DEITEL® DEVELOPER SERIES

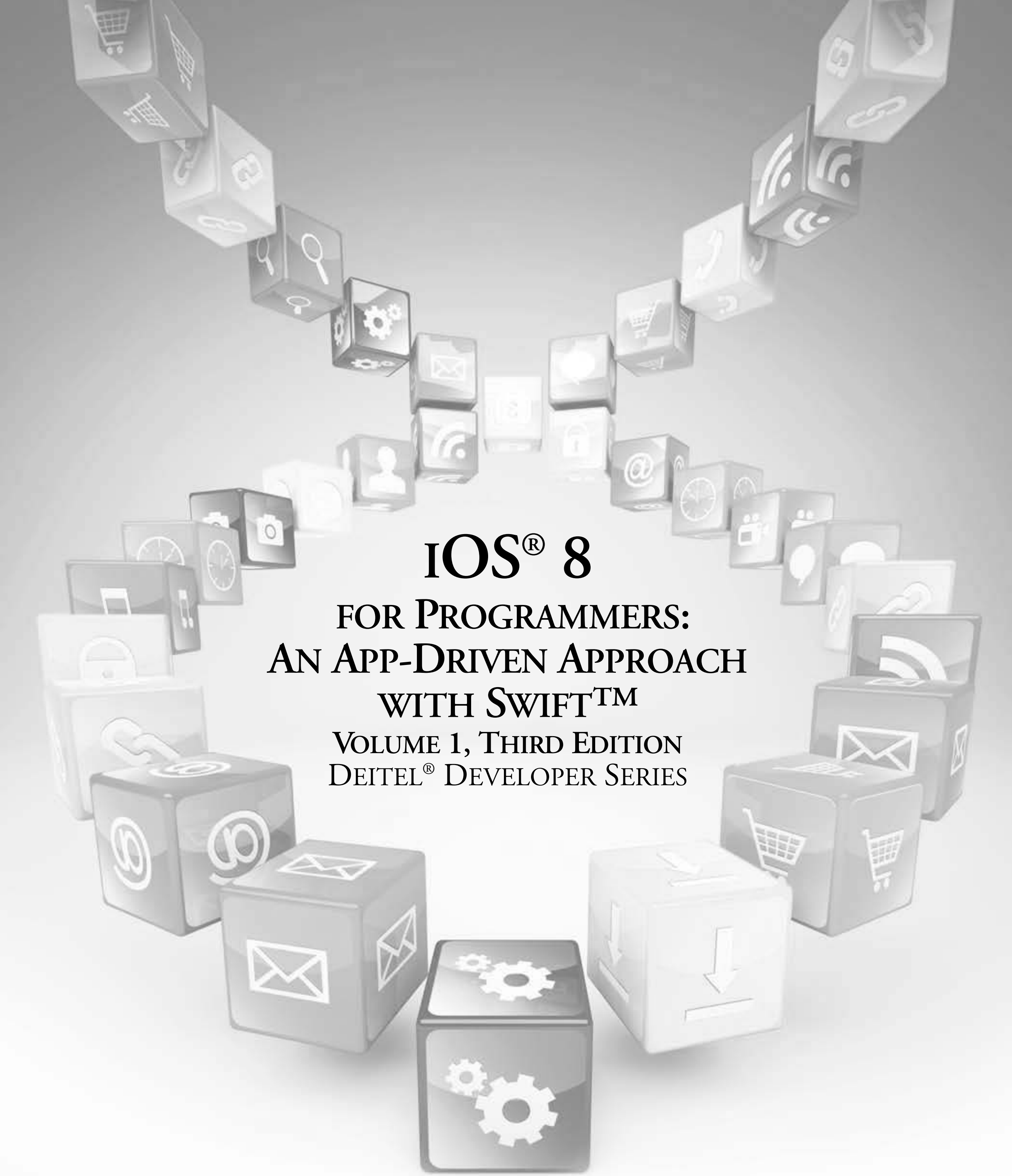
THIRD EDITION

7 Fully Coded iOS® Apps

iOS® 8 for Programmers

An App-Driven Approach

PAUL DEITEL • HARVEY DEITEL
ABBEY DEITEL



iOS® 8

for Programmers: An App-Driven Approach with Swift™

Volume 1, Third Edition

Deitel® Developer Series

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

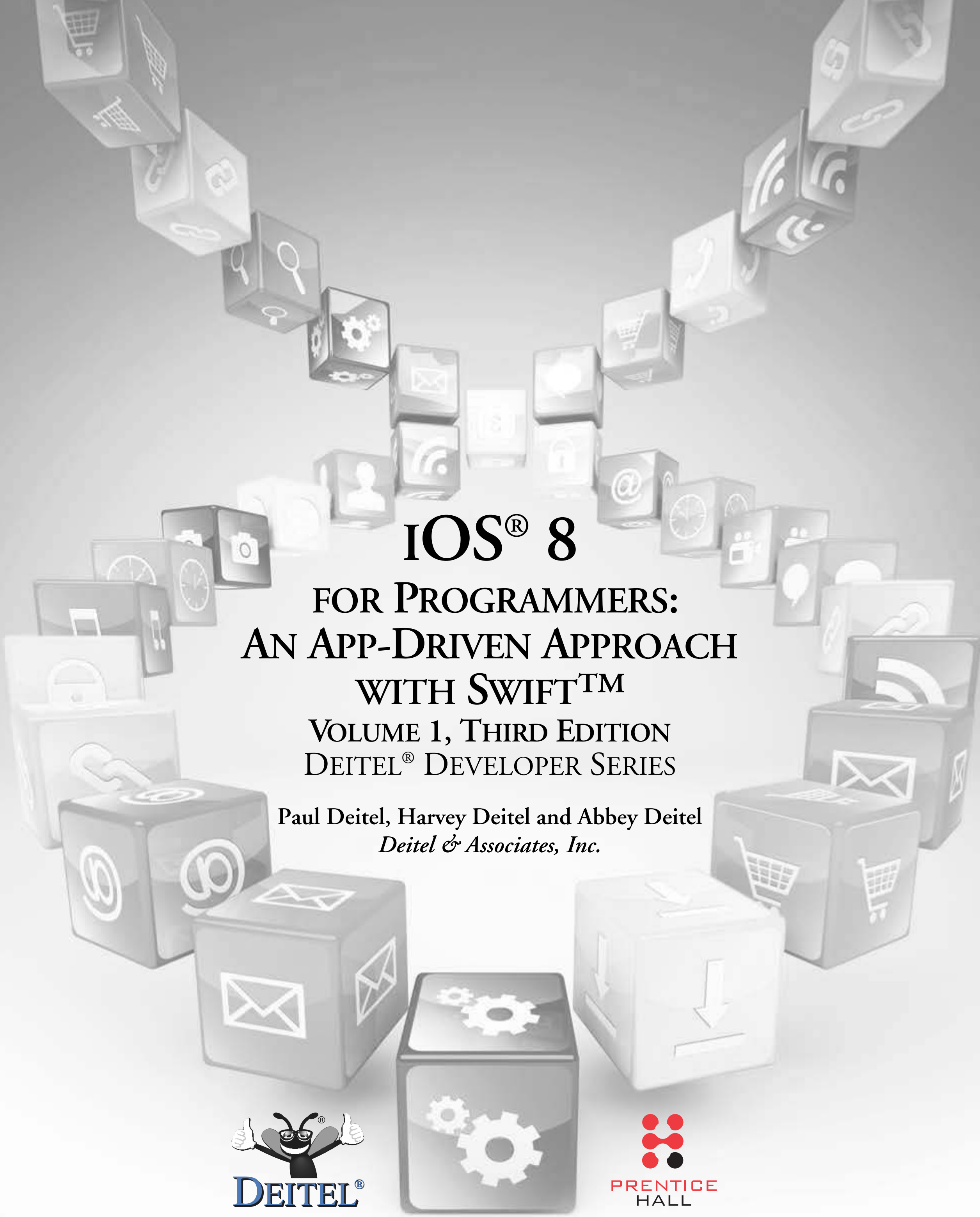
Visit us on the web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

On file

ISBN-13: 978-0-13396526-1
ISBN-10: 0-13-396526-0

Text printed in the United States at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, December 2014



iOS® 8 for Programmers: An App-Driven Approach with Swift™

Volume 1, Third Edition

Deitel® Developer Series

Paul Deitel, Harvey Deitel and Abbey Deitel
Deitel & Associates, Inc.

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

Deitel® Series Page

Deitel® Developer Series

Android for Programmers: An App-Driven Approach, 2/E, Volume 1
C for Programmers with an Introduction to C11
C++11 for Programmers
C# 2012 for Programmers
iOS® 8 for Programmers: An App-Driven Approach with Swift™
Java™ for Programmers, 3/E
JavaScript for Programmers

How To Program Series

Android How to Program, 2/E
C++ How to Program, 9/E
C How to Program, 7/E
Java™ How to Program, Early Objects Version, 10/E
Java™ How to Program, Late Objects Version, 10/E
Internet & World Wide Web How to Program, 5/E
Visual C++® 2008 How to Program, 2/E
Visual Basic® 2012 How to Program, 6/E
Visual C#® 2012 How to Program, 5/E

Simply Series

Simply C++: An App-Driven Tutorial Approach
Simply Java™ Programming: An App-Driven Tutorial Approach
(continued in next column)
(continued from previous column)
Simply C#: An App-Driven Tutorial Approach
Simply Visual Basic® 2010: An App-Driven Approach, 4/E

CourseSmart Web Books

`www.deitel.com/books/CourseSmart/`

C++ How to Program, 8/E and 9/E
Simply C++: An App-Driven Tutorial Approach
Java™ How to Program, 9/E and 10/E
Simply Visual Basic® 2010: An App-Driven Approach, 4/E
Visual Basic® 2012 How to Program, 6/E
Visual Basic® 2010 How to Program, 5/E
Visual C#® 2012 How to Program, 5/E
Visual C#® 2010 How to Program, 4/E

LiveLessons Video Learning Products

`www.deitel.com/books/LiveLessons/`

Android App Development Fundamentals, 2/e
C++ Fundamentals
Java™ Fundamentals, 2/e
C# 2012 Fundamentals
C# 2010 Fundamentals
iOS® 6 App Development Fundamentals
JavaScript Fundamentals
Swift Fundamentals

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please join the Deitel communities on

- Facebook®—`facebook.com/DeitelFan`
- Twitter®—`@deitel`
- Google+™—`google.com/+DeitelFan`
- YouTube™—`youtube.com/DeitelTV`
- LinkedIn®—`linkedin.com/company/deitel-&-associates`

and register for the free *Deitel® Buzz Online* e-mail newsletter at:

`www.deitel.com/newsletter/subscribe.html`

To communicate with the authors, send e-mail to:

`deitel@deitel.com`

For information on *Dive-Into® Series* on-site seminars offered by Deitel & Associates, Inc. worldwide, write to us at `deitel@deitel.com` or visit:

`www.deitel.com/training/`

For continuing updates on Pearson/Deitel publications visit:

`www.deitel.com`
`www.pearsonhighered.com/deitel/`

Visit the Deitel Resource Centers that will help you master programming languages, software development, Android and iOS app development, and Internet- and web-related topics:

`www.deitel.com/ResourceCenters.html`

In Memory of Amar G. Bose, MIT Professor and Founder and Chairman of the Bose Corporation:

It was a privilege being your student—and members of the next generation of Deitels, who heard our dad say how your classes inspired him to do his best work.

