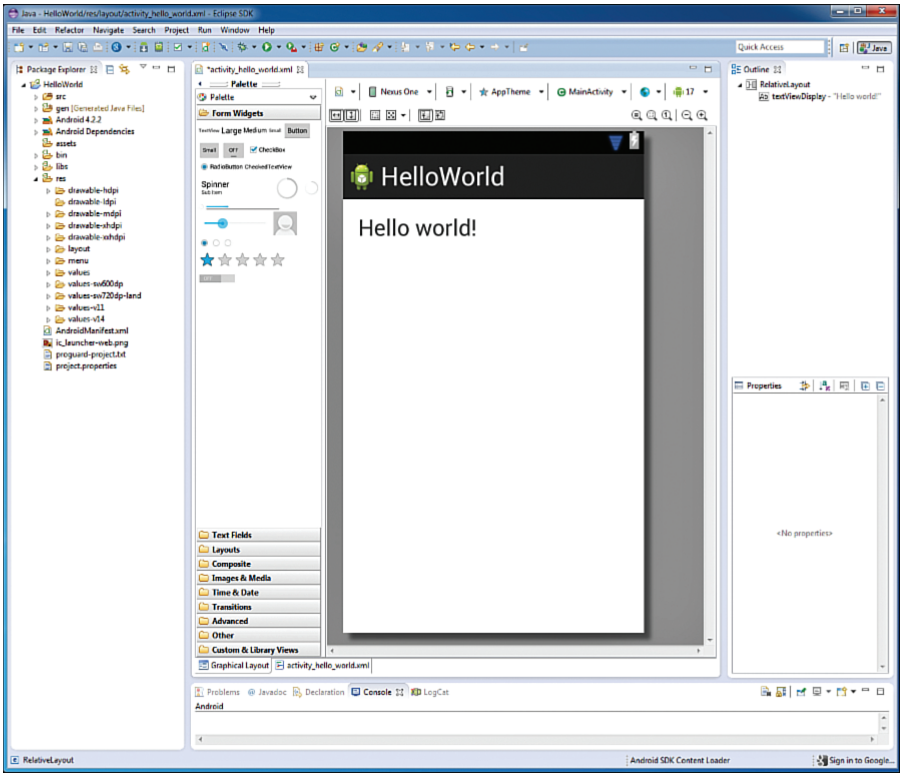


Figure 3.3 Eclipse with the newly created Hello World project.

The left side of the IDE shows the Package Explorer. The Package Explorer displays the structure of the Android app and is used to move between different components of the app. Many of these items will be generated for you, and many others you will work with as you create your app. The src folder will contain all the Java code files for the app. Each file typically represents one class. Double-click the folder and its subfolders until you see HelloWorldActivity.java. This is where the code to create the activity's functionality is written. Double-click the HelloWorld.java file. The file contents are displayed in the editor with some Java code listed. This code is explained later.

Next, look for the res folder in the Package Explorer. This folder contains a number of folders that all contain a different kind of resource file needed for your Android app. One very important note about resource files: There are no capital letters allowed in the file names! Double-click through the drawable-xxx folders. The drawable folders are for images. Android uses Portable Network Graphics (PNG) files for its images. Notice the ic_launcher.png file is in all the drawable folders except the drawable-lhdp folder. Each one of these files is the launcher icon in a different size to match the size recommendations for different screen resolutions.

The lhdp folder does not contain an icon because no Android devices with low resolution are available with an API 8 or higher. When your app is installed on a device, Android automatically uses the one appropriate for the device it is installed in by selecting it from the correct folder.

Next is the layout folder. This folder holds all the layouts for the user interface of your app. The menu folder holds the menu items to be displayed in your app when a user clicks the device's menu button. Menu functionality is not required for an app, and this book will not work with them.

The final set of folders is that of the values folders. Double-click the values folder. Three XML files will be displayed: dimens.xml, strings.xml, and styles.xml. The values files hold configuration data for an Android app. Android uses this information to limit the hard-coding of potentially changeable data. For example, the dimens.xml file could hold a value for screen title size that could be reused on each layout in the app. If you later decide that you want the screen title size to be different, you only have to change the value in the dimens.xml file and it automatically applies the new size to all titles that use that dimension. The values folders with a dash and number or other information are for values to be used for specific versions of the Android operating system. This enables the developer to take advantage of different OS capabilities within the same app. Some common values files are described below:

- dimens.xml—Values for the display size of items in a layout.
- color.xml—Values for the displayed color of item in a layout.
- strings.xml—Values for text.
- array.xml—Defines string arrays and the values in those arrays.
- ids.xml—IDs that cannot be reused by items in layouts.

The Android Manifest

The final and very important item in the Package Explorer that we will examine is the AndroidManifest.xml file. The manifest file is not in a folder but is listed as one of the folder independent files following all the folders in the project. Double-click this file. The Manifest editor will be displayed in the editor. The manifest is used to configure the whole app and tell the device it is installed on what it can and should be able to do. There are multiple tabs (at the bottom of the editor) associated with the manifest. These are used to configure different aspects of your app. The Manifest tab (which is the initial tab open) includes several important elements. First, note the Version Code and Version Name elements. Version code is an integer value. It is used to indicate that there is a new version of the app available. Increasing the value enables the Play Store to notify users of the app that a new version is available. It also controls the install of the upgrade so that no user data is lost during an upgrade. The Version Name is the displayed version of your app. Beyond that it is nonfunctioning. However, it is good practice to have a consistent approach to changing this so that you know what version of the app is at issue when communicating with users about their problems with the app. Click Uses Sdk. The current selections for minimum and target SDK are displayed. These can be modified here. Next click the Application tab at the bottom of the editor. This tab provides the capability to configure specific operational and display elements of the app. Finally, click the AndroidManifest.xml tab. The selections made in the editors generate code that is displayed here.

Interpreting the XML

Although the tabs in the Manifest editor can be used to create a basic configuration of the manifest, the ability to read and manipulate XML is a critical skill for the Android app developer. Modifying a manifest to allow your app to do more advanced behaviors is common, and most online help on doing so, either from the Android Developer site or developer forums, is provided in XML. To get started, take a look at the manifest components in the AndroidManifest.xml file (Listing 3.1).