You taught us that if we go after the really hard problems, then great things can happen.

Harvey Deitel
Paul and Abbey Deitel

Trademarks

Contents

Preface

Welcome to the world of iOS® 8 app development with Apple's new and rapidly evolving Swift™ programming language, the Cocoa Touch® frameworks and the Xcode® 6 development tools.

iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 1, 3/e presents leading-edge mobile computing technologies for professional software developers. At the heart of the book is our *app-driven approach*—we present concepts in the context of *seven completely coded and fully tested iOS 8 apps* rather than using code snippets. We've always favored teaching by example—in an app-development world, the best examples are real, working apps.

Chapters 2–8 each present one app. We begin each of these chapters with an introduction to the app, an app test-drive showing one or more sample executions and a technologies overview. Then we proceed with a detailed source code walkthrough. We don't try to be exhaustive—our goal is to get you developing apps quickly with the Xcode 6 integrated development environment, the Swift programming language and the Cocoa Touch frameworks. All of the source code is available at

```
http://www.deitel.com/books/iOS8FP1
```

We recommend that you keep the code open in the IDE as you read the book. You should study the apps sequentially because each introduces technologies that are used in subsequent apps.

This book is Volume 1 of what will become a multi-volume set. Volume 1 presents seven fully coded apps of increasingly rich functionality. The apps cover a range of topics from simple visual programming (without code), to simple programming with Swift, to more involved programming.

Explosive Growth of the iPhone and iPad Is Creating Opportunity for Developers

iPhone and iPad device sales have been growing exponentially, creating significant opportunities for iOS app developers. The first-generation iPhone, released in June 2007, sold 6.1 million units in its initial five quarters of availability.[1] The iPhone 5s and the iPhone 5c, released simultaneously in September 2013, sold over nine million combined in the first three days of availability.[2] The most recent iPhone 6 and iPhone 6 Plus, announced in September 2014, pre-sold four million combined in just one day—double the number of

1. `http://www.apple.com/pr/library/2009/07/21results.html`.
2. `https://www.apple.com/pr/library/2013/09/23First-Weekend-iPhone-Sales-Top-Nine-Million-Sets-New-Record.html`.

iPhone 5 pre-sales in its first day of pre-order availability.[3] Apple sold 10 million iPhone 6 and iPhone 6 Plus units combined in their first weekend of availability.[4]

Sales of the iPad are equally impressive. The first generation iPad, launched in April 2010, sold 3 million units in its first 80 days of availability[5] and over 40 million worldwide by September 2011.[6] The iPad mini with Retina display (the second-generation iPad mini) and the iPad Air (the fifth-generation iPad) were released in November 2013. In just the first quarter of 2014, Apple sold a record 26 million iPads.[7]

There are over 1.3 million apps in the App Store[8] and over 75 billion iOS apps have been downloaded.[9] The potential for iOS app developers is enormous.

SafariBooksOnline e-Book and LiveLessons Videos

If you have a subscription to Safari Books Online (`www.safaribooksonline.com`), check out the e-book and LiveLessons video versions of *iOS® 8 for Programmers: An App-Driven Approach with Swift*. Safari is a subscription service popular with large companies, colleges, libraries and individuals who would like access to video training and electronic versions of print publications.

Copyright Notice and Code License

All of the code and iOS apps in the book are copyrighted by Deitel & Associates, Inc. The sample iOS apps are licensed under a Creative Commons Attribution 3.0 Unported License (`http://creativecommons.org/licenses/by/3.0`*),* ***with the exception that they may not be reused in any way in educational tutorials and textbooks, whether free or for a fee and whether in print or digital format****. Additionally, the authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs. You're welcome to use the apps in the book as shells for your own apps, building on their existing functionality. If you have any questions, contact us at* `deitel@deitel.com`*.*

Intended Audience

This book is part of the *Deitel Developer Series* intended for experienced programmers who know object-oriented programming in a C-based programming language such as Objective-C, Java, C# or C++. Objective-C experience is helpful, but not specifically required. If you have not worked in any of these languages, you should still be able to learn a good amount of iOS 8 app development and object-oriented programming in Swift and Cocoa

3. `http://techcrunch.com/2014/09/15/apple-sells-4m-iphone-6-and-6-plus-pre-orders-in-opening-24-hours/`.
4. `http://www.apple.com/pr/library/2014/09/22First-Weekend-iPhone-Sales-Top-10-Million-Set-New-Record.html`.
5. `http://www.ipadinsider.com/tag/ipad-sales-figures/`.
6. `http://www.statista.com/statistics/180656/sales-of-tablets-and-ipads-in-the-us-until-2012/`.
7. `http://www.theverge.com/2014/1/27/5350106/apple-q1-2014-earnings`.
8. `http://mashable.com/2014/09/09/apple-1-3-million-apps-app-store/`.
9. `http://techcrunch.com/2014/06/02/itunes-app-store-now-has-1-2-million-apps-has-seen-75-billion-downloads-to-date/`.

Touch by reading the code and our code walkthroughs, running the apps and observing the results. We review the basics of object-oriented programming in Chapter 1. We also assume that you're comfortable with OS X, as you'll need to work on a Mac to develop iOS apps. The book does not include exercises.

This book is *not* a Swift tutorial, but it presents a significant amount of Swift in the context of iOS 8 app development. If you're interested in learning Swift, check out our publications:

- *Swift for Programmers* print book (`www.deitel.com/books/swiftfp`). This book is also available as an e-book on `SafariBooksOnline.com`, `Informit.com`, Amazon® Kindle® and a growing number of other electronic platforms.
- *Swift Fundamentals: Parts I, II and III* LiveLessons videos (`www.deitel.com/books/LiveLessons`), available on `SafariBooksOnline.com`, `Informit.com`, `Udemy.com` and soon on other popular e-learning platforms.

Academic Bundle iOS® 8 for Programmers and Swift™ for Programmers

The *Academic Bundle iOS® 8 for Programmers and Swift™ for Programmers* is designed for professionals, students and instructors interested in learning or teaching iOS 8® app development with a broader and deeper treatment of Swift. You can conveniently order the Academic Bundle with one ISBN: 0-13-408775-5. The Academic Bundle includes:

- *Swift™ for Programmers* (print book)
- *iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 1, 3/e* (print book)
- Access Code Card for Academic Package to accompany *Swift™ for Programmers*
- Access Code Card for Academic Package to accompany *iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 1, 3/e*

The two Access Code Cards for the Academic Packages (when used together) give you access to the companion websites, which include self-review questions (with answers), short-answer questions, programming exercises, programming projects and selected videos chosen to get you up to speed quickly with Xcode 6, visual programming and basic Swift-based, iOS 8 programming.

Ordering the Books and Supplements Separately

The print books and Access Code Cards may be purchased separately using the following ISBNs:

- *Swift™ for Programmers* (print book): ISBN 0-13-402136-3
- Standalone access code card for Academic Package to accompany *Swift™ for Programmers*: ISBN 0-13-405818-6
- *iOS® 8 for Programmers: An App-Driven Approach with Swift™* (print book): ISBN 0-13-396526-0
- Standalone access code card for Academic Package to accompany *iOS® 8 for Programmers: An App Driven Approach with Swift™, Volume 1, 3/e*: ISBN 0-13-405825-9

Instructor Supplements

Instructor supplements are available online at Pearson's Instructor Resource Center IRC). The supplements include:

- Solutions Manual with selected solutions to the short-answer exercises.
- Test Item File of multiple-choice examination questions (with answers).
- PowerPoint® slides with the book's source code and tables.

Please do not write to us requesting access to the Pearson Instructor's Resource Center. Certified instructors who adopt the book for their courses can obtain password access from their regular Pearson sales representatives (`www.pearson.com/replocator`). Solutions are *not* provided for "project" exercises.

Key Features of *iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 1, 3/e*

Here are some of this book's key features:

App-Driven Approach. Chapters 2–8 each present one completely coded app—we discuss what the app does, show screen shots of the app in action, test-drive it and overview the technologies and architecture we'll use to build it. Then we build the app's GUI and resource files, present the complete code and do a detailed code walkthrough. We discuss the Swift programming concepts and demonstrate the functionality of the Cocoa Touch APIs used in the app.

Swift Programming Language. Swift was arguably the most significant announcement at Apple's Worldwide Developers Conference in 2014. Although apps can still be programmed in Objective-C, Swift is Apple's language of the future for app development and systems programming.

We've programmed all the book's apps in Swift—previous editions were programmed in Objective-C. Swift is a contemporary language with simpler syntax than Objective-C. It enables a clean, concise coding style and has a strong focus on error prevention. Our own experience with Swift has been that we can develop apps faster and with significantly less code than when we program in Objective-C.

At the time of this writing, Apple had not as yet published coding guidelines for Swift—we'll conform to them when they appear. We use a mix of Apple's Objective-C coding guidelines and Deitel coding guidelines for this edition.

Cocoa Touch Frameworks. Cocoa Touch is the groups of reusable components (known as frameworks) for building iOS apps. Throughout this edition, we use many of the Cocoa Touch features and frameworks, even though they're programmed mostly in Objective-C. Apple has made this easy with a technique called "bridging." We simply call Cocoa Touch methods and receive the returns *transparently*—it feels as if Cocoa Touch is written in Swift.

iOS SDK 8. Between Volumes 1 and 2 of *iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 1, 3/e*, we cover a broad range of the features included in iOS Software Development Kit (SDK) 8.

Xcode 6. Apple's Xcode integrated development environment (IDE) and its associated tools for Mac OS X, combined with the iOS 8 Software Development Kit (SDK), provide all the software you need to develop and test iOS 8 apps.

Instruments. The Instruments tool, which is packaged with the SDK, is used to inspect apps while they're running to check for memory leaks, monitor processor (CPU) usage and network activity, and review the objects allocated in memory.

iOS Human Interface Guidelines. We encourage you to read Apple's *iOS Human Interface Guidelines* (HIG) and follow them as you design and develop your apps. The HIG discusses human interface principles, app design strategies, user experience guidelines, iOS technology usage guidelines and more. We gradually introduce HIG issues as we encounter them in the apps we develop. Section 9.3 overviews the HIG, discusses features and functionality required to get your app accepted on the App Store and lists reasons why Apple rejects apps.

Multimedia. The apps use iOS 8 multimedia capabilities, including graphics, images, animation and audio. We'll present video capabilities in Volume 2.

iOS App Design Patterns. This book adheres to Apple's app coding standards, including design patterns, such as Model-View-Controller (MVC), Delegation, Target-Action and Observer.

Features

Syntax Coloring. For readability, we syntax color the code, similar to Xcode's use of syntax coloring. Our syntax-coloring conventions are as follows:

```
comments appear in green
keywords appear in blue
constants and literal values appear in light blue
all other code appears in black
```

Code Highlighting. We highlight the key code segments in each app that exercise the new technologies the app features.

Using Fonts for Emphasis. We place key terms and the index's page reference for each term's defining occurrence in **bold maroon** text for easier reference. We emphasize on-screen components in the **bold Helvetica** font (e.g., the **File** menu) and emphasize Swift program text in the `Lucida` font (for example, `var x = 5`).

Source Code. All of the source-code examples are available for download from:

```
http://www.deitel.com/books/iOS8FP1/
```

Documentation. All of the manuals that you'll need to develop iOS 8 apps are available free at `http://developer.apple.com/ios`.

Chapter Objectives. Each chapter begins with a list of objectives.

Figures. Abundant tables, source-code listings and iOS screen shots are included.

Index. We include an extensive index, which is especially useful when you use the book as a reference. Defining occurrences of key terms are highlighted with a **bold** page number.

iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 2

Volume 2 of this series will contain additional app-development chapters. **For the status of Volume 2 and for continuing book updates, visit**

```
http://www.deitel.com/books/iOS8fp2
```

iOS® 8 Fundamentals LiveLessons Video Training Products

Our *iOS 8 Fundamentals* LiveLessons videos show you what you need to know to start building robust, powerful iOS apps with the iOS Software Development Kit (SDK) 8, the Swift programming language, Xcode and Cocoa Touch. It will include approximately 10+ hours of expert training synchronized with *iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 1, 3/e*. For additional information about Deitel LiveLessons video products, visit

```
www.deitel.com/livelessons
```

or contact us at `deitel@deitel.com`. You can also access our LiveLessons videos if you have a subscription to Safari Books Online (`www.safaribooksonline.com`). You can get a free 10-day subscription to SafariBooksOnline at

```
http://www.safaribooksonline.com/register
```

Acknowledgments

We'd like to thank Barbara Deitel for long hours spent researching iOS 8 and its many related technologies.

Pearson Education Team

We're fortunate to have worked on this project with the dedicated publishing professionals at Prentice Hall/Pearson. We appreciate the extraordinary efforts and 19-year mentorship of our friend and professional colleague Mark L. Taub, Editor-in-Chief of Pearson Technology Group. Kim Boedigheimer recruited distinguished members of the iOS community to review the manuscript and she managed the review process. We selected the cover art and Chuti Prasertsith designed the cover. John Fuller managed the book's publication.

Reviewers

We wish to acknowledge the efforts of our current and recent editions reviewers. They scrutinized the text and the programs and provided countless suggestions for improving the presentation.

iOS 8 edition reviewers: Scott Bossak (Lead iOS Developer, Thrillist Media Group), Charles E. Brown (Independent Contractor affiliated with Apple and Adobe), Matt Galloway (iOS Developer and author of *Effective Objective-C 2.0*), Michael Haberman (Software Engineer, Instructor at University of Illinois), Rob McGovern (Indie Developer) and Rik Watson (Technical Team Lead, HP Enterprise Services).

Earlier iOS editions reviewers: Cory Bohon (Indie Developer at `CocoaApp.com` and Writer at Mac|Life), Scott Gustafson (Owner/Developer, Garlic Software LLC), Firoze Lafeer (Master Developer, Capital One Labs), Dan Lingman (Partner, `www.nogotog-`

ames.com), Marcantonio Magnarapa (Chief Mobile Officer, www.bemyeye.com), Nik Saers (iOS Developer, SAERS), Zach Saul (Founder, Retronyms) and Rik Watson (then a Senior Software Engineer, Lockheed Martin).

Keeping in Touch with the Authors

As you read the book, we'd appreciate your comments, criticisms, corrections and suggestions for improvement. Please address all correspondence to:

```
deitel@deitel.com
```

We'll respond promptly. For updates on this book, visit

```
http://www.deitel.com/books/iOS8FP1
```

subscribe to the *Deitel® Buzz Online* newsletter at

```
http://www.deitel.com/newsletter/subscribe.html
```

and join the Deitel social networking communities on

- Facebook® (`http://www.deitel.com/deitelfan`)
- Twitter® (`@deitel`)
- LinkedIn® (`http://linkedin.com/company/deitel-&-associates`)
- Google+™ (`http://google.com/+DeitelFan`)
- YouTube® (`http://youtube.com/DeitelTV`)

Well, there you have it! We hope you enjoy working with *iOS® 8 for Programmers: An App-Driven Approach with Swift, Volume 1* as much as we enjoyed writing it!

Paul, Harvey and Abbey Deitel

About the Authors

Paul Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. He holds the Java Certified Programmer and Java Certified Developer designations, and is an Oracle Java Champion. Paul was also named as a Microsoft® Most Valuable Professional (MVP) for C# in 2012–2014. Through Deitel & Associates, Inc., he has delivered hundreds of programming courses worldwide to clients, including Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey Deitel, are the world's best-selling programming-language textbook/professional book/video authors.

Dr. Harvey Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has over 50 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul. The Deitels' publications have earned international recognition, with translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, academic, government and military clients.

Abbey Deitel, President of Deitel & Associates, Inc., is a graduate of Carnegie Mellon University's Tepper School of Management where she received a B.S. in Industrial Management. Abbey has been managing the business operations of Deitel & Associates, Inc. for 17 years. She has contributed to numerous Deitel & Associates publications including *Swift™ for Programmers* and, together with Paul and Harvey, is the co-author of *iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 1, 3/e, Android for Programmers: An App-Driven Approach, 2/e, Internet & World Wide Web How to Program, 5/e, Visual Basic 2012 How to Program, 6/e* and *Simply Visual Basic 2010, 5/e*.

About Deitel® & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in mobile app development, computer programming languages, object technology and Internet and web software technology. The company's training clients include many of the world's largest companies, government agencies, branches of the military, and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including Swift™, Objective-C and iOS® app development, Java™, Android app development, C++, C, Visual C#®, Visual Basic®, Python®, object technology, Internet and web programming and a growing list of additional programming and software development courses.

Through its 40-year publishing partnership with Pearson/Prentice Hall, Deitel & Associates, Inc., publishes leading-edge programming textbooks and professional books in print and a wide range of e-book formats, and *LiveLessons* video courses. Deitel & Associates, Inc. and the authors can be reached at:

```
deitel@deitel.com
```

To learn more about Deitel's *Dive-Into® Series* Corporate Training curriculum, visit:

```
http://www.deitel.com/training
```

To request a proposal for worldwide on-site, instructor-led training at your organization, send an e-mail to `deitel@deitel.com`.

Individuals wishing to purchase Deitel books and *LiveLessons* video training can do so through `www.deitel.com`. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

```
http://www.informit.com/store/sales.aspx
```

Before You Begin

This section contains information you should review before using this book. Updates will be posted at:

```
http://www.deitel.com/books/iOS8FP1
```

Font and Naming Conventions

We use fonts to distinguish between on-screen components (such as menu names and menu items) and Swift code. Our convention is to emphasize on-screen components in a sans-serif bold **Helvetica** font (for example, **File** menu) and to emphasize Swift code and commands in a sans-serif `Lucida` font (for example, `import UIKit`). When building user interfaces (UIs) using Xcode's Interface Builder, we also use the bold **Helvetica** font to refer to property names for UI components (such as a **Label**'s **Text** property).

Conventions for Referencing Menu Items in a Menu

We use the **>** character to indicate selecting a menu item from a menu. The notation **File > Open…** indicates that you should select the **Open…** menu item from the **File** menu.

Software Used in this Book

To execute our apps and write your own iOS 8 apps, you must install Xcode 6. You can install the currently released Xcode version for free from the Mac App Store. When you open Xcode for the first time, it will download and install additional features required for development. For the latest information about Xcode, visit

```
https://developer.apple.com/xcode
```

A Note Regarding the Xcode 6 Toolbar Icons

We developed this book's examples with Xcode 6 on OS X Yosemite. If you're running OS X Mavericks, some Xcode toolbar icons we show in the text may differ on your screen.

Becoming a Registered Apple Developer

Registered developers have access to the online iOS documentation and other resources. Apple also now makes Xcode pre-release versions (such as the next point release or major version) available to all registered Apple developers. To register, visit:

```
https://developer.apple.com/register
```

To download the next pre-release Xcode version, visit:

```
https://developer.apple.com/xcode/downloads
```

Once you download the DMG (disk image) file, double click it to launch the installer, then follow the on-screen instructions.