Listing 3.1 **Manifest XML**

```
<?xml version="1.0" encoding="utf-8"?>
                                                                                             //1
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.helloworld"
    android:versionCode="1"
    android:versionName="1.0" >
                                                                                             //2
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />
                                                                                             //3
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
                                                                                             //4
        <activity
            android:name="com.example.helloworld.HelloWorldActivity"
            android:label="@string/app_name" >
                                                                                             //5
            <intent-filter>
                                                                                             //6
                <action android:name="android.intent.action.MAIN" />
                                                                                             //7
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

The manifest contains a number of XML elements. Those elements and their attributes define basic operational aspects of your app. Refer to the numbers in Listing 3.1 to see the complete code associated with each element explanation below.

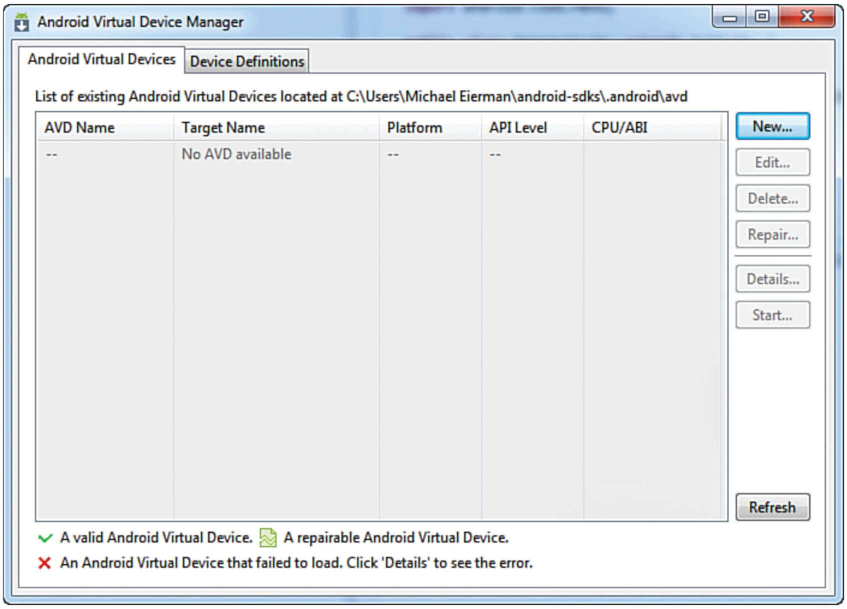
1. The `<manifest>` component is the root element. The attributes associated with this element define the application package, version code, and version name (as well as others).
2. The `<uses-sdk>` element and its attributes define the minimum and target SDKs for the app.
3. The `<application>` element has both attributes and child elements that configure how the app works. Application attributes in this manifest define the app icon, theme, and name. Each activity in an app must have an entry in the `<application>` element. In our manifest there is one activity: the one created when we created the project. Its attributes identify the Java class file for the activity and the display name of the activity. Currently, that name is the same as the app's name.
4. The `<activity>` element tells the operating system that an activity has permission to run in your application. All activities used in an app must be defined in the manifest. If they are not, the app will crash when the user navigates to that activity. In this element the Java source file for the activity and the activity's title are identified.
5. A child element of the `<activity>` element, the `<intent-filter>` element, defines what the Android OS should do with this activity. Not all activities will have an intent-filter. Specifically, activities that you want users to launch when they are using the app do not need intent-filters. However, for this app you want this activity to be displayed when the user runs it.
6. Therefore, the `<action>` tag identifies the activity as the main or first activity to run.
7. The `<category>` tag tells the OS to use the app launcher to start this activity.

Configuring the Emulator

Now that you have some understanding of the development environment, you are almost ready to start creating the app. Don't worry. Future projects will take less time to set up. You could start coding at this point, but until you tell Eclipse how to execute the app, you will not be able to see your results. Therefore, the next step will be to set up the test environment.

Android apps may be tested on either the emulator provided by the Eclipse IDE or on an Android device. The emulator is a program that simulates an Android device. If you choose to test on the emulator, you should also test on several varieties of real devices before you publish your app. Real devices often perform differently than the emulator. If you do not test on a real device, you will likely have many unhappy users.

To set up the emulator, we first must set up an Android Virtual Device (AVD). An AVD is a software replication of one or more types of Android devices. Multiple AVDs with different characteristics may be set up for testing. To set up an AVD we use the AVD Manager. From the main menu select Window > Android Device Manager to display the Android Virtual Device Manager (Figure 3.4).



List of existing Android Virtual Devices located at C:\Users\Michael Eierman\android-sdks\.android\avd

AVD Name	Target Name	Platform	API Level	CPU/ABI
--	No AVD available	--	--	

✓ A valid Android Virtual Device. A repairable Android Virtual Device.
✗ An Android Virtual Device that failed to load. Click 'Details' to see the error.

Figure 3.4 Android Device Manager in initial state.

The manager opens with the Virtual Devices tab displayed. Click the Device Definitions tab. This displays all the device configurations your system knows about. Scroll through these to see how many devices your app could run on. Press the Device Definitions tab and then click the New button. The Create New Android Virtual Device (AVD) window is displayed. Complete the device definition as follows, changing only these options:

AVD Name: MyTestDevice

Device: 3.2 QVGA (ADP2) (320 x 480: mdpi)

Target: Android 4 2.2 – API Level 17

SD Card: Size 1024 MiB

When you click the Device drop-down, a large number of devices are available. Scroll down the list to find the device: 3.2 QVGA (ADP2) (320 x 480: mdpi) and select it. After you've selected the device, choose ARM from the CPU/ABI drop-down. Most devices have an SD card. However, if you want to test your app for those that do not, don't change anything for the SD Card option. Click OK. The new AVD will be displayed in the Android Virtual Devices tab. Click the new AVD named MyTestDevice that now shows in the existing AVD list, and the buttons on the right of the AVD Manager will be enabled. Click the Start button and the Launch Options window will be displayed. Leave all the defaults. Checking the Scale Display to Real Size box will show the virtual device at the size of the real device. However, this can be hard to use during initial development. Checking the Wipe User Data box will wipe out any data created in a previous session. It is useful to leave the data intact so that you will not have to reenter data every time you want to test some aspect of the app.

Note

I like to start my development with one of the smaller devices because I find it easier to scale up when developing the user interface than to scale down. Also, I like to pick a lower API for the device for similar reasons. Later, you can create different AVDs to test different device configurations.

Click Launch. The Start Android Emulator window will display and start loading the AVD. When it is done, the virtual device displays (Figure 3.5) and begins further loading. The speed at which the device loads depends greatly on your computer. At times it can be quite slow. If I am testing with the emulator, my first task when beginning any development session is to start the virtual device so that it is ready when I am. After the AVD is displayed, you can close the Start Android Emulator and AVD Manager windows. The AVD will remain running.

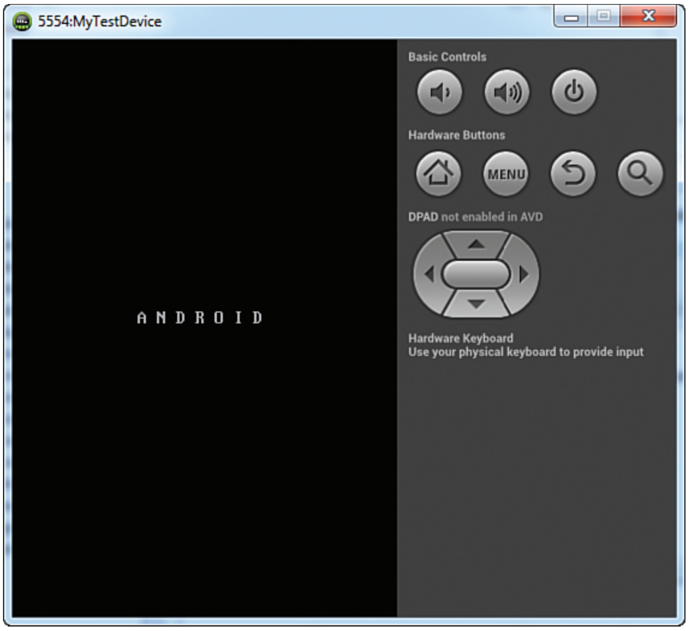


Figure 3.5 Android Emulator at initial launch.

Setting Up Run Configurations

The final step in setting up the test environment is to tell our app to use this newly created AVD. To do this you need to set up a Run Configurations.

1. From the main menu select Run > Run Configurations. The Run Configurations window is displayed.
2. Click Android Application in the left side of the screen. Then click the New button, which is the leftmost button above the text box that says Type Filter Text. The window changes, showing configuration options. Change the name to HelloWorldRunConfig.
3. Use the Browse button to select your HelloWorld project. Click the Launch Default Activity option button.
4. Click the Target tab. Click the box next to MyTestDevice. When you start testing on a real device, you will need to click the option button next to Always Prompt to Pick Device. This displays a device selection window where you can pick the device you want to test on.
5. Click the Apply button and then the Close button. You are ready to begin coding your app!

Coding the Interface

As mentioned earlier, the interface for any Android app is created through the use of a layout file. A layout file is an XML file that contains the XML used to create the objects and controls that the user can interact with. The first step in coding the HelloWorld app is to modify the layout so that it has some controls that the user can interact with. Your modifications will be simple. You will make the app take a name entered by the user and display Hello [entered name] after a button click.

Double-click the activity_hello_world.xml file in the layout folder of the Package Explorer to begin work coding the interface. If it is already open in the editor, click the activity_hello_world.xml tab at the top of the editor (Figure 3.6, #1). If the Graphical Layout is displayed, click the activity_hello_world.xml tab at the bottom of the editor (Figure 3.6, #2). The XML code that creates the user interface is displayed with two *elements* in it. The root element is a RelativeLayout. Because Android devices have so many screen sizes and resolutions, it is often best to design the UI components as relative to one another rather than designing them as a fixed position. Because the RelativeLayout is the root, it encompasses the whole screen. You must have only one layout root in an Android layout file. All other items are children of this root element.

Examine the attributes of the RelativeLayout element (Listing 3.2). A closer look at the attributes reveals a certain structure. Attributes have the format *`library:attribute name = "attribute value"`*. First, all the attributes in the listing start with `android:` This indicates that the attribute is associated with Android SDK library and that is where the compiler should look for information on what to do. Other libraries are available from third parties. Adding other libraries will be covered later in this text. The attribute name and values differ based on the element to which they are applied.

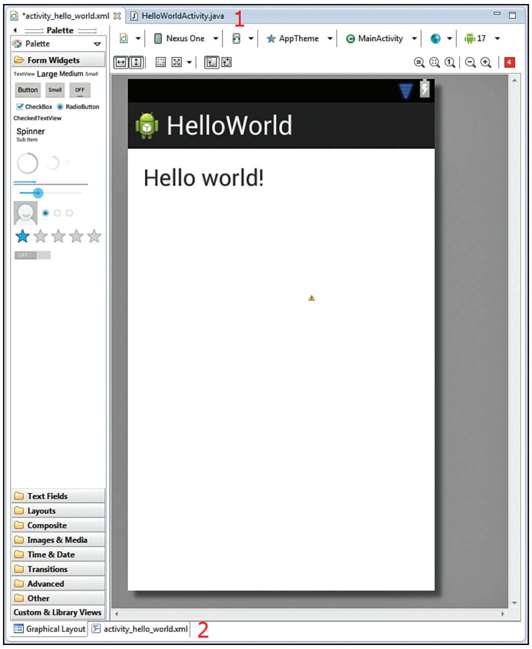


Figure 3.6 Editor and layout tabs.

Listing 3.2 **Layout XML**

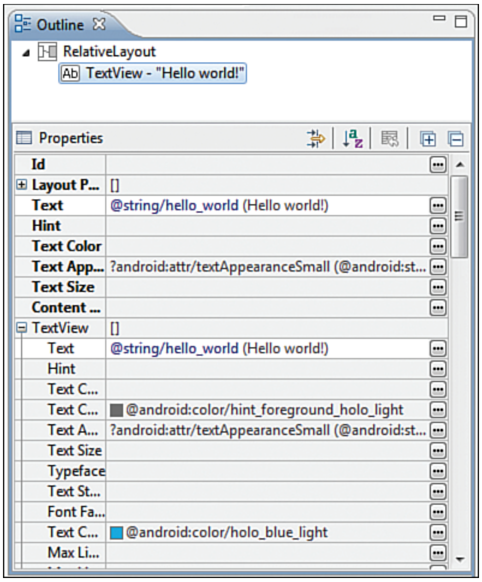
```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
                                                                            //1
    android:layout_width="match_parent"
    android:layout_height="match_parent"
                                                                            //2
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >
                                                                            //3
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world"/>

</RelativeLayout>
```

1. The first attributes of interest in the RelativeLayout are `android:layout_width="match_parent"` and `android:layout_height="match_parent"`. These attributes define the size of the element. In this case the value `"match_parent"` indicates that the layout should be the height and width of the device screen. If a child element of RelativeLayout has this value for either `layout_height` or `layout_width`, it will fill up as much of the RelativeLayout as it can.
2. The next few attributes: `paddingRight`, `paddingLeft`, `paddingBottom`, and `paddingTop`, all tell Android that it should not fill the entire screen with the RelativeLayout. Instead, there should be blank space between the edge of the screen and the edge of the layout. The amount of space is dictated by the value. The values in these attributes are your first introduction to the use of the XML files in the values folder. To refer to values from these XML files, Android also has a specific structure. That structure is *`"@xml_file_name/value_name"`*. All values are enclosed in quotation marks. The value for the attribute `android:paddingBottom` is `"@dimen/activity_vertical_margin"`. This tells Android it should use the value named `activity_vertical_margin` from the dimens.xml file. Double-click the dimens.xml file in the values folder in the Package Explorer. The file will open to the Resources tab. Click the dimens.xml tab at the bottom of the editor. This displays the XML used to define the dimensions. The `<dimen>` tag is used to define each dimension. Each dimension has a name attribute, a value, and then a closing tag that looks like this: `</dimen>`. The value between the beginning tag and the closing tag is the value that Android uses as the size of the padding.

   Valid dimensions for Android include `px` (pixels), `in` (inches), `mm` (millimeters), `pt` (points), `dp`/`dip` (density-independent pixels), and `sp` (scale-independent pixels). It is generally recommended that `dp` be used for most dimensions and `sp` be used for specifying font sizes. These two units of measure are relative to screen density. They help keep your UI consistent among different devices. The reason that the `sp` unit is recommended for fonts is because it also scales to the user's preference in font size.
3. The only child element of the RelativeLayout, and thus the only item on the screen, is a TextView. TextView is Android's version of a label. It is primarily used to display text. This element currently has only three attributes. The two size attributes differ from the RelativeLayout in that they have the value `"wrap_content"`. This tells Android to size the TextView to the size of the text displayed in it. The only other attribute tells Android what text to display. In this case it gets the text from the strings.xml file in the values folder. Open the strings.xml file and examine the XML to find the "hello_world" item. Note that its value is "Hello World!", exactly what is displayed in the running app and on the Graphical Layout view of the activity_hello_world.xml file. The TextView does not have any attributes describing its positioning, so Android puts it in the first available position, which is the very top-left position in the RelativeLayout.

Switch back to the Graphical Layout view of the activity_hello_world.xml file. At the left of the layout is a panel titled Palette. Palette contains a set of folders with different components (called widgets) that can be used to design a user interface. If it is not open, click on the Form Widgets folder in the Palette. Form Widgets contains a set of widgets for designing the user interaction with your app. Hover your mouse over each of the icons to see what type of control the widget implements. Notice that some controls have multiple versions that enable you to pick the size that you want for your interface.

A TextView that displays “Hello World” is already on the layout. This is used to display your app’s message. However, the size of the text needs to be bigger. To the right of the editor should be a panel with a tab with the label Outline (Figure 3.7). If this is not present, click Window > Show View > Outline to display it. The top of the tab should show the structure of the layout. It should have RelativeLayout as its root and textView1 indented below it. The TextView should be displaying “Hello World” after it. As widgets are added to the layout, they are displayed in the structure. This is very useful because sometimes controls are added to the layout that get lost (not visible) in the Graphical Layout. However, if they are in the layout they will be displayed in the structure.



RelativeLayout
TextView - "Hello world!"

Id	
Layout P...	[]
Text	@string/hello_world (Hello world!)
Hint	
Text Color	
Text App...	?android:attr/textAppearanceSmall (@android:st...
Text Size	
Content ...	
TextView	[]
Text	@string/hello_world (Hello world!)
Hint	
Text C...	
Text C...	■ @android:color/hint_foreground_holo_light
Text A...	?android:attr/textAppearanceSmall (@android:st...
Text Size	
Typeface	
Text St...	
Font Fa...	
Text C...	🟦 @android:color/holo_blue_light
Max Li...	

Figure 3.7 Layout Outline and Properties panels.

Below the structure is the Properties window. If you haven’t clicked anything in the Graphical Layout, that window will be displaying <No Properties>. Click “Hello World” in the Graphical Layout. The Properties window should populate with all the attributes that can be set for a TextView widget. Locate and click the **...** button next to the bold attribute Text Size. The Resource Chooser window is displayed. Two dimensions created when the project was created are listed (the padding margins). Click the New Dimension button at the bottom of the Resource Chooser. In the window that opens, enter **message_text_size** as the dimension name and **24sp** as the value. Click OK until you have closed these two windows. The size of Hello World! should be increased. Open the dimen.xml file and switch to the XML view to see the dimension you created. Close this file and click back to the activity_hello_world.xml tab. Switch from Graphical Layout to the XML view and examine the XML changes to the TextView element. Switch back to the Graphical Layout.

Note

The values files are used to hold values that are going to be reused in your app. Unfortunately, the only way to know what values are available is to open the file and inspect its contents for the value you'd like to use. We recommend that when you add values, you name them very clearly and limit the number of values you use to keep it somewhat manageable. Naming clearly is very important because Eclipse's code completion capability will list the value names but not their actual value.

Locate the Small TextView widget just below the Form Widgets folder label. Click and drag it to the layout, position it as in Figure 3.8, and drop it. Notice the green arrows pointing to the left side of the layout and to the Hello World! TextView. These arrows show what object the widget is relative to for positioning purposes. Click the XML view (Listing 3.3). A number of changes have been made to the XML.

Listing 3.3 **Layout XML with TextView Added**

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".HelloWorldActivity" >

    <TextView
                                                                       //1
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world"
        android:textSize="@dimen/message_text_size" />

    <TextView
        android:id="@+id/textView1"                                    //2
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView2"                      //3
        android:layout_below="@+id/textView2"
        android:layout_marginLeft="19dp"                              //4
        android:layout_marginTop="36dp"
        android:text="Name:"                                          //5
        android:textAppearance="?android:attr/textAppearanceSmall" />        //6

</RelativeLayout>
```