Fee-Based Developer Programs

iOS Developer Program

The fee-based **iOS Developer Program** allows you to load your iOS apps onto iOS devices for testing and to submit your apps to the App Store. If you intend to distribute iOS apps, you'll need to join the fee-based program. You can sign up at

```
https://developer.apple.com/programs
```

iOS Developer Enterprise Program

Organizations may register for the iOS Developer Enterprise Program at

```
https://developer.apple.com/programs/ios/enterprise
```

which enables developers to deploy proprietary iOS apps to employees within their organization.

iOS Developer University Program

Colleges and universities interested in offering iOS app-development courses can apply to the iOS Developer University Program at

```
https://developer.apple.com/programs/ios/university
```

Qualifying schools receive free access to all the developer tools and resources. Students can share their apps with each other and test them on iOS devices.

Adding Your Paid iOS Developer Program Account to Xcode

Xcode can interact with your paid iOS Developer Program account on your behalf so that you can install apps onto your iOS devices for testing. If you have a paid iOS Developer Program account, you can add it to Xcode. To do so:

1. Select **Xcode > Preferences…**.
2. In the **Accounts** tab, click the **+** button in the lower left corner and select **Add Apple ID…**.
3. Enter your Apple ID and password, then click **Add**.

Obtaining the Code Examples

The final versions of the apps you'll build in this book are available for download as a ZIP file from

```
http://www.deitel.com/books/iOS8FP1
```

under the heading **Download Code Examples and Other Premium Content**. When you click the link to the ZIP file, it will be placed by default in your user account's `Downloads` folder. We assume that the examples are located in the `iOS8Examples` folder in your user account's `Documents` folder. You can use Finder to move the ZIP file there, then double click the file to extract its contents.

Xcode Projects

For each app, we provide a project that you can open in Xcode by double clicking its project file, which has the `.xcodeproj` extension. You'll use these projects to test-drive the apps before building them.

Configuring Xcode to Display Line Numbers

Many programmers find it helpful to display line numbers in the code editor. To do so:

1. Open Xcode and select **Preferences…** from the **Xcode** menu.
2. Select the **Text Editing** tab, then ensure that the **Editing** subtab is selected.
3. Check the **Line Numbers** checkbox.

Configuring Xcode's Code Indentation Options

Xcode uses four space indents by default. To configure your own indentation preferences:

1. Open Xcode and select **Preferences…** from the **Xcode** menu.
2. Select the **Text Editing** tab, then ensure that the **Indentation** subtab is selected.
3. Specify your indentation preferences.

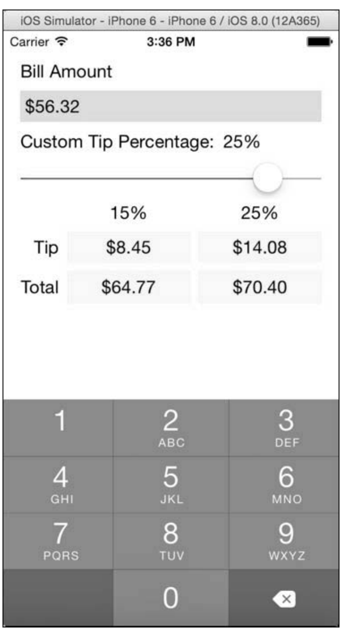
You're now ready to begin working with *iOS® 8 for Programmers: An App-Driven Approach with Swift™, Volume 1, 3/e*. We hope you enjoy the book! If you have any questions, please email us at `deitel@deitel.com`.

This page intentionally left blank

3

Tip Calculator App

Introducing Swift, Text Fields, Sliders, Outlets, Actions, View Controllers, Event Handling, `NSDecimalNumber`, `NSNumberFormatter` and Automatic Reference Counting



Objectives

In this chapter you'll:

- Learn basic Swift syntax, keywords and operators.
- Use object-oriented Swift features, including objects, classes, inheritance, functions, methods and properties.
- Use `NSDecimalNumber`s to perform precise monetary calculations.
- Create locale-specific currency and percentage `String`s with `NSNumberFormatter`.
- Use **Text Field**s and **Sliders** to receive user input.
- Programmatically manipulate UI components via outlets.
- Respond to user-interface events with actions.
- Understand the basics of automatic reference counting (ARC).
- Execute an interactive iOS app.

Outline

3.1 Introduction

The **Tip Calculator** app (Fig. 3.1(a))—which you test-drove in Section 1.17—calculates and displays possible tips and bill totals for a restaurant bill amount. As you enter each digit of an amount by touching the *numeric keypad*, the app calculates and displays the tip amount and total bill amount for a 15% tip and a custom tip (Fig. 3.1(b)). You specify the custom tip percentage by moving a **Slider**'s *thumb*—this updates the custom tip percentage **Label**s and displays the custom tip and bill total in the righthand column of yellow **Label**s below the **Slider** (Fig. 3.1(b). We chose 18% as the default custom percentage, because many restaurants in the U.S. add this tip percentage for parties of six people or more, but you can easily change this.

First, we'll overview the technologies used to build the app. Next, you'll build the app's UI using Interface Builder. As you'll see, Interface Builder's visual tools can be used to connect UI components to the app's code so that you can manipulate the corresponding UI components programmatically and respond to user interactions with them.

For this app, you'll write Swift code that responds to user interactions and programmatically updates the UI. You'll use Swift object-oriented programming capabilities, including objects, classes, inheritance, methods and properties, as well as various data types, operators, control statements and keywords. With our *app-driven approach*, we'll present the app's complete source code and do a detailed code walkthrough, introducing the Swift language features as we encounter them.

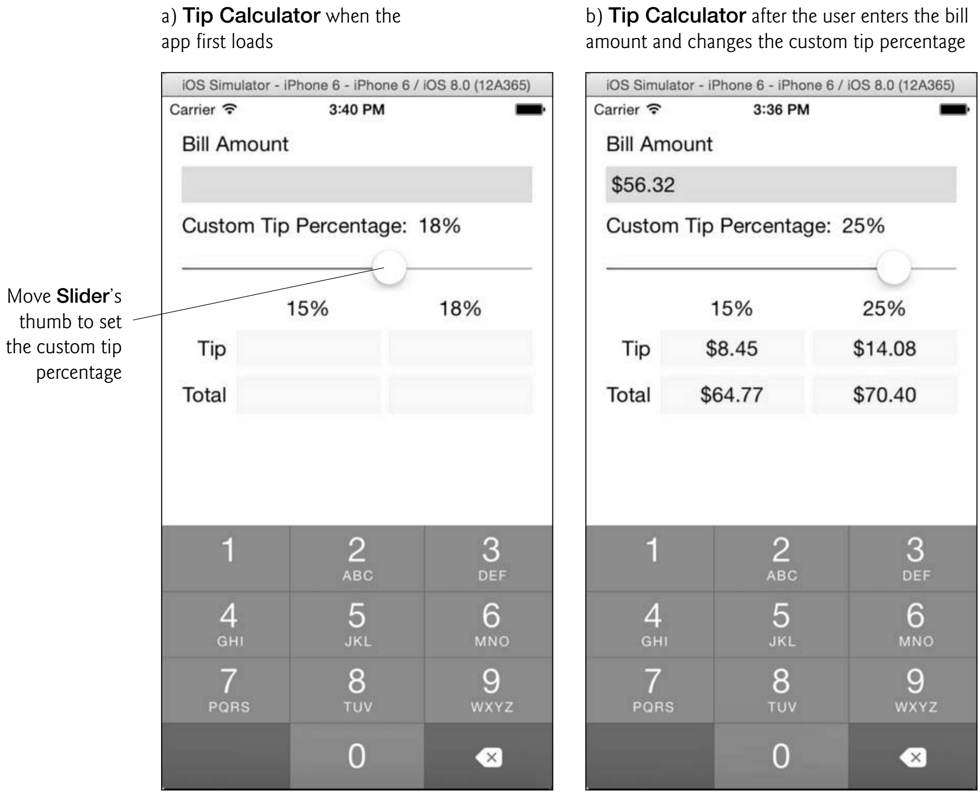


Fig. 3.1 | **Tip Calculator** when the app first loads, then after the user enters the bill amount and changes the custom tip percentage.

3.2 Technologies Overview

This section introduces the Xcode, Interface Builder and Swift features you'll use to build the **Tip Calculator** app.

3.2.1 Swift Programming

Swift is Apple's programming language of the future for iOS and OS X development. The app's code uses Swift data types, operators, control statements and keywords, and other language features, including functions, overloaded operators, type inference, variables, constants and more. We'll introduce Swift object-oriented programming features, including objects, classes, inheritance, methods and properties. We'll explain each new Swift feature as we encounter it in the context of the app. Swift is based on many of today's popular programming languages, so much of the syntax will be familiar to programmers who use C-based programming languages, such as Objective-C, Java, C# and C++. For a detailed introduction to Swift, visit:

```
https://developer.apple.com/library/ios/documentation/Swift/
   Conceptual/Swift_Programming_Language/
```

3.2.2 Swift Apps and the Cocoa Touch® Frameworks

A great strength of iOS 8 is its rich set of prebuilt components that you can *reuse* rather than "reinventing the wheel." These capabilities are grouped into iOS's **Cocoa Touch frameworks**. These powerful libraries help you create apps that meet Apple's requirements for the look-and-feel of iOS apps. The frameworks are written mainly in Objective-C (some are written in C). Apple has indicated that new frameworks will be developed in Swift.

Foundation Framework

The **Foundation** framework includes classes for basic types, storing data, working with text and strings, file-system access, calculating differences in dates and times, inter-app notifications and much more. In this app, you'll use Foundation's `NSDecimalNumber` and `NSNumberFormatter` classes. Foundation's class names begin with the prefix `NS`, because this framework originated in the NextStep operating system. Throughout the book, we'll use many Foundation framework features—for more information, visit:

```
http://bit.ly/iOSFoundationFramework
```

UIKit Framework

Cocoa Touch's **UIKit** framework includes multi-touch UI components appropriate for mobile apps, event handling (that is, responding to user interactions with the UI) and more. You'll use many UIKit features throughout this book.

Other Cocoa Touch Frameworks

Figure 3.2 lists the Cocoa Touch frameworks. You'll learn features from many of these frameworks in this book and in *iOS 8 for Programmers: An App-Driven Approach, Volume 2*. For more information on these frameworks, see the *iOS Developer Library Reference* (`http://developer.apple.com/ios`).