1. The Hello World! TextView now has an attribute `android:id="@+id/textView2"`. To correctly relatively position the new TextView, Android needed a way to reference it so it added the ID. The `+id` tells Android to create the ID for the widget. IDs can be defined in the ids.xml values file. However, to use these IDs for widgets, you need to define them prior to use, and they cannot be reused. Using `+id` enables you to tell Android to create an ID for the widget as you need it. `textView2` is not a very useful ID. It does not describe what the TextView is used for, so change the ID to `textViewDisplay`.
2. The new TextView also has a `+id`. However, it is different from the first one. `+id`s may be reused in different layouts but cannot be reused within the same layout! Next come the widget size attributes. All items in a layout must contain these attributes.
3. As the arrows on the Graphical Layout showed, this widget is positioned relative to the Hello World! TextView. The layout attributes are the XML used to do the relative positioning. `alignLeft` tells Android to align this widget's left edge with the referenced widget's left edge. `alignBelow` tells Android to position the widget below the referenced widget.
4. The margin attributes `layout_marginLeft` and `layout_marginTop` tell Android how much space to put between the widget and the referenced widget. Change the left margin to 20dp and the top margin to 55dp. You will often have to tweak these values to get the layout to look exactly the way you want it to.
5. The `android:text` attribute indicates what text should be displayed. This attribute is underlined with a yellow triangle on the left edge. This is a warning. Hover over or click the yellow triangle. The warning is displayed. The value `"Small Text"` is a hard-coded value. Android wants all values to be referenced from a value's XML file. This is for ease of maintenance. You can change a string value used multiple times just once in the strings.xml file, and the changes will be made throughout your app. Also, by substituting a different string's.xml file, you can adapt your app to different languages more easily. To simplify this example, leave the string hard-coded but change it to meet your needs. Delete `"Small Text"` and replace it with `"Name:"`.
6. The final attribute in the new TextView is `textAppearance`. The value for this attribute references the Android attr.xml file and is used in place of the `textSize` attribute. The attr.xml file is a file supplied by the Android SDK. Switch back to the Graphical Layout view. The TextView you added should now be displaying Name:.

UI Design—Android Versus iOS

UI design in Android is done through relative positioning of the controls that make up the interface. However, in iPhone and iPad, absolute positioning is used. Absolute position holds the control to a fixed position on the screen. The use of absolute position makes the design of the UI easier. Unlike in Android, when you move a control it has no effect on other controls in the UI. Often in Android, moving one control changes the whole design. This can be frustrating! When moving or deleting a control in an Android layout, especially if you do this in the XML, be sure to check the impact of the change in the Graphical Layout.

Interface design is not without its challenges in iOS. Devices that run iOS have a fixed screen size, which is controlled by Apple. This enables the use of absolute positioning because all device screen sizes are known by the developer. However, this means that the UI has to be created multiple times for each device that you want your app to run on. These different screens all run on the same code, so during design, the developer must be sure to be perfectly consistent among the different screens needed.

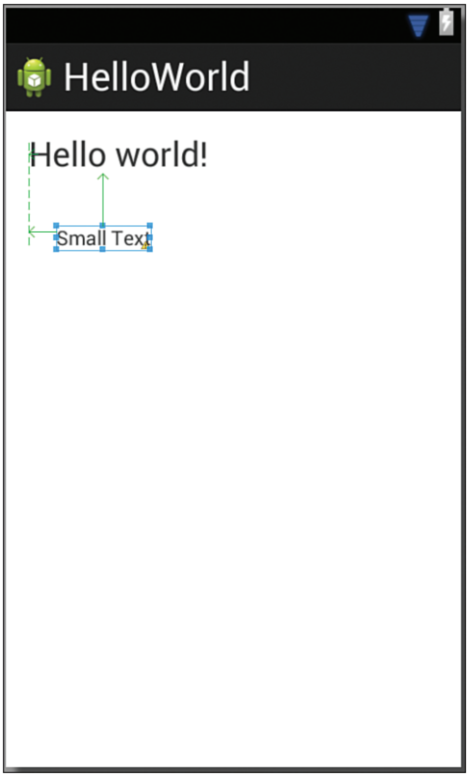


Figure 3.8 A Small TextView positioned properly on a Graphical Layout.

Locate and click the Text Fields folder in the Palette. A number of widgets for entering information are displayed. The widget for entering data in Android is called an `EditText`. Each of the EditText widgets listed is configured for the entry of a different type of data. The different configurations dictate what soft keyboard is displayed when the widget is clicked and, in some cases, how the text is formatted as it is entered. For example, the EditText with the number 42 in it will display a keyboard with only numbers on it, whereas the EditText with `Firstname Lastname` in it will display an alpha character keyboard and it will capitalize each word entered. Drag the Firstname Lastname EditText to the right of the `Name: TextView`. As you are dragging it, pay attention to the green arrows. You want this relative to the Name: TextView, so there should be only one arrow, and it should point at the TextView. A dotted green line should go from the bottom of the TextView through the EditText. This aligns the EditText with the bottom of the TextView.

Click the Form Widgets folder and drag a Small Button below the EditText. In this case you want the green arrow pointing to the EditText and the dotted green line going through the middle of the bottom, from the top of the screen to the bottom, to center it horizontally in the RelativeLayout.

Note

Although Eclipse is a very powerful and useful tool in Android development, the need to make all items in the UI relative makes designing a layout difficult. We recommend that you use the Graphical Layout to get the UI approximately correct and then fine-tune in the XML.

Switch to the XML view for the layout. Locate the `EditText` element. Change the default `id` to `"@+id/editTextName"` so that we have some understanding what data that widget is handling. Change the `marginLeft` attribute to `"5dp"`. There are two new attributes. The first is `android:ems`. This attribute sets the displayed size of the layout to 10 ems. Ems is a size measurement equal to the number of capital Ms that would fit into the control. The second new attribute is `android:inputType`. This attribute tells Android how you want text handled as it's entered and the type of keyboard to display when the user is entering data.

Locate the `Button` element. Change the default id to `"@+id/buttonDisplay"`. There is also a new attribute in this element: `layout_centerHorizontal`. This attribute is set to true to tell Android to center the widget in the parent. Finally, change the text attribute to `"Display"`. Change the value in the `layout_below` attribute to `@+id/editTextName` to match the change you made in the `EditText` element. Switch to the Graphical Layout to see the changes.

Run the app in the emulator using Run > Run Configurations > HelloWorldRunConfig and click the Run button to see the layout as it would appear running (Figure 3.9). The first time you run the emulator, you will have to slide the lock to unlock the device (like a real phone). Note that the emulator might be behind Eclipse, so you will have to minimize windows or in some other way bring it to the foreground. The button clicks but does not do anything. For this you need to write code.

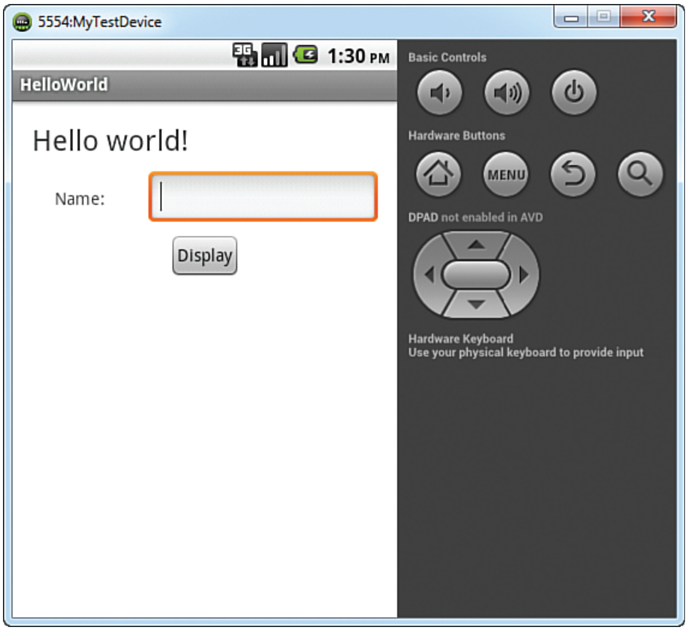


Figure 3.9 Initial run of Hello World.

> **Note**
>
> Either close the activity_hello_world.xml file or switch to the XML view after you are done editing it. The reason is that if you close Eclipse with the layout file open in Graphical mode, Eclipse will take a long time opening the project the next time you want to work on it.

Coding App Behavior

Code to give behavior to the layout is written and stored in the Java class file associated with the layout. Open the HelloWorldActivity.java file by double-clicking it. If it is already open, click its tab in the editor. You should see the basic code structure (Listing 3.4).

Listing 3.4 **Initial Activity Code**

```
                                                                  //1
package com.example.helloworld;
                                                                  //2
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
                                                                  //3
```

```
public class HelloWorldActivity extends Activity {

    @Override
                                                                    //4
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_world);
    }


    @Override
                                                                    //5
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

This code was generated by Eclipse when you created the activity at the start of the HelloWorld project. It is important to understand what this code does to properly code an activity.

1. At the top of the file is the keyword "package" followed by `com.example.helloworld`. This identifies this class as belonging to the Hello World package. All source Java files (in src folder) will have this entry as the first code in the file.
2. After the package line and before any other code are the imports. Click the plus (+) sign in front of the `import android.os.Bundle;` line of code. You should now see three import lines. This code is used to get the source code needed for your activity. The Activity class provides the functionality required for any class that uses or interacts with other Activities used in this class. The Menu class provides the functionality for the menu that is displayed when the user presses the device's Menu button. The Bundle import requires a bit more explanation.

   A `Bundle` is an object for passing data between activities. In this way we can have an application that can perform some activity based on what another activity has done or the data it has used. You will use this functionality later in the book. However, Bundle also performs another very important function. It passes data back to the activity itself. When the user rotates the device, the displayed activity is destroyed and re-created in the new orientation. So that the user doesn't have to start over if this happens, the activity stores its current state just before it is destroyed in a bundle and passes that data to itself when it re-creates the activity in the new orientation.
3. The `public class` line of code begins the Activity class and declares that this class is referred to as `HelloWorldActivity` and that it is a subclass of the SDK-provided Activity class. Within the class are two methods, `onCreate` and `onCreateOptionsMenu`.

Before each method declaration is `@Overide`. This annotation tells the compiler that the following method is to be used in place of the super class's method of the same name.

4. The `onCreate` method is the first method executed by the `Activity` when it is started. The method has a parameter that is of type `Bundle` named `savedInstanceState`. This is the object that contains information on the state of the Activity if it was destroyed in an orientation change as explained earlier. The next line `super.onCreate` calls the super class's `onCreate` method. Because this method is overriding the Activity class's inherited `onCreate` method, it must call that method explicitly to use that functionality to create the Activity. It is passed the `savedInstanceState` bundle. The final line of code is `setContentView(R.layout.activity_hello_world)`. This code tells the activity to use the activity_hello_world.xml file as the layout to be displayed when the activity is running. It is very important to understand the parameter `R.layout.activity_hello_world`. The `R` parameter tells the compiler that we want to use a resource from the layout folder named *activity_hello_world*. Whenever we want to access or manipulate a resource, it has to be referred to in this manner. However, this does not refer directly to the res folders; instead it refers to a file generated by the compiler that is named R.java. To see this file, double-click into the gen folder in the Package Explorer until you see it. You should not edit this file because it is automatically generated by the compiler. The `onCreate` method will be modified with our code to add further functionality to the activity.
5. The `onCreateOptionsMenu(Menu menu)` method is called when the user clicks the device's Menu button. It returns a Boolean (true or false) value indicating whether the menu was successfully created. The first line of code (`getMenuInflator()`) gets an object that can create a menu from the running activity. It then tells it to inflate (create) a visual representation of the menu based on the main.xml file in the menu resource folder and refer to it with the name "menu".

Adding Code

Our app has only one function, to display the name entered into the EditText when the Display button is pressed. Enter the code in Listing 3.5 before the last curly bracket in the activity Java file:

Listing 3.5 **Display Button Code**

```
                                                                                  //1
private void initDisplayButton() {
    Button displayButton = (Button) findViewById(R.id.buttonDisplay);            //2
    displayButton.setOnClickListener(new OnClickListener () {                    //3

      @Override
      public void onClick(View arg0) {
            EditText editName = (EditText) findViewById(R.id.editTextName);         //4
            TextView textDisplay = (TextView) findViewById(R.id.textViewDisplay);   //5
            String nameToDisplay = editName.getText().toString();                   //6
            textDisplay.setText("Hello " + nameToDisplay);                          //7
        }
    });
}
```

This code does the work and illustrates a number of important concepts in Android development.

1. This line declares a new method in the HelloWorldActivity class. The method is only useable by this class (`private`) and does not return any value (`void`). The method signature is `initDisplayButton()`. The signature, or name, of the method is completely up to you. However, you should name it to give some idea what it does.
2. Associate the code with the button on the layout. This line of code declares a variable of type `Button` that can hold a reference to a button and then gets the button reference using the command `findViewById`. All widgets on a layout are subclasses of the `View` class. The method `findViewById` is used to get a reference to a widget on a layout so it can be used by the code. The method can return any `View` object, so you have to use `(Button)` before it to cast the returned `View` to a `Button` type before it can be used as a `Button` by the code. `Button` is underlined in red after you type it in. This is because the code for the button class is not automatically available in the class. You have to import it. Fortunately, this is easy. Hover your cursor over the underlined word and a menu will pop up. Select Import Button... Do this for any other items underlined in red.
3. Set the button's listener. There are a number of different listeners for widgets, which gives great flexibility when coding app behavior. For this button we use an `onClickListener`. The code creates a new instance of the listener and then adds a method (`public void onClick(View arg0)`) to be executed when the button is clicked.

4–5. The code for when the button is clicked gets references to the `EditText` where the name was entered and the `TextView` where the message will be displayed.

6. The name entered by the user is retrieved from the `EditText` and stored in a `String` variable named `nameToDisplay`.
7. The `text` attribute of the `TextView` is changed to the value of the `String` variable.

Notice that `initDisplayButton()` is underlined in yellow. This is because the method is never called by the code. To call it and get the behavior associated with the button to execute, you have to call the method in the `onCreate` method. After the `setContentView` line of code enter

```
initDisplayButton();
```

The yellow underline goes away and your code is done! Run the app in the emulator using Run > Run Configurations, and click the Run button to test your first app. You could also run your app using Run > Run or by pressing Ctrl+F11.

Connecting Code to UI—Android Versus iOS

In both Android and iOS (iPhone and iPad), the user interface (UI) and the code that makes the UI work are stored in different files. This means that both types of app coding require that the code has to be linked to the UI in some way. The chapters in this book that cover iOS explain the process of "wiring up" an interface using the features of the Xcode IDE. However, in Android, connecting the UI to the code is done entirely in the code itself.

Whenever some code needs to use a widget on a layout, it has to get a reference to it using the `findViewById` command. This requires extra coding but provides great flexibility. Forgetting to connect the code to the UI widget needed in both operating systems will result in a runtime error.

Summary

Congratulations! You have built your first app. You created an Android project, designed and coded a user interface, and, finally, made the app do something. Along the way you learned the process of Android App development, the Eclipse development environment, and the components of an Android app.

Exercises

1. Change the Hello World app to allow the entering of a first and a last name and display "Hello *firstname lastname*!" when the button is clicked. Be sure to label the `EditText`s to reflect the new data that is to be input.
2. Add a Clear button. The Clear button should remove any data in the `EditText`(s) and change the display back to "Hello world!"
3. Create a new Android Virtual Device that uses a bigger device to test your app on a different screen size. Run the app using the new AVD.

9

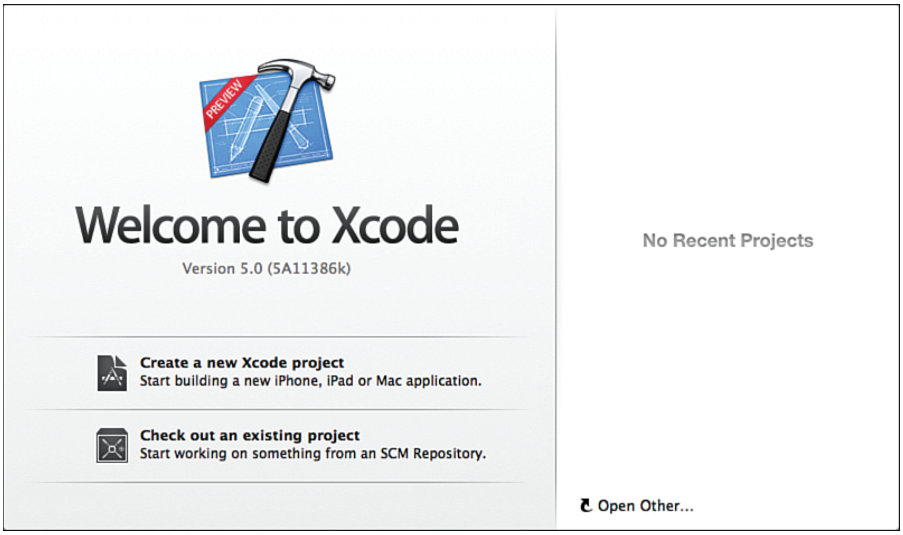
Using Xcode for iOS Development

This part of the book covers how to create iOS apps. You learn to use the powerful Xcode development environment. If you need to get this installed on your computer, refer to Appendix B, "Installing Xcode and Registering Physical Devices," before continuing. In this first chapter, you learn to build a simple but complete iOS app—a variation on the traditional "Hello World" app—and run it on the simulator.

Creating the Xcode Project

You're no doubt eager to get started creating your first iOS app, so jump right in and launch Xcode. You should find it in the Applications folder on your Mac. After Xcode starts, you should see the screen shown in Figure 9.1. Select the Create a New Xcode Project option.

Next, you're given a number of options for creating projects based on various templates, as shown in Figure 9.2. You'll notice in the left sidebar of the window that you can create projects for both iOS and OS X. Our focus here is on iOS applications, so choose that entry. You will see several templates that will make creating a new app simpler. For our first app, choose Single View Application, and click Next.



Welcome to Xcode

Version 5.0 (5A11386k)

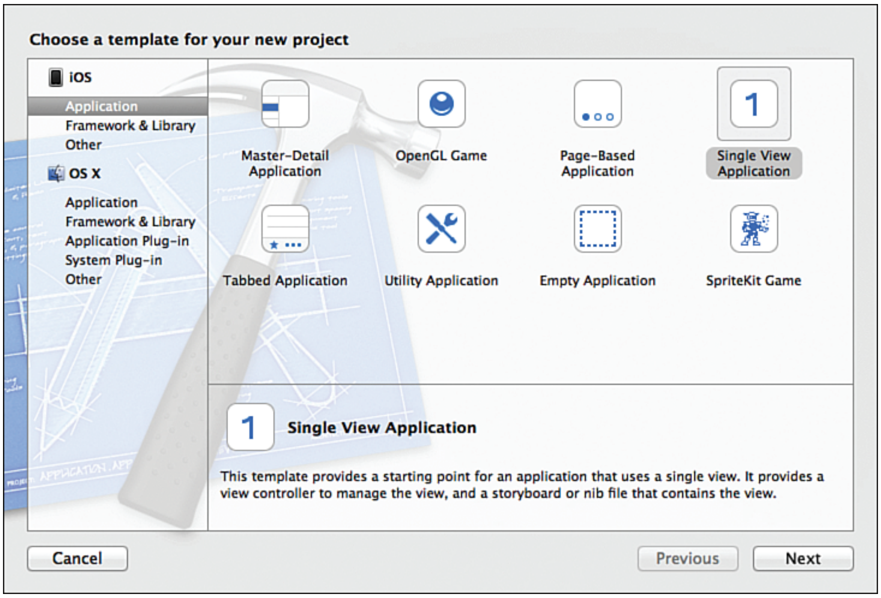
Create a new Xcode project
Start building a new iPhone, iPad or Mac application.

Check out an existing project
Start working on something from an SCM Repository.

No Recent Projects

Open Other...

Figure 9.1 Xcode's Welcome screen.



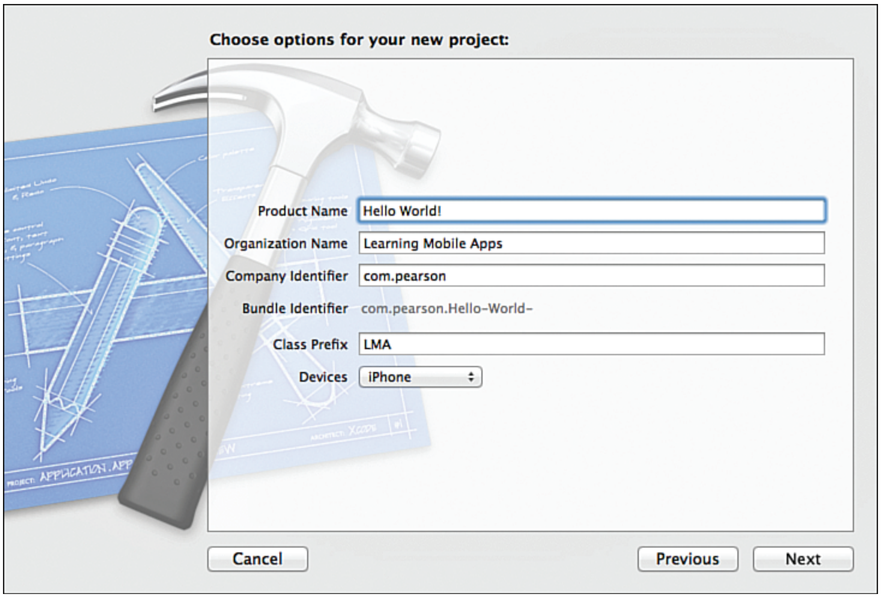
Choose a template for your new project

Single View Application

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

Figure 9.2 Choose the Single View template for the first project.

On the next screen (shown in Figure 9.3), you choose a name for your app. Type "Hello World!" The next fields are not all that important for sample projects like this, but for real projects, you should add your company's name, identifier (reverse web address), and class prefix. The class prefix is at least three capital letters that are prepended to any class you create to distinguish them from library classes. Apple has reserved two-letter prefixes for use with the frameworks that come with the platform, so you will see these in classes like `NSArray`, `CLLocation`, and `UIButton`. Although you don't have to use prefixes or stay with three-letter prefixes, it is a best practice that you should follow. I typically use my initials or initials of the organization. Throughout this book, we use the initials LMA (for Learning Mobile Apps). You can choose your own abbreviation, but it will be easier to follow the code examples if you use LMA. On this screen you can also choose which device to target (iPhone, iPad, or Universal). Universal creates a single app but with a different user interface for iPhone and iPad, enabling the same app to be installed on both devices.



Product Name: Hello World!
Organization Name: Learning Mobile Apps
Company Identifier: com.pearson
Bundle Identifier: com.pearson.Hello-World-
Class Prefix: LMA
Devices: iPhone

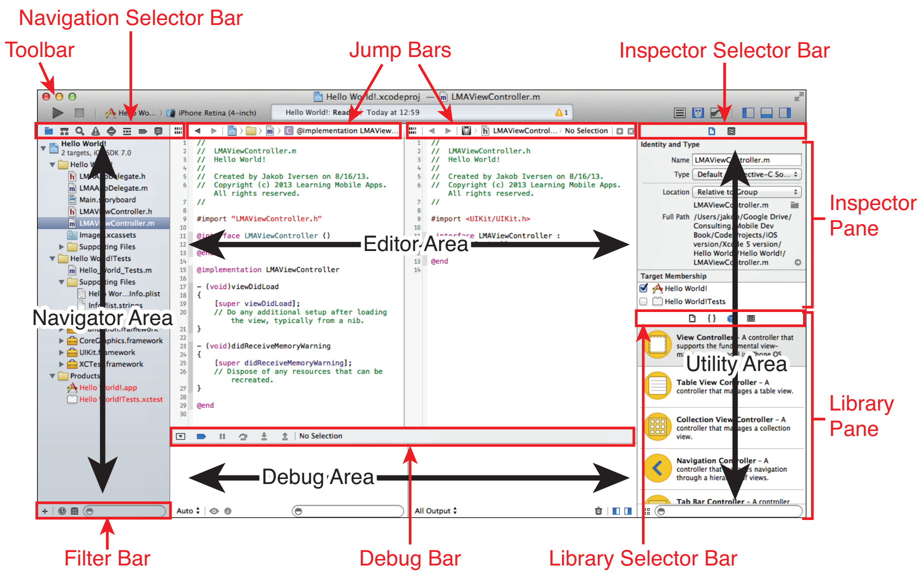
Figure 9.3 Choosing options for the iOS project.

Then click Next, and you will have to choose a location to save your project. You can navigate to an appropriate place on the disk to create the project files. Click Create.

Xcode Project Folder

The Xcode project is created in a folder and consists of a file with the extension .xcodeproj and a number of other files and folders. You can easily move the entire project between computers by copying the directory that contains the .xcodeproj file and any subfolders as well. We have found that when we work on our regular computers (office, home, and so on), using Google Drive or Dropbox works well to keep all the files of a project in sync. But we often find ourselves compressing the project folder and emailing or copying to a thumb drive to make sure we have a good copy of the project. To reopen a project that has been moved, you can double-click the .xcodeproj file. You also have the option of using version control systems by taking advantage of Xcode's built-in support of Git.

After creating the project, you're now looking at the main Xcode workspace window. Figure 9.4 shows an overview of the Xcode workspace.