List of Cocoa Touch frameworks

Cocoa Touch Layer	AssetsLibrary	OpenAL	CoreLocation	Social
AddressBookUI	AudioToolbox	OpenGLES	CoreMedia	StoreKit
EventKitUI	AudioUnit	Photos	CoreMotion	SystemConfiguration
GameKit	CoreAudio	QuartzCore	CoreTelephony	UIAutomation
MapKit	CoreGraphics	SceneKit	EventKit	WebKit
MessageUI	CoreImage	SpriteKit	Foundation	
NotificationCenter	CoreMIDI		HealthKit	***Core OS Layer***
PhotosUI	CoreText	***Core Services Layer***	HomeKit	Accelerate
Twitter	CoreVideo	Accounts	JavaScriptCore	CoreBluetooth
UIKit	GLKit	AdSupport	MobileCoreServices	ExternalAccessory
iAd	GameController	AddressBook	MultipeerConnectivity	LocalAuthentication
	ImageIO	CFNetwork	NewsstandKit	Security
Media Layer	MediaAccessibility	CloudKit	PassKit	System
AVFoundation	MediaPlayer	CoreData	QuickLook	
	Metal	CoreFoundation		

Fig. 3.2 | List of Cocoa Touch frameworks.

3.2.3 Using the UIKit and Foundation Frameworks in Swift Code

To use UIKit framework classes (or classes from any other existing framework), you must **`import`** the framework into each source-code file that uses it (as we do in Section 3.6.1). This exposes the framework's capabilities so that you can access them in Swift code. In addition to UIKit framework UI components, this app also uses various classes from the Foundation framework, such as `NSDecimalNumber` and `NSNumberFormatter`. We do not `import` the Foundation framework—its features are available to your code because the UIKit framework indirectly imports the Foundation framework.

3.2.4 Creating Labels, a Text Field and a Slider with Interface Builder

You'll again use Interface Builder and auto layout to design this app's UI, which consists of **Label**s for displaying information, a **Slider** for selecting a custom tip percentage and a **Text Field** for receiving the user input. Several **Label**s are configured identically—we'll show how to duplicate components in Interface Builder, so you can build UIs faster. **Label**s, the **Slider** and the **Text Field** are objects of classes **`UILabel`**, **`UISlider`** and **`UITextField`**, respectively, and are part the UIKit framework that's included with each app project you create.

3.2.5 View Controllers

Each *scene* you define is managed by a **view controller** object that determines what information is displayed. iPad apps sometimes use multiple view controllers in one scene to make better use of the larger screen size. Each scene represents a *view* that contains the UI components displayed on the screen. The view controller also specifies how user interactions with the scene are processed. Class **`UIViewController`** defines the basic view controller capabilities. Each view controller you create (or that's created when you base a new app on one of Xcode's app templates) inherits from `UIViewController` or one of its subclasses. In this app, Xcode creates the class `ViewController` to manage the app's scene, and you'll place additional code into that class to implement the **Tip Calculator**'s logic.

3.2.6 Linking UI Components to Your Swift Code

Properties

You'll use Interface Builder to generate *properties* in your view controller for programmatically interacting with the app's UI components. Swift classes may contain variable properties and constant properties. Variable properties are read/write and are declared with the **`var`** keyword. Constant properties, which cannot be modified after they're initialized, are read-only and are declared with **`let`**. These keywords can also be used to declare local and global variables and constants. A variable property defines a *getter* and a *setter* that allow you to obtain and modify a property's value, respectively. A constant property defines only a *getter* for obtaining its value.

`@IBOutlet` Properties

Each property for programmatically interacting with a UI component is prefixed with **`@IBOutlet`**. This tells Interface Builder that the property is an **outlet**. You'll use Interface Builder to *connect* a UI control to its corresponding outlet in the view controller using *drag-and-drop* techniques. Once connected, the view controller can manipulate the corresponding UI component programmatically. `@IBOutlet` properties are *variable* properties so they can be modified to refer to the UI controls when the storyboard creates them.

Action Methods

When you interact with a UI component (e.g., touching a **Slider** or entering text in a **Text Field**), a user-interface *event* occurs. The view controller handles the event with an **action**—an *event-handling method* that specifies what to do when the event occurs. Each action is annotated with **`@IBAction`** in your view controller's class. `@IBAction` indicates to Interface Builder that a method can respond to user interactions with UI components. You'll use Interface Builder to visually *connect* an action to a specific user-interface event using *drag-and-drop* techniques.

3.2.7 Performing Tasks After a View Loads

When a user launches the **Tip Calculator**:

- Its main storyboard is loaded.
- The UI components are created.
- An object of the app's initial view controller class is instantiated.
- Using information stored in the storyboard, the view controller's `@IBOutlets` and `@IBActions` are connected to the appropriate UI components.

In this app, we have only one view-controller, because the app has only one scene. After all of the storyboard's objects are created, iOS calls the view controller's **`viewDidLoad`** method—here you perform view-specific tasks that can execute only *after* the scene's UI components exits. For example, in this app, you'll call the method **`becomeFirstResponder`** on the `UITextField` to make it the active component—as if the user touched it. You'll configure the `UITextField` such that when it's the *active* component, the numeric keypad is displayed in the screen's lower half. Calling `becomeFirstResponder` from `viewDidLoad` causes iOS to display the keypad immediately after the view loads. (Keypads are *not* displayed if a Bluetooth keyboard is connected to the device.) Calling this method also indicates that the `UITextField` is the **first responder**—the first component that will receive notification when an event occurs. iOS's **responder chain** defines the order in which components are notified that an event occurred. For the complete responder chain details, visit:

```
http://bit.ly/iOSResponderChain
```

3.2.8 Financial Calculations with NSDecimalNumber

Financial calculations performed with Swift's `Float` and `Double` numeric types tend to be inaccurate due to rounding errors. For precise floating-point calculations, you should instead use objects of the Foundation framework class **`NSDecimalNumber`**. This class provides various methods for creating `NSDecimalNumber` objects and for performing arithmetic calculations with them. This app uses the class's methods to perform division, multiplication and addition.

Swift Numeric Types

Though this app's calculations use only `NSDecimalNumbers`, Swift has its own numeric types, which are defined in the Swift Standard Library. Figure 3.3 shows Swift's numeric and boolean types—each type name begins with a capital letter. For the integer types, each type's minimum and maximum values can be determined with its **`min`** and **`max`** properties—for example, `Int.min` and `Int.max` for type `Int`.

Type	Description
Integer types	
`Int`	Default signed integer type—4 or 8 bytes depending on the platform.
`Int8`	8-bit (1-byte) signed integer. Values in the range –128 to 127.
`Int16`	16-bit (2-byte) signed integer. Values in the range –32,768 to 32767.
`Int32`	32-bit (4-byte) signed integer. Values in the range –2,147,483,648 to 2,147,483,647.
`Int64`	64-bit (8-byte) signed integer. Values in the range –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
`UInt8`	8-bit (1-byte) unsigned integer. Values in the range 0 to 255.
`UInt16`	16-bit (2-byte) unsigned integer. Values in the range 0 to 65,535.
`UInt32`	32-bit (4-byte) unsigned integer. Values in the range 0 to 4,294,967,295.
`UInt64`	64-bit (8-byte) unsigned integer. Values in the range 0 to 18,446,744,073,709,551,615.
Floating-point types (conforms to IEEE 754)	
`Float`	4-byte floating-point value. *Negative range:* –3.4028234663852886e+38 to –1.40129846432481707e–45 *Positive range:* 1.40129846432481707e–45 to 3.4028234663852886e+38
`Double`	8-byte floating-point value. *Negative range:* –1.7976931348623157e+308 to –4.94065645841246544e–324 *Positive range:* 4.94065645841246544e–324 to 1.7976931348623157e+308
Boolean type	
`Bool`	`true` or `false` values.

Fig. 3.3 | Swift numeric and boolean types.

Swift also supports standard arithmetic operators for use with the numeric types in Fig. 3.3. The standard arithmetic operators are shown in Fig. 3.4.

Operation	Operator	Algebraic expression	Swift expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	$b \cdot m$	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	`%`	$r \bmod s$	`r % s`

Fig. 3.4 | Arithmetic operators in Swift.

3.2.9 Formatting Numbers as Locale-Specific Currency and Percentage Strings

You'll use Foundation framework class **`NSNumberFormatter`**'s **`localizedStringFromNumber`** method to create locale-specific currency and percentage strings—an important part of internationalization. You could also add accessibility strings and internationalize the app using the techniques you learned in Sections 2.7–2.8.

3.2.10 Bridging Between Swift and Objective-C Types

You'll often pass Swift objects into methods of classes written in Objective-C, such as those in the Cocoa Touch classes. Swift's numeric types and its `String`, `Array` and `Dictionary` types can all be used in contexts where their Objective-C equivalents are expected. Similarly, the Objective-C equivalents (`NSString`, `NSArray`, `NSMutableArray`, `NSDictionary` and `NSMutableDictionary`), when returned to your Swift code, are automatically treated as their Swift counterparts. In this app, for example, you'll use class `NSNumberFormatter` to create locale-specific currency and percentage strings. These are returned from `NSNumberFormatter`'s methods as `NSString` objects, but are automatically treated by Swift as objects of Swift's type `String`. This mechanism—known as **bridging**—is transparent to you. In fact, when you look at the Swift version of the Cocoa Touch documentation online or in Xcode, you'll see the Swift types, not the Objective-C types for cases in which this bridging occurs.

3.2.11 Swift Operator Overloading

Swift allows operator overloading—you can define your own operators for use with existing types. In Section 3.6.7, we'll define overloaded addition, multiplication and division operators to simplify the `NSDecimalNumber` arithmetic performed throughout the app's logic. As you'll see, you define an overloaded operator by creating a Swift function, but with an operator *symbol* as its name and a parameter list containing parameters that represent each operand. So, for example, you'd provide two parameters for an overloaded-operator function that defines an addition (+) *binary* operator—one for each operand.