```
//
//  LMAViewController.m
//  Hello World!
//
//  Created by Jakob Iversen on 8/16/13.
//  Copyright (c) 2013 Learning Mobile Apps.
    All rights reserved.
//

#import "LMAViewController.h"

@interface LMAViewController ()


@implementation LMAViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading
       the view, typically from a nib.
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be
       recreated.
}

@end
```

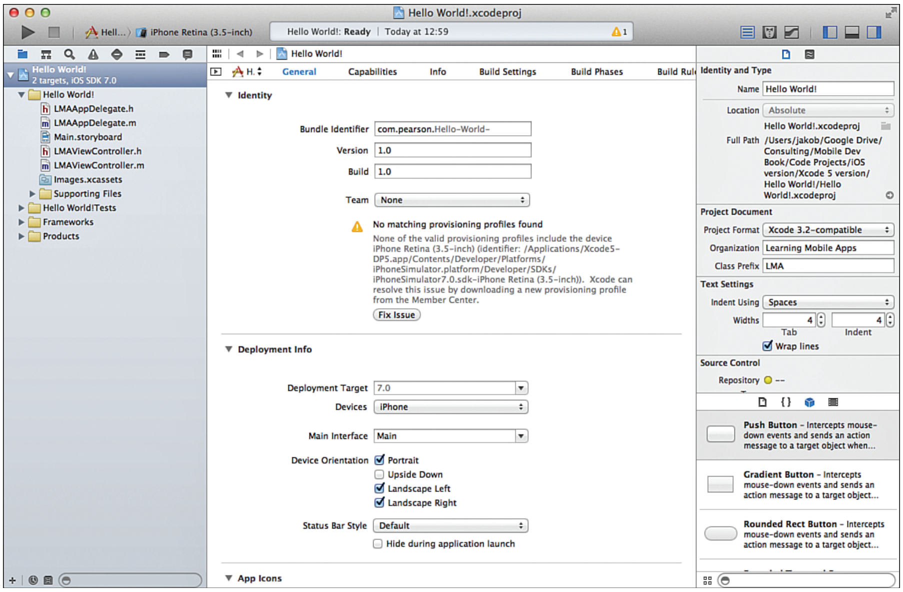
```
//
//  LMAViewController.h
//  Hello World!
//
//  Created by Jakob Iversen on 8/16/13.
//  Copyright (c) 2013 Learning Mobile Apps.
    All rights reserved.
//

#import <UIKit/UIKit.h>

@interface LMAViewController :

@end
```

Figure 9.4 Overview of the Xcode workspace.

Xcode is a very powerful development environment with a lot of functionality. If you decide to do any serious development for iOS, you should take some time to figure out how everything works. You can find a detailed description of the Xcode workspace in the documentation (Help > Xcode Overview). We won't go into a lot of detail now, but you will discover some of the Xcode functionality as you need it. However, if you take some time to look through the documentation, you will likely save a lot of time later on.

Project Settings

After you've created the Hello World project, you should see the view of Xcode as shown in Figure 9.5. In the center of the workspace is a summary of the app and several appwide settings. The first section enables you to specify the version and build for the app. The version is used when the app is published to the App Store. Anytime an app with a higher version number is published, all your users will be prompted to download a new version. The build number is for internal use by the developer. You can choose which device types to target, as well as which version of iOS you want to target. As of this writing, the current version of iOS is 7.0. This setting determines the minimum version of iOS your users have to be running in order to run your app. This is just a signal within the app store. Apps are typically built using the latest available base SDK, so if you use features in a later SDK than your deployment target, you will need to insert checks in your code to make sure your app doesn't crash on devices with older versions of iOS.



Identity

Bundle Identifier: com.pearson.Hello-World-
Version: 1.0
Build: 1.0
Team: None

No matching provisioning profiles found
None of the valid provisioning profiles include the device iPhone Retina (3.5-inch) (identifier: /Applications/Xcode5-DP5.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator7.0.sdk-iPhone Retina (3.5-inch)). Xcode can resolve this issue by downloading a new provisioning profile from the Member Center.
Fix Issue

Deployment Info

Deployment Target: 7.0
Devices: iPhone
Main Interface: Main
Device Orientation: ☑ Portrait ☐ Upside Down ☑ Landscape Left ☑ Landscape Right
Status Bar Style: Default
☐ Hide during application launch

App Icons

Identity and Type
Name: Hello World!
Location: Absolute
Hello World!.xcodeproj
Full Path: /Users/jakob/Google Drive/Consulting/Mobile Dev Book/Code Projects/iOS version/Xcode 5 version/Hello World!/Hello World!.xcodeproj

Project Document
Project Format: Xcode 3.2-compatible
Organization: Learning Mobile Apps
Class Prefix: LMA

Text Settings
Indent Using: Spaces
Widths: Tab 4, Indent 4
☑ Wrap lines

Source Control
Repository --

Push Button – Intercepts mouse-down events and sends an action message to a target object when...
Gradient Button – Intercepts mouse-down events and sends an action message to a target object...
Rounded Rect Button – Intercepts mouse-down events and sends an action message to a target object...

Figure 9.5 Overview of the Xcode workspace with our newly created Hello World app.

This is also where you can specify which device orientations are supported. By default, iPhone apps support Portrait and Landscape Left and Right but not Upside Down (iPad default is to support all four orientations).

On the left, in the navigation area, you can see the files that Xcode created for you. Figure 9.6 shows what it should look like. The exact number and types of files created depends on which choices you made when creating the project. Here's an overview of some of the files and folders created in this project:

- AppDelegate.h and AppDelegate.m—The App Delegate files manage issues related to the entire app and are primarily used to manage the life cycle of the app—how it is started, what happens when it goes to the background, and so on. This life cycle is covered in more detail in Chapter 2, "App Design Issues and Considerations," and in Chapter 11, "Persistent Data in iOS." Objective-C programs follow the C-style and have both a header (.h) and method (.m) file. See Appendix C, "Introduction to Objective-C" for more detail.

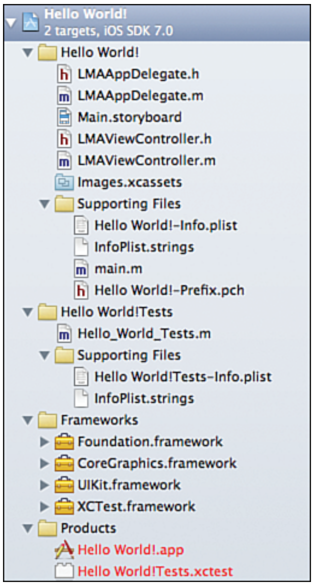


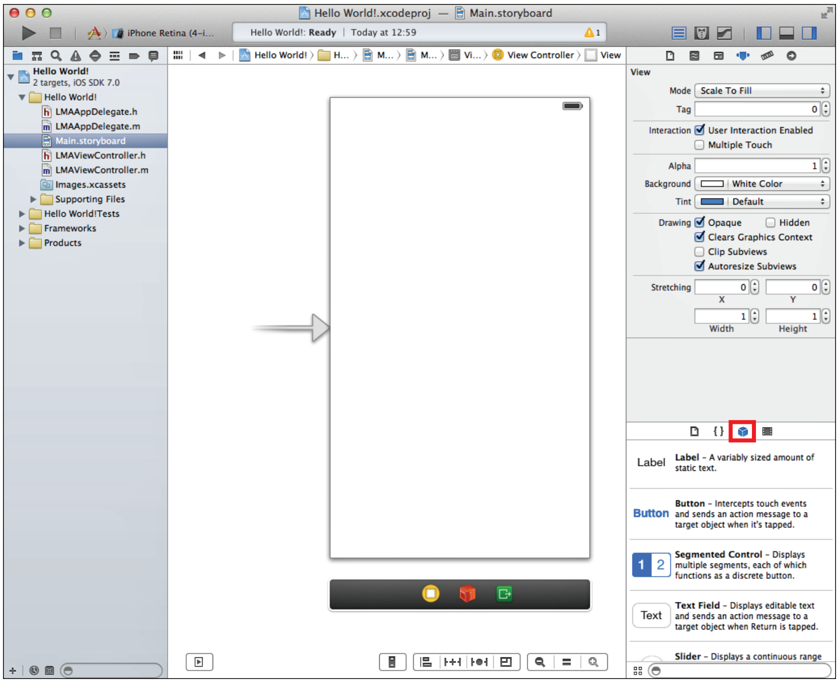
Figure 9.6 Contents of the Hello World project.

- Main.storyboard—The storyboard is used to design the interaction between multiple screens in your app as well as designing the layout of the individual screens.
- ViewController.h and ViewController.m—The view controller contains the code that controls the user interactions with the app. Most of the programming we do in this book will be in these files.
- Images.xcassets—This folder contains all the images, including icons, needed for your app.
- Supporting Files—This directory contains a number of files that the app may or may not use. Here's a description of a few of them:
  - Hello World!-Info.plist—This file contains a few app-specific settings. Most of these are controlled in other parts of Xcode.
  - main.m—The file that is responsible for launching the app.
- Hello World! Tests—Xcode comes with a built-in Unit Test framework, which you can use to create automated testing frameworks to ensure you deliver quality code. We recommend using Unit Tests on all production projects, but here the focus is on learning how to create iOS apps, so we don't have room to also cover unit testing. You can read more about this framework in *Test-Driven iOS Development* by Graham Lee.

- Frameworks—These are various libraries that you can include in your project to add functionality to your app, such as maps or audio capabilities. The default ones that are already included are *UIKit*, which is responsible for all the user interface controls, *Foundation*, which has a lot of the core functionality needed in any program, such as object-oriented data types, and *CoreGraphics*, which handles low-level graphical tasks and is used by UIKit.
- Products—This is your compiled app file.

Creating the User Interface

To open a file for editing in Xcode, you need to click it only once. Double-clicking will open it in a separate window. Click once on the Main.storyboard file. This opens the file in Interface Builder (see Figure 9.7), where you can easily create the user interface for your app. With the storyboard open, you can drag user interface elements from the utility pane on the bottom right and control a range of settings on the top of the utility pane.



View

Mode: Scale To Fill
Tag: 0
Interaction: ☑ User Interaction Enabled ☐ Multiple Touch
Alpha: 1
Background: White Color
Tint: Default
Drawing: ☑ Opaque ☐ Hidden ☑ Clears Graphics Context ☐ Clip Subviews ☑ Autoresize Subviews
Stretching: X 0, Y 0, Width 1, Height 1

Label — **Label** – A variably sized amount of static text.

Button — **Button** – Intercepts touch events and sends an action message to a target object when it's tapped.

1 2 — **Segmented Control** – Displays multiple segments, each of which functions as a discrete button.

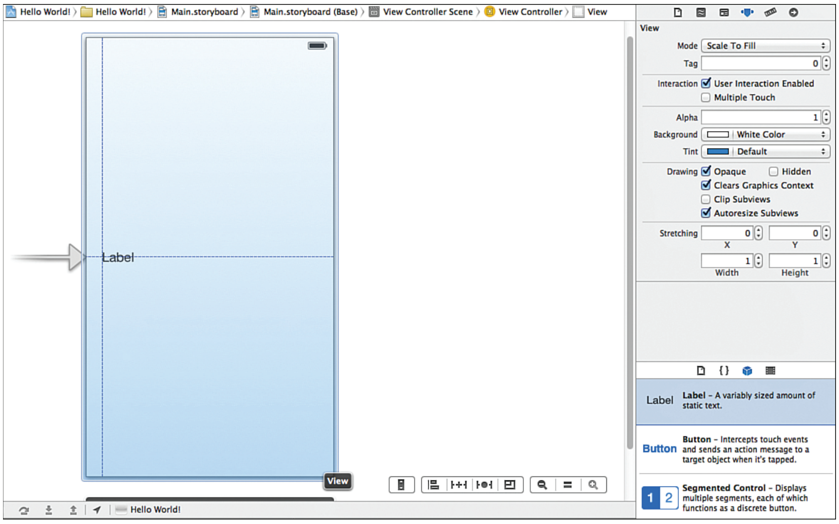
Text — **Text Field** – Displays editable text and sends an action message to a target object when Return is tapped.

Slider – Displays a continuous range

Figure 9.7 Interface Builder.

In the lower right of the utility area, you should see the Object Library, which contains all the user interface elements you can use in your app. If you don't see the Object Library, click the cube-shaped icon highlighted in Figure 9.7.

Start by dragging a label onto the user interface canvas (see Figure 9.8). You may have to scroll down the list of controls to find the label. You can also use the Search bar below the controls and type in **label**. Notice that you get blue dotted guidelines as you drag the label around. Drag it to the middle left of the screen and let go when both guidelines appear. Double-click the label to change the text of it to "Hello World!" and then press Enter. Expand the label by clicking the right side of it and dragging it to the right side until the blue dotted guidelines appear. Above the utility area on the right, you have a little menu bar of five items. The fourth one from the right should be selected. This is the Attributes Inspector, which enables you to set many properties for the currently selected user interface element. For the label, you can change its appearance quite a bit. Feel free to play around, but you just need to center the text of the label.


Hello World! 〉 Hello World! 〉 Main.storyboard 〉 Main.storyboard (Base) 〉 View Controller Scene 〉 View Controller 〉 View
Label
View
Hello World!

View

Mode: Scale To Fill
Tag: 0
Interaction: ✓ User Interaction Enabled
☐ Multiple Touch
Alpha: 1
Background: White Color
Tint: Default
Drawing: ✓ Opaque ☐ Hidden
✓ Clears Graphics Context
☐ Clip Subviews
✓ Autoresize Subviews
Stretching: X 0, Y 0, Width 1, Height 1

Label – A variably sized amount of static text.

Button – Intercepts touch events and sends an action message to a target object when it's tapped.

Segmented Control – Displays multiple segments, each of which functions as a discrete button.

Figure 9.8 Dragging a label onto the canvas and using guidelines for placement.

Running the App in the Simulator

Launching the app in the built-in simulator that comes with Xcode is quite simple. In the top-right corner of Xcode, you will see a big Run button, and next to that, something called a Scheme, which enables choosing which device is targeted (see Figure 9.9). Click the right side of the scheme and choose the iPhone simulator. If you have a registered physical device connected to your computer, it will also show up in this list. See Appendix B for how to register physical devices to run your apps.

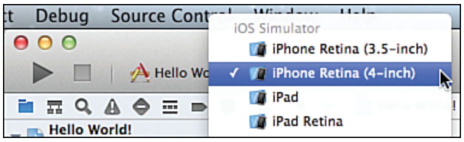


Figure 9.9 Choosing the iPhone simulator to run the app.

Click the Run button on the top left of the Xcode toolbar and wait a few seconds for the simulator to launch with your app (see Figure 9.10). You can control how the simulator looks and behaves in the Hardware menu. If you choose a device with a high-resolution screen, it may not fit on your computer screen comfortably. In that case, you can go to Window > Scale and choose a zoom level.

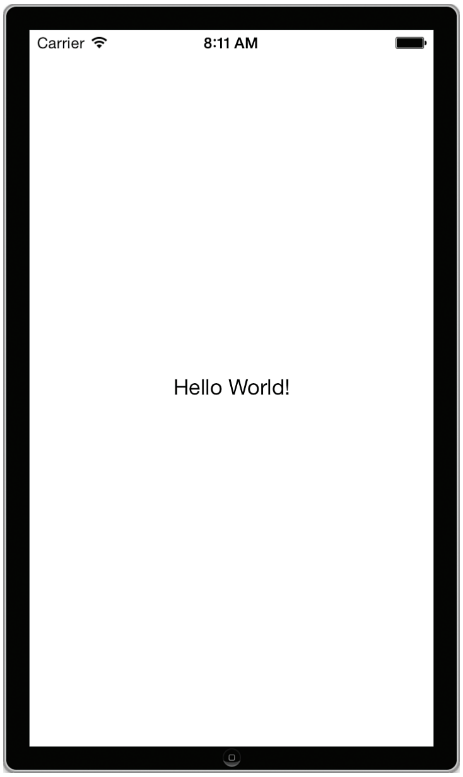


Figure 9.10 iOS simulator with the Hello World! App.

Adding App Behavior

Switch back to Xcode. Next, you'll see how to add some real functionality to your app. First, you'll need to set up the user interface, so make sure the storyboard is open. Then double-click the Hello World label and change it to **Please enter your name.** Drag a Text Field (scroll or search for it in the Object Library) onto the canvas and place it below the label. Then drag a button below the text field. Double-click the button and change its text to **Tap Here!** Finally, add a label below the button. Make the label as wide as the width of the screen, delete its text, and specify its content to be centered and blue.

Select the text field and look in the Attributes Inspector at some of the settings available. Change the Capitalization to Words, and then look at the Keyboards option. You can specify a keyboard that will show up on screen that is suited to the kind of data being input (for instance, if you need to have the user input only numbers, you can specify a Number Pad. This would be a good time to try out the different keyboards. You need to run the simulator to test the effect each time you choose a different one. Before moving on, make sure the keyboard is set to Default. Figure 9.11 shows the completed UI in both the Interface Designer and the iOS simulator with the default keyboard activated.

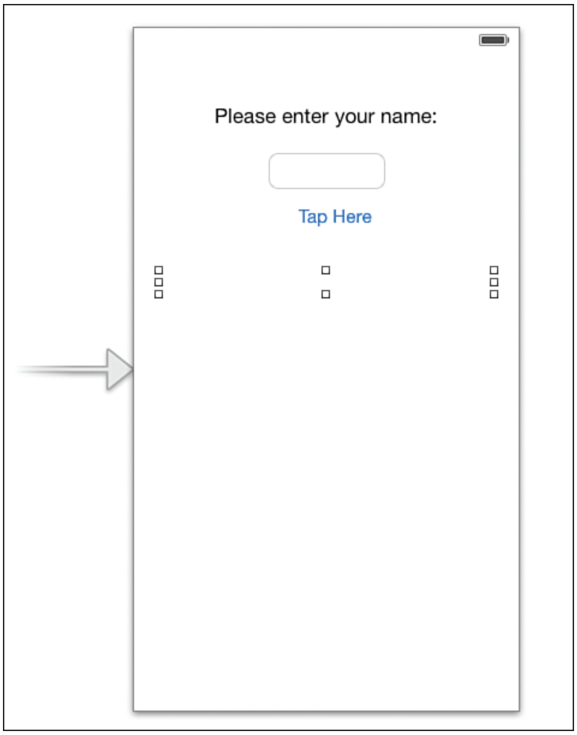


Figure 9.11 Completed UI in Interface Designer and the iOS simulator.

Having created the user interface, your next task is to add some action to the app. The action you want is to have the app take the name entered and say Hello to the user by name.

To do this, return to Xcode and click Stop to quit the simulator. Then make sure the storyboard is open. Next, click the Show Assistant Editor button in the top-right corner of the Xcode window (it's the second button from the left; see Figure 9.12).

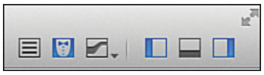
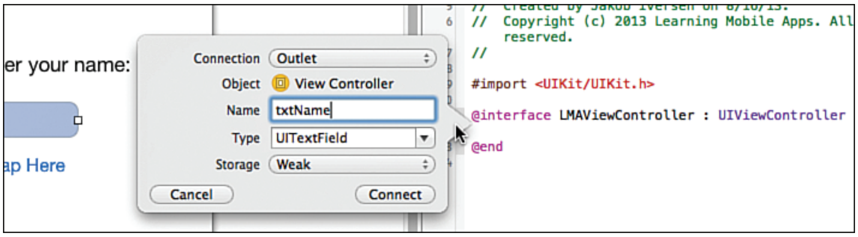


Figure 9.12 Creating an outlet for a user interface element.

This opens an extra editor that by default contains the file that best matches what is displayed in the main window. For this user interface screen, this is the header file for the view controller (LMAViewController.h). We will need to create what's called outlets for those user interface elements we want to be able to access from the code. This includes the two text fields, where we need to be able to read the text the user entered and the bottom label that will be updated to contain our own text string based on the two text fields. To create the outlets, hold down the Control key, click the text field, and drag to the view controller between the `@interface` and `@end` entries. Then let go, and you should see the situation as shown in Figure 9.13.



Connection: Outlet
Object: View Controller
Name: txtName
Type: UITextField
Storage: Weak

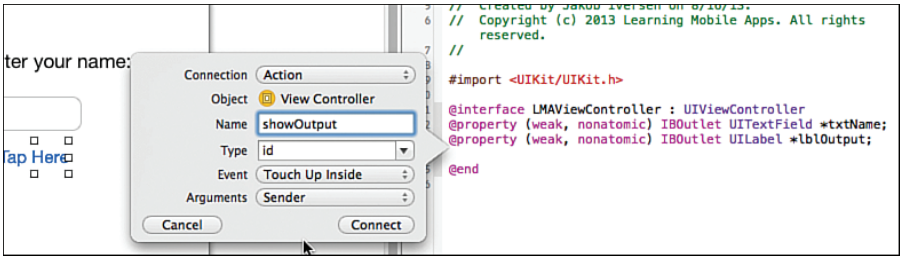
```
//  Copyright (c) 2013 Learning Mobile Apps. All
    reserved.
//