3.2.12 Variable Initialization and Swift Optional Types

In Swift, every constant and variable you create (including a class's properties) must be initialized (or for variables, assigned to) before it's used in the code; otherwise, a compilation error occurs. A problem with this requirement occurs when you create `@IBOutlet` properties in a view controller using Interface Builder's drag-and-drop techniques. Such properties refer to objects that are not created in your code. Rather, they're created by the *storyboard* when the app executes, then the storyboard *connects* them to the view controller—that is, the storyboard assigns each UI component object to the appropriate property so that you can programmatically interact with that component.

For scenarios like this in which a variable receives its value at runtime, Swift provides **optional types** that can indicate the presence or absence of a value. A variable of an optional type can be initialized with the value **`nil`**, which indicates the *absence* of a value.

When you create an `@IBOutlet` with Interface Builder, it declares the property as an **implicitly unwrapped optional** type by following the type name with an exclamation point (`!`). Properties of such types are initialized *by default* to `nil`. Such properties must be declared as variables (with `var`) so that they can *eventually* be assigned actual values of the specified type. Using optionals like this enables your code to compile because the `@IBOutlet` properties *are*, in fact, initialized—just not to the values they'll have at runtime.

As you'll see in later chapters, Swift has various language features for testing whether an optional has a value and, if so, *unwrapping* the value so that you can use it—known as explicit unwrapping. With implicitly unwrapped optionals (like the `@IBOutlet` properties), you can simply assume that they're initialized and use them in your code. If an implicitly unwrapped optional is `nil` when you use it, a runtime error occurs. Also, an optional can be set to `nil` at any time to indicate that it no longer contains a value.

3.2.13 Value Types vs. Reference Types

Swift's types are either **value types** or **reference types**. Swift's numeric types, `Bool` type and `String` type are all values types.

Value Types

A value-type constant's or variable's value is *copied* when it's passed to or returned from a function or method, when it's assigned to another variable or when it's used to initialize a constant. Note that Swift's `String`s are value types—in most other object-oriented languages (including Objective-C), `String`s are reference types. Swift enables you to define your own value types as `struct`s and `enum`s (which we discuss in later chapters). Swift's numeric types and `String` type are defined as `struct`s. An `enum` is often used to define sets of named constants, but in Swift it's much more powerful than in most C-based languages.

Performance Tip 3.1

You might think that copying objects introduces a lot of runtime overhead. However, the Swift compiler optimizes copy operations so that they're performed only if the copy is modified in your code—this is known as ***copy-on-write****.*

Reference Types

You'll define a class and use several existing classes in this chapter. All class types (defined with the keyword `class`) are reference types—all other Swift types are value types. A constant or variable of a reference type (often called a **reference**) is said to **refer to an object**. Conceptually this means that the constant or variable stores the object's *location*. Unlike Objective-C, C and C++, that location is not the *actual* memory address of the object, rather it's a *handle* that enables you to locate the object so you can interact with it.

Both `struct`s and `enum`s in Swift provide many of the same capabilities as classes. In many contexts where you'd use classes in other languages, Swift idiom prefers `struct`s or `enum`s. We'll say more about this later in the book.

Reference-Type Objects That Are Assigned to Constants Are Not Constant Objects

Initializing a constant (declared with `let`) with a reference-type object simply means that the constant always *refers to the same object*. You can still use a reference-type constant to access read/write properties and to call methods that modify the referenced object.

Assigning References

Reference-type objects are *not copied*. If you assign a reference-type variable to another variable or use it to initialize a constant, then both *refer to the same object* in memory.

Comparative Operators for Value Types

Conditions can be formed by using the **comparative operators** (==, !=, >, <, >= and <=) summarized in Fig. 3.5. These operators all have the same level of precedence and do not have associativity in Swift.

Algebraic operator	Comparative operator	Sample condition	Meaning of condition
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Fig. 3.5 | Comparative operators for value types.

Comparative Operators for Reference Types

One key difference between value types and reference types is comparing for equality and inequality. Only value-type constants and variables can be compared with the == (is equal to) and != (is not equal to) operators. In addition to the operators in Fig. 3.5, Swift also provides the **=== (identical to)** and **!== (not identical to)** operators for comparing reference-type constants and variables to determine whether they *refer to the same object*.

3.2.14 Code Completion in the Source-Code Editor

As you type code in the source-code editor, Xcode displays *code-completion suggestions* (Fig. 3.6) for class names, method names, property names, and more. It provides one suggestion inline in the code (in gray) and below it displays a list of other suggestions (with the current inline one highlighted in blue). You can press *Enter* to select the highlighted suggestion or you can click an item from the displayed list to choose it. You can press the *Esc* key to close the suggestion list and press it again to reopen the list.

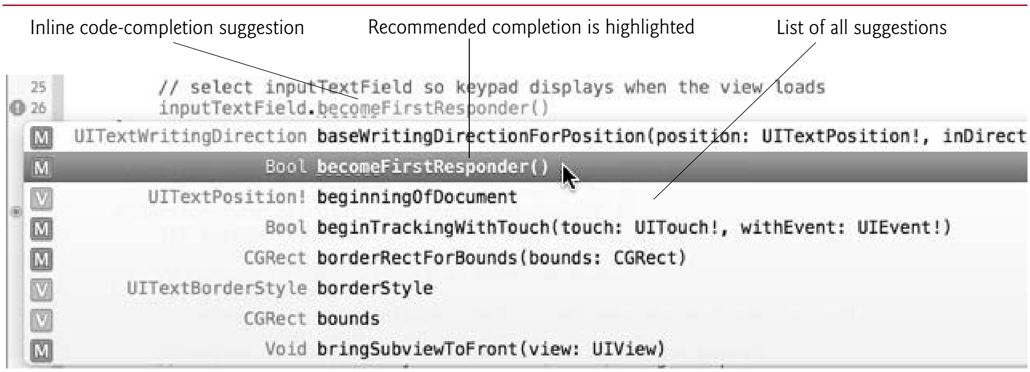


Fig. 3.6 | Code-completion suggestions in Xcode.

3.3 Building the App's UI

In this section, you'll build the **Tip Calculator** UI using the techniques you learned in Chapter 2. Here, we'll show the detailed steps for building the UI—in later chapters, we'll focus on new UI features.

3.3.1 Creating the Project

As you did in Section 2.3, begin by creating a new **Single View Application** iOS project. Specify the following settings in the **Choose options for your new project** sheet:

- **Product Name:** `TipCalculator`.
- **Organization Name:** `Deitel and Associates, Inc.`—or you can use your own organization name.
- **Company Identifier:** `com.deitel`—or you can use your own company identifier or use `edu.self`.
- **Language**—Swift.
- **Devices: iPhone**—This app is designed for iPhones and iPod touches. The app will run on iPads, but it will fill most of the screen and be centered, as in Fig. 3.7.

After specifying the settings, click **Next**, indicate where you'd like to save your project and click **Create** to create the project.

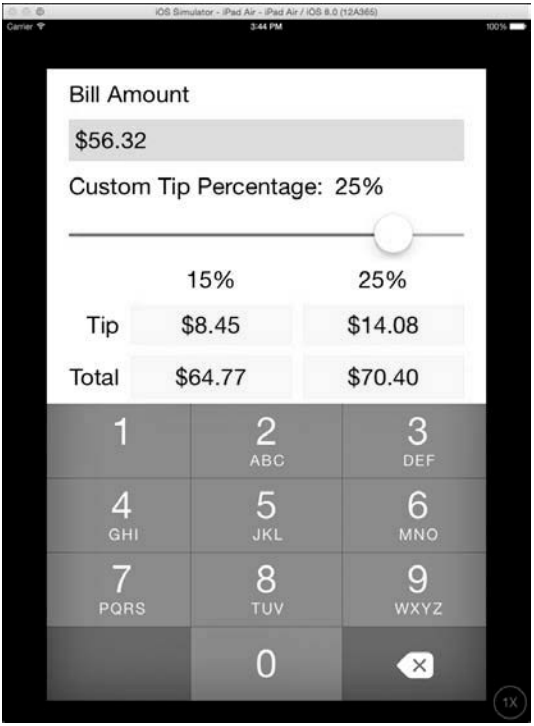


Fig. 3.7 | **Tip Calculator** running in the iPad Air simulator.

Configuring the App to Support Only Portrait Orientation

In landscape orientation, the numeric keypad would obscure parts of the **Tip Calculator**'s UI. For this reason, this app will support only portrait orientation. In the project settings' **General** tab that's displayed in the Xcode **Editor** area, scroll to the **Deployment Info** section, then for **Device Orientation** ensure that only **Portrait** is selected. Recall from Section 2.5.1 that most iPhone apps should support *portrait*, *landscape-left* and *landscape-right* orientations, and most iPad apps should also support *upside down* orientation. You can learn more about Apple's *Human Interface Guidelines* at:

```
http://bit.ly/HumanInterfaceGuidelines
```

3.3.2 Configuring the Size Classes for Designing a Portrait Orientation iPhone App

In Chapter 2, we designed a UI that supported both portrait and landscape orientations for any iOS device. For that purpose, we used the default size class **Any** for the design area's width and height. In this section, you'll configure the *design area* (also called the *canvas*) for a tall narrow device, such as an iPhone or iPod touch in portrait orientation. Select `Main.storyboard` to display the design area—also known as the canvas. At the bottom of the canvas, click the **Size Classes** control to display the size classes tool, then click in the lower-left corner to specify the size classes **Compact Width** and **Regular Height** (Fig. 3.8).

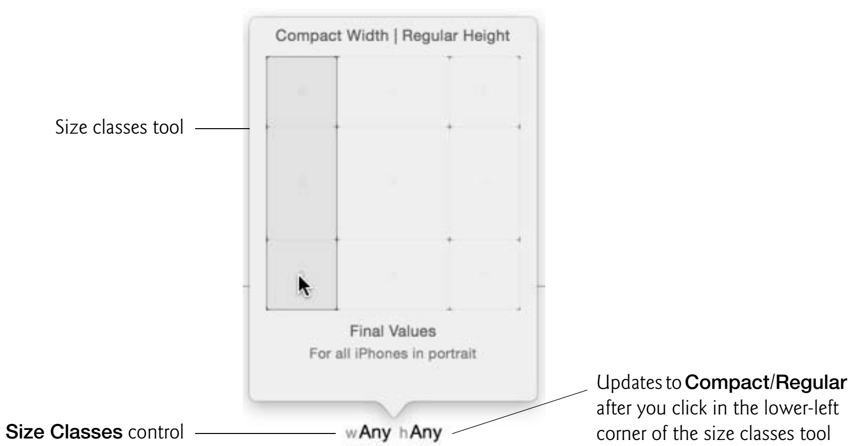


Fig. 3.8 | Size classes tool with **Compact Width** and **Regular Height** selected.