#import <UIKit/UIKit.h>

@interface LMAViewController : UIViewController

@end
```

Figure 9.13 Creating an outlet for a user interface element.

Enter `txtName` in the Name field and click Connect. You should now have this line of code in LMAViewController:

```
@property (weak, nonatomic) IBOutlet UITextField *txtName;
```

For more detail on what this code means, you can look in Appendix C. For now, all you need to know is that this has provided a name for the text field that we can reference in our code by adding an underscore in front of the property name. Do the same for the bottom label, naming it `lblOutput`. Next, you'll add the code for the button. Switch back to the storyboard. The first step is the same: control-drag from the button to the view controller below the properties for the text fields and label. However, this time, in the top drop-down choose Action instead of Outlet (See Figure 9.14). Give it the name `showOutput`. For the Event, we will use the default Touch Up Inside, but take a moment and look through the list of all the possible events that this button will respond to. Touch Up Inside means that the button responds to events where users touched the button and then released their fingers while still inside the button. The convention in iOS is that to cancel a touch, you would drag your finger outside the target and then let go (try it on your own device to see how it works). Leave the Arguments as `Sender`.



Connection: Action
Object: View Controller
Name: showOutput
Type: id
Event: Touch Up Inside
Arguments: Sender

```
//  Copyright (c) 2013 Learning Mobile Apps. All rights
    reserved.
//

#import <UIKit/UIKit.h>

@interface LMAViewController : UIViewController
@property (weak, nonatomic) IBOutlet UITextField *txtName;
@property (weak, nonatomic) IBOutlet UILabel *lblOutput;

@end
```

Figure 9.14 Creating the action for the button.

Connecting Code to UI—iOS Versus Android

In both Android and iOS, the user interface (UI) and the code that makes the UI work are stored in different files. This means that both types of app coding require that the code has to link to the UI in some way. In iOS, this is often referred to as "wiring up" the user interface. However, in Android, connecting the UI to the code is done entirely in the code itself.

If you've ever created a program with a UI on a different platform, you're probably used to having to provide variable names for all UI elements. In iOS and Android, we just provide names to those UI elements we will need to access in code. So, the static label at the top of our UI isn't given a name. The same goes for the button, where we will need to intercept the event that happens when the user taps the button.

In Android, whenever some code needs a reference to a control, we use a special command that will find it by its ID. This requires extra coding but provides great flexibility. Forgetting to connect the code to the UI widget needed in either operating system will result in a runtime error.

Switch the LMAViewController.m file, and notice that you now have a method at the bottom of the file called `showOutput:`. Between the curly braces of this method, enter the code shown in Listing 9.1.

Listing 9.1 **The `showOutput:` Method**

```
- (IBAction)showOutput:(UIButton *)sender {
    NSString *name = [_txtName text];
    NSString *output = [NSString stringWithFormat:
                                @"Hello %@!", name];
     [_lblOutput setText:output];
}
```

A brief explanation: The first line is the method declaration (Appendix C has more detail on how Objective-C methods are declared). The second line declares a string variable (`NSString`) and assigns the value in the text field (`_txtName`) by calling the text methods on the property. Notice the use of the underscore to refer to the property. The third line declares an `NSString` object that is initialized with a string that combines the string "Hello" with the name variable followed by an exclamation mark. The last line calls the `setText` method on the label, passing in the value of the output string. Run the app and test it by entering different names and touching the Tap Here! button.

UI Design—iOS Versus Android

UI design in iOS is done through absolute positioning, where each UI item is held to a fixed position on the screen. However, in Android, relative positioning is used. This enables creation of UI designs that are independent of the physical screen of the device. This means that app developers don't have to worry (too much) about screen sizes of different devices. However, relative positioning also means that the position of one control affects other controls. Often in Android, moving one control changes the whole design.

In iOS we need to provide a different UI layout for different screen sizes. If you want your app to run on both iPad and iPhone, you have to create two separate storyboard files and corresponding view controllers. These screens all run on the same code, so during design the developer must be sure to be perfectly consistent between the different screens needed.

Dismissing the Keyboard

As you may have noticed, the keyboard doesn't go away by itself when a text field loses focus. To get the keyboard to disappear, you have to add a little code to the program. What you need to do is change the View, which is the background of the app, so that it is able to respond to a tap. When the view intercepts a tap, it will then send a message to the text field to resign control. This makes the keyboard disappear. The first thing to do is set up the code to handle the event and then tie the event to the code.

In LMAViewController.h, add this line between the `@interface` line and the `@end` line to define a new action method:

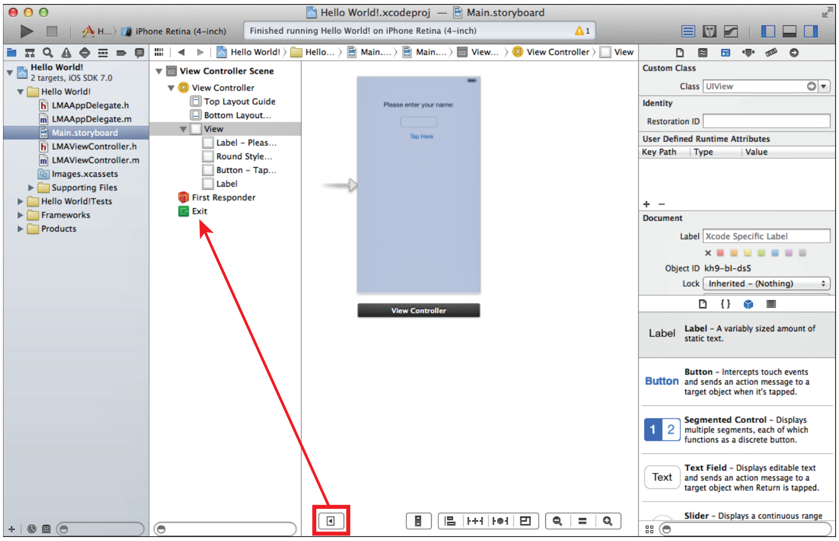
```
- (IBAction)backgroundTap:(id)sender;
```

In LMAViewController.m, add the code in Listing 9.2 to implement the method.

Listing 9.2 **The `backgroundTap:` Method**