3.3.3 Adding the UI Components

In this section, you'll add and arrange the UI components to create the basic design. In Section 3.3.4, you'll add auto layout constraints to complete the design.

Step 1: Adding the "Bill Amount" Label

First, you'll add the "**Bill Amount**" **Label** to the UI:

1. Drag a **Label** from the **Object** library to the scene's upper-left corner, using the blue guide lines to position the **Label** at the recommended distance from the

scene's top and left (Fig. 3.9). The ⊕ symbol indicates that you're adding a new component to the UI.

Fig. 3.9 | Adding the "**Bill Amount**" **Label** to the scene.

2. Double click the **Label**, type `Bill Amount`, then press *Enter* to change its **Text** attribute.

Step 2: Adding the **Label** *That Displays the Formatted User Input*

Next, you'll add the blue **Label** that displays the formatted user input:

1. Drag another **Label** below the "**Bill Amount**" **Label**, such that the placement guides appear as shown in Fig. 3.10. This is where the user input will be displayed.

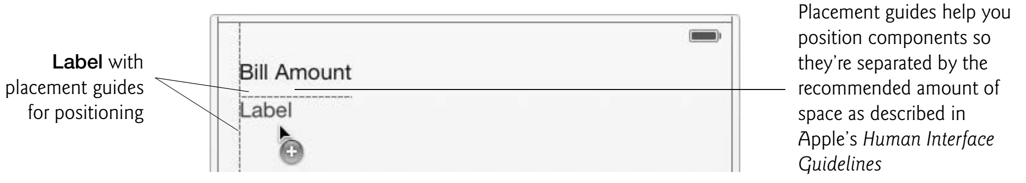


Fig. 3.10 | Adding the **Label** in which the formatted user input will be displayed.

2. Drag the middle sizing handle at the new **Label**'s right side until the blue guide line at the scene's right side appears (Fig. 3.11).

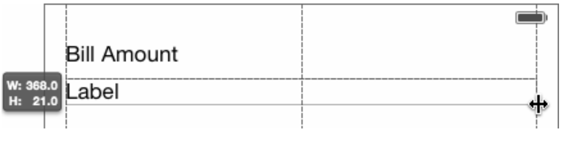


Fig. 3.11 | Resizing the **Label** where the formatted user input will be displayed.

3. In the **Attributes** inspector, scroll to the **View** section and locate the **Label**'s **Background** attribute. Click the attribute's value, then select **Other...** to display the **Colors** dialog. This dialog has five tabs at the top that allow you to select colors different ways. For this app, we used the **Crayons** tab. On the bottom row, select the **Sky** (blue) crayon as the color (Fig. 3.12), then set the **Opacity** to 50%—this allows the scene's white background to blend with the **Label**'s color, resulting in a lighter blue color. The **Label** should now appear as shown in Fig. 3.13.

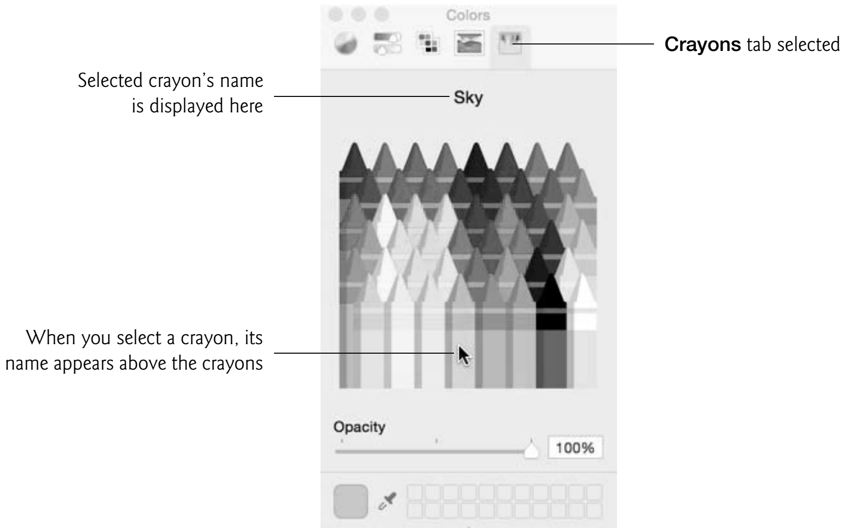


Fig. 3.12 | Selecting the **Sky** crayon for the **Label**'s background color.

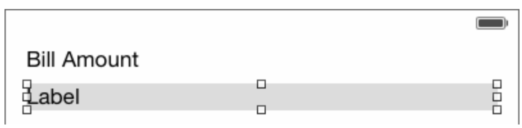


Fig. 3.13 | **Label** with **Sky** blue background and 50% opacity.

4. A **Label**'s default height is 21 points. We increased this **Label**'s height to add space above and below its text to make it more readable against the colored background. To do so, drag the bottom-center sizing handle down until the **Label**'s height is 30 (Fig. 3.14).

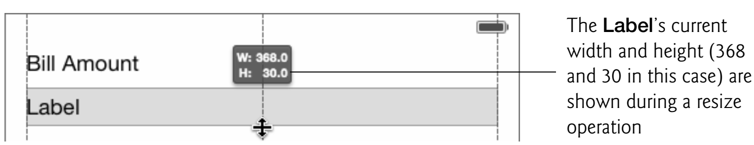


Fig. 3.14 | **Label** with **Sky** blue background and 50% opacity.

5. With the **Label** selected, delete the value for its **Text** property in the **Attributes** inspector. The **Label** should now be empty.

Step 3: Adding the "Custom Tip Percentage:" Label and a Label to Display the Current Custom Tip Percentage

Next, you'll add the **Label**s in the UI's third row:

1. Drag another **Label** onto the scene and position it below the blue **Label** as shown in Fig. 3.15.

Fig. 3.15 | Adding the "**Custom Tip Percentage:**" **Label** to the scene.

2. Double click the **Label** and set its text to `Custom Tip Percentage:`.
3. Drag another **Label** onto the scene and position it to the right of the "**Custom Tip Percentage:**" **Label** (Fig. 3.16), then set its text to 18%—the initial custom tip percentage we chose in this app, which the app will update when the user moves the **Slider**'s thumb. The UI should now appear as shown in Fig. 3.17.

Fig. 3.16 | Adding the **Label** that displays the current custom tip percentage.

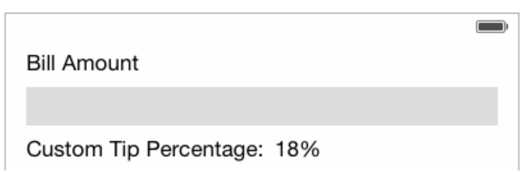


Fig. 3.17 | UI design so far.

Step 4: Creating the Custom Tip Percentage Slider

You'll now create the **Slider** for selecting the custom tip percentage:

1. Drag a **Slider** from the **Object** library onto the scene so that it's the recommended distance from the "**Custom Tip Percentage:**" **Label**, then size and position it as shown in Fig. 3.18.
2. Use the **Attributes** inspector to set the **Slider**'s **Minimum** value to `0` (the default), **Maximum** value to `30` and **Current** value to `18`.

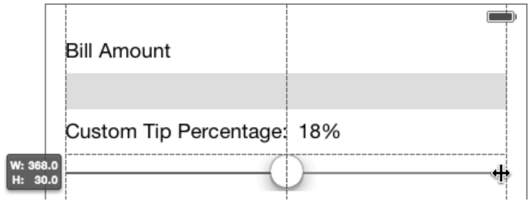


Fig. 3.18 | Creating and sizing the **Slider**.

Step 5: Adding the "15%" and "18%" Labels

Next, you'll add two more **Labels** containing the text **15%** and **18%** to serve as column headings for the calculation results. The app will update the "**18%**" **Label** when the user moves the **Slider**'s thumb. Initially, you'll position these **Labels** approximately—later you'll position them more precisely. Perform the following steps:

1. Drag another **Label** onto the scene and use the blue guides to position it the recommended distance below the **Slider** (Fig. 3.19), then set its **Text** to 15% and its **Alignment** to centered.

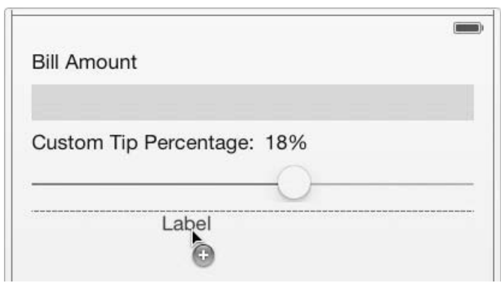


Fig. 3.19 | Adding the **Label** and right aligning it with the blue **Label**.

2. Next you'll duplicate the "**15%**" **Label**, which copies all of its settings. Hold the *option* key and drag the "**15%**" **Label** to the right (Fig. 3.20). You can also duplicate a UI component by selecting it and typing ⌘ + *D*, then moving the copy. Change the new **Label**'s text to 18%.

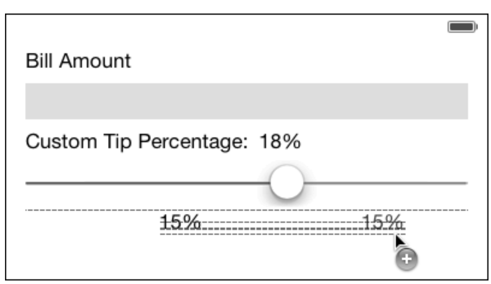


Fig. 3.20 | Duplicating the "**15%**" **Label** so that you can create the "**18%**" **Label**.