```
-(IBAction)backgroundTap:(id)sender
{
    [self.view endEditing:YES];
}
```

This code tells the View to end editing, which will cause the keyboard to disappear. Next, you will have to specify how this code gets called. Select the storyboard file. Make sure the Dock is in list mode. The Dock is the vertical bar between the left and center panes. To expand it to list view, click the triangle in a rounded rectangle (this is already done in Figure 9.15). After the Dock is in list mode, select the top-level View (see Figure 9.15). The View is the background canvas that all the other controls sit on. First, you will need to change this to a Control, so it can fire events. With the View selected, show the Identity Inspector in the far-right pane (you can also press Option-Cmd-3 to open the Identity Inspector—and the other options in that section can be accessed by just changing the number, so Option-Cmd-4 will open the Attributes Inspector). Change the Class field to `UIControl` by simply typing over `UIView` (see Figure 9.16). All controls that are capable of firing events are subclasses of `UIControl`, so by changing the underlying class from `UIView` to `UIControl` we make the View capable of firing events.



Finished running Hello World! on iPhone Retina (4-inch)

Custom Class
Class UIView
Identity
Restoration ID
User Defined Runtime Attributes
Key Path | Type | Value
Document
Label Xcode Specific Label
Object ID kh9-bl-dsS
Lock Inherited – (Nothing)

Label – A variably sized amount of static text.

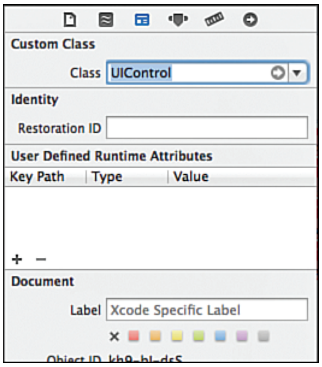
Button – Intercepts touch events and sends an action message to a target object when it's tapped.

Segmented Control – Displays multiple segments, each of which functions as a discrete button.

Text Field – Displays editable text and sends an action message to a target object when Return is tapped.

Slider – Displays a continuous range

Figure 9.15 Xcode with the Dock in List View.



Custom Class
Class UIControl
Identity
Restoration ID
User Defined Runtime Attributes
Key Path | Type | Value
Document
Label Xcode Specific Label
Object ID kh9-bl-dsS

Figure 9.16 Changing the View class to `UIControl`.

Select the Connections Inspector in the right pane (Option-Cmd-6). This shows all the possible actions that can be taken for the current control and can be used to connect those actions to methods in the code. Drag the circle by Touch Down to the View Controller in the Dock (see Figure 9.17). When you release it, select `backgroundTap:`. This will then call the `background-Tap:` method every time the Touch Down event fires on the View.

Note

Touch Down means the event fires as soon as the user taps, in contrast to the Touch Up Inside event that we used earlier, which fires only if the user releases the finger inside the specified control.

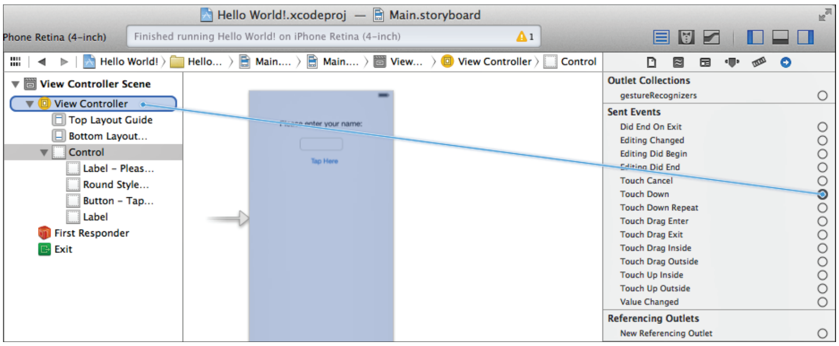


Figure 9.17 Connecting the Touch Down event for the View to File’s Owner.

The View Controller object is the object that loads the current view controller (single screen in an iOS app)—typically the `UIViewController` class itself. Connecting to View Controller in this manner is the same as connecting to the methods in the code, so this is just a different technique for achieving the same result. Click some of the other controls to view them in the Connections Inspector and see how events are linked to methods (see Figure 9.18).

Run the app and see how the keyboard disappears when you tap outside the text field.

Note

If you are developing iPad apps, this is not an issue, because the iPad keyboard has a key to make it disappear, but this doesn’t exist in the iPhone keyboard.

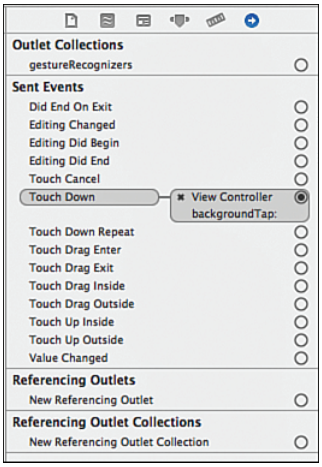


Figure 9.18 Connection Inspector after setting up the action to dismiss the keyboard.

Quick Reference: Dismiss the Keyboard

For future reference, here are the four steps needed to have the keyboard dismissed:

1. Make the View a control, by changing its class from `UIView` to `UIControl`.
2. Define the `backgroundTap:` action method in the .h file.
3. Implement `backgroundTap:` in the .m file to end editing (Listing 9.2).
4. Control-drag from Touch Down in Connections Inspector with Control (formerly View) selected to View Controller in the expanded Dock. Choose the `backgroundTap:` method.

App Icons and Launch Images

The images.xcassets folder is called the Asset Catalog and was introduced with Xcode 5 as a way to manage all the images needed for your app, including app icons and launch images.

App Icons are graphical images that are used to indicate your app on the home screen of the iOS device your app is running on. When you create the icon for your app, you should be prepared to create it in a number of resolutions so that it looks great on different devices, and for various uses within the app as well. The icon is used for three places:

- On the home screen, to give the user an easily recognizable image of your app.
- In Spotlight results when the user is searching on the device.
- In the Settings app where the user can change various settings for your app.

In each of these three places, the icon is supplied in different resolutions, and the resolutions also differ between iPad and iPhone as well as between whether the device is a regular display or a retina display. There can also be differences between whether your app is targeting iOS 7 or earlier versions of iOS. In all, a Universal app targeting both iPhone and iPad and made available for both iOS 6 and iOS 7 may have to have as many as 14 versions of the app icon.

Fortunately, the asset catalog makes it relatively simple to find out what you need. Click the images.xcassets folder and then AppIcon (Figure 9.19). On the right you see three spots for icons. For this app, the icons will be supplied only for iOS 7 and iPhone. To determine the resolution you need to supply, you look at the number in the last line under each spot (29pt, 40pt, and 60pt). This is how many logical points the image takes up. However, if you look just below each of these images, it says 2x, which means these images will be used on a retina display, so the resolution has to be doubled because a retina display has twice as many pixels in each direction as a regular display. This means that these three images have to be 58x58, 80x80, and 120x120 pixels, respectively.

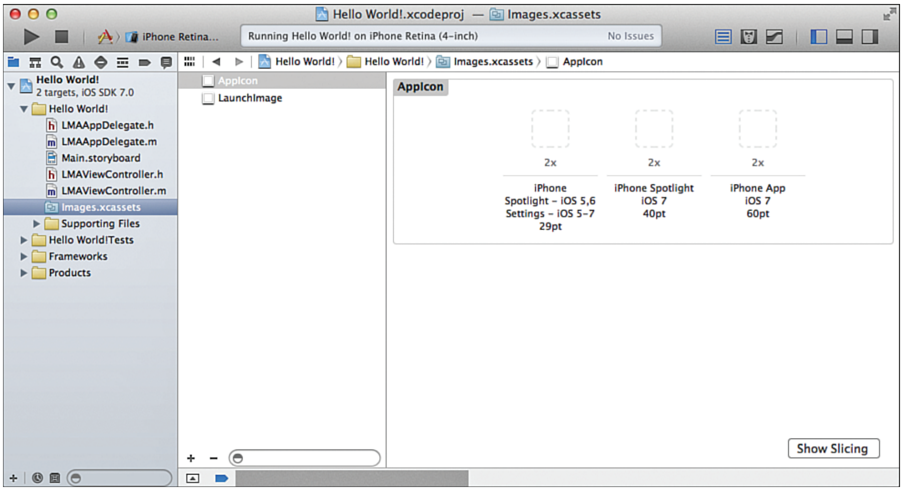


Figure 9.19 Asset catalog.

To supply icons for other situations, you can right-click anywhere with a white background in the asset catalog and select New App Icon. This will give you many more options, as shown in Figure 9.20. The same principle applies, though. Use the pt number and multiply by the 1x or 2x number to get the resolution of the image.

Test out the app icon resolutions by dragging the icons available into the appropriate spot. A very simple icon is available in various resolutions. You can test how the icon looks by running the app in the simulator and then clicking the Home button (Hardware > Home) to get to the home screen where the app icon will be shown along with the app name. You can see the Spotlight icon in the simulator if you click the home screen and drag down. This will pull down a search field. Type **Hello** into the resulting search field, and the icon will show up in the search results.

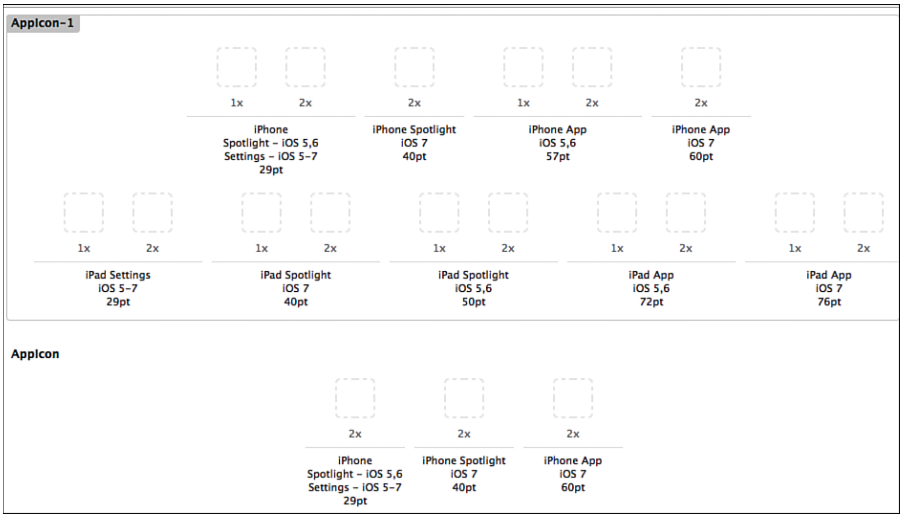


Figure 9.20 Possible app icon resolutions.

To have your app listed in the app store, you also need to supply icons in sizes of 512x512 and 1024x1024 pixels.

The other option that is available in the asset catalog is launch images. A launch image is shown as the app is launching. Apple recommends that your launch image is a blank version of the app's first screen. This way, the user will quickly see the app and what it looks like, before it is filled in with data. Launch images are also supplied in different resolutions depending on the device. Table 9.1 shows the possible resolutions for launch images.

Table 9.1 **Typical Launch Image Dimensions**

Device	Portrait	Landscape
iPhone and iPod touch	320 x 480 pixels 640 x 960 pixels (`@2x`)	Not supported
iPhone 5 and iPod touch (5th generation)	640 x 1136 pixels (`@2x`)	Not supported
iPad	768 x 1004 pixels 1536 x 2008 pixels (`@2x`)	1024 x 748 pixels 2048 x 1496 pixels (`@2x`)

iPhone Screen Sizes and Resolutions

When the first iPhone was released, the resolution was set to 320x480 pixels. Programmers would use this coordinate grid to arrange their user interface. When the iPhone 4 was released, it sported a retina display with double the resolution (640x960). However, from the perspective of the programmer, the original 320x480 grid was still used to position everything on the screen, so apps didn't have to be updated to handle the higher resolution. But all the UIKit controls were rendered in higher resolution, and all images could now be supplied in retina versions with double the resolution. If you need an image in your app, you can create a regular file to be used on nonretina screens, as well as a version with double the resolution for retina screens. By dragging the file into the appropriate spot in the images.xcassets container, the system will automatically pick the right one for the device the app is running on.

With the iPhone 5 and iPod touch 5, Apple again changed the resolution, this time increasing the vertical size to 1136 pixels (giving a grid of 320x568). Images can now also be supplied in this higher resolution. The iPhone 5 increased the physical size of the screen from 3.5 to 4 inches, so when you see references to a 4-inch screen, this is the screen introduced with the iPhone 5 (and iPod touch 5).

Summary

Congratulations! You have built your first iOS app. You created an Xcode project, designed and coded a user interface, and finally made the app do something. Along the way you learned the process of iOS app development, the Xcode development environment, and the components of an iOS app.

Exercises

1. Split the name field into first and last name. Then make sure both first and last names show up when tapping the button.
2. Change the functionality of the `showOutput` method so that if no text has been entered, the output changes to Hello World!
3. Explore the properties for the text fields, buttons, and labels within Xcode. Change the label to green and bold text. Change the border style for the text field, and add a clear button.
4. Add a new button to the app, with a method that will change the `lblOutput` text to Hello World!
5. Rotate the simulator while running the app.

6. Run the app in the iPad simulator. Then change the Devices setting to Universal and run again in the iPad simulator.
7. Have the keyboard dismiss when the user taps the button.
8. Add a new field to enter a number, and set the keyboard to be numeric. Be sure that the keyboard dismisses appropriately from this new field.

Index