Step 6: Creating the **Labels** *That Display the Tips and Totals*

Next, you'll add four **Labels** in which the app will display the calculation results:

1. Drag a **Label** onto the UI until the blue guides appear as in Fig. 3.21.

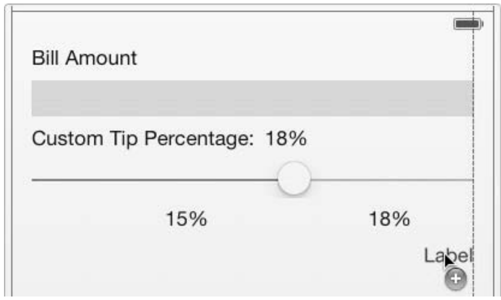


Fig. 3.21 | Creating the first yellow **Label**.

2. Drag the **Label**'s bottom-center sizing handle until the **Label**'s **Height** is `30`, and drag its left-center sizing handle until the **Label**'s **Width** is `156`.
3. Use the **Attributes** inspector to clear the **Text** attribute, set the **Alignment** so the text is centered and set the **Background** color to **Banana**, which is located in the **Color** dialog's **Crayons** tab in the second row from the bottom.
4. Set the **Autoshrink** property to **Minimum Font Scale** and change the value to `.75`—if the text becomes too wide to fit in the **Label**, this will allow the text to shrink to 75% of its original font size to accommodate more text. If you'd like the text to be able to shrink even more, you can choose a smaller value.
5. Next duplicate the yellow **Label** by holding the *option* key and dragging the **Label** to the left to create another **Label** below the "**15%**" **Label**.
6. Select both yellow **Labels** by holding the *Shift* key and clicking each **Label**. Hold the *option* key and drag any one of the selected **Labels** down until the blue guides appear as shown in Fig. 3.22.

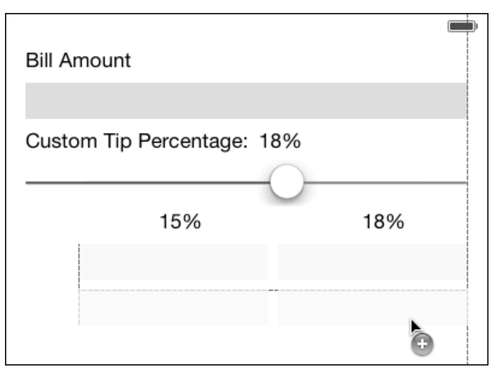


Fig. 3.22 | Creating the second row of yellow **Labels**.

7. Now you can center the "**15%**" and "**18%**" **Labels** over their columns. Drag the "**Tip**" **Label** so that the blue guide lines appear as shown in Fig. 3.23. Repeat this for the "**18%**" **Label** to center it over the right column of yellow **Labels**.

Fig. 3.23 | Repositioning the "**15%**" **Label**.

*Step 7: Creating the "**Tip**" and "**Total**" **Labels** to the Left of the Yellow **Labels***

Next you'll create the "**Tip**" and "**Total**" **Labels**:

1. Drag a **Label** onto the scene, change its **Text** to `Total`, set its **Alignment** to right aligned and position it to the left of the second row of yellow **Labels** as in Fig. 3.24.

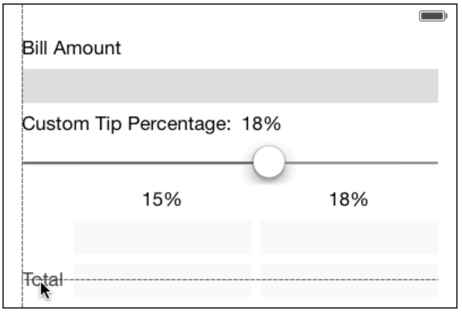


Fig. 3.24 | Positioning the "**Total**" **Label**.

2. Hold the *option* key and drag the "**Total**" **Label** up until the blue guides appear as shown in Fig. 3.25. Change the new **Label**'s text to `Tip`, then drag it to the right so that the right edges of the "**Tip**" and "**Total**" **Labels** align.

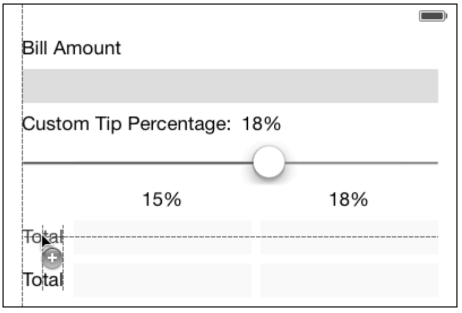


Fig. 3.25 | Duplicating the "**Total**" **Label** so that you can create the "**Tip**" **Label**.

Step 8: Creating the Text Field for Receiving User Input

You'll now create the **Text Field** that will receive the user input. Drag a **Text Field** from the **Object** library to the bottom edge of the scene, then use the **Attributes** inspector to set its **Keyboard Type** attribute to **Number Pad** and its **Appearance** to **Dark**. This **Text Field** will be *hidden* behind the numeric keypad when the app first loads. You'll receive the user's input through this **Text Field**, then format and display it in the blue **Label** at the top of the scene.

3.3.4 Adding the Auto Layout Constraints

You've now completed the **Tip Calculator** app's basic UI design, but have not yet added any auto layout constraints. If you run the app in the simulator or on a device, however, you'll notice that—depending on which simulator you use—some of the UI components extend beyond the trailing edge (Fig. 3.26). In this section, you'll add auto layout constraints so that the UI components can adjust to display properly on devices of various sizes and resolutions.

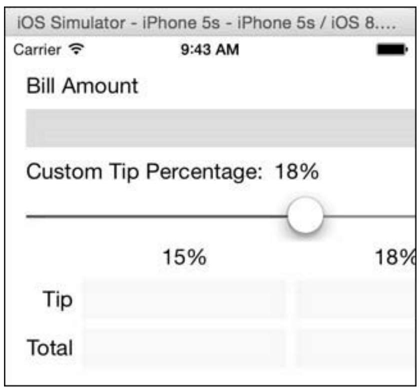


Fig. 3.26 | App in the iPhone 5s simulator without auto layout constraints added to the UI—some components flow off the trailing edge (the right side in this screen capture).

In Chapter 2, you manually added the required auto layout constraints. In this section, you'll use Interface Builder to add missing constraints automatically, then run the app again to see the results. You'll then create some additional constraints so that the app displays correctly in the simulator or on a device.

Step 1: Adding the Missing Auto Layout Constraints

To add the missing auto layout constraints:

1. Click the white background in the design area or select **View** in the document outline window.
2. At the bottom of the canvas, click the **Resolve Auto Layout Issues** (|△|) button and under **All Views in View Controller** select **Add Missing Constraints.**

Interface Builder analyzes the UI components in the design and based on their sizes, locations and alignment, then creates a set of auto layout constraints for you. In some cases, these constraints will be enough for your design, but you'll often need to tweak the results. Figure 3.27 shows the UI in the iPhone 5s simulator after Interface Builder adds the missing constraints. Now, all of the UI components are completely visible, but some of them are not sized and positioned correctly. In particular, the yellow **Label**s should all be the same width.

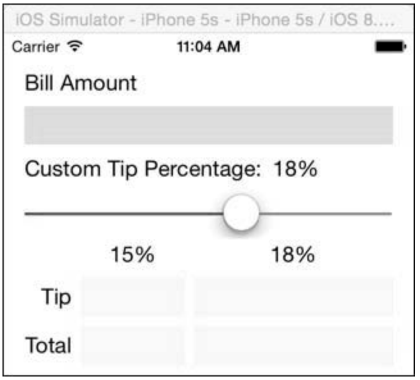


Fig. 3.27 | App in the simulator after Interface Builder adds the missing auto layout constraints—some components are not sized and positioned correctly.

Step 2: Setting the Yellow `Label`*s to Have Equal Widths*

To set the yellow **Label**s to have equal widths:

1. Select all four yellow **Label**s by holding the *shift* key and clicking each one.
2. In the auto layout tools at the bottom of the canvas, click the **Pin** tools icon (|□|). Ensure that **Equal Widths** is checked and click the **Add 3 Constraints** button, as shown in Fig. 3.28. Only three constraints are added, because three of the **Label**s will be set to have the same width as the fourth.

Fig. 3.28 | Setting **Equal Widths** for the yellow **Label**s.

Figure 3.29 shows the UI in the simulator. Setting the yellow **Label**s to **Equal Widths** caused the 18% **Label** over the right column to disappear and the "**Tip**" and "**Total**" **Label**s to become too narrow to display.

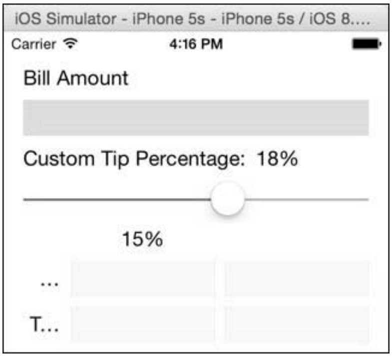


Fig. 3.29 | App in the simulator after setting the yellow **Label**s to equal widths.

Step 3: Debugging the Missing "18%" Label

Based on the initial design, the missing "**18%**" **Label** should be centered over the right column of yellow **Label**s. If you select that **Label** in the canvas and select the **Size** inspector in the **Utilities** area, you can see the missing **Label**'s complete set of constraints (Fig. 3.30).

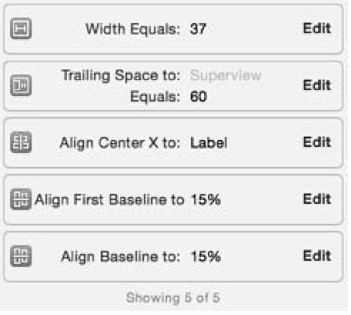


Fig. 3.30 | "**18%**" **Label**'s constraints.

There are two constraints on the "**18%**" **Label**'s horizontal positioning:

- The **Trailing Space to: Superview** constraint specifies that this **Label** should be 60 points from the scene's trailing edge.
- The **Align Center X to: Label** constraint specifies that this **Label** should be centered horizontally over the specified **Label**.

These two constraints *conflict* with one another—depending on the yellow **Label**'s width, the "**18%**" **Label** could appear different distances from the scene's trailing edge. By removing the **Trailing Space to: Superview** constraint, we can eliminate the conflict. To do so, simply click that constraint in the **Size** inspector and press the *delete* key. Figure 3.31

shows the final UI in the iPhone 5s simulator, but you can test the UI in other simulators to confirm that it works correctly in each.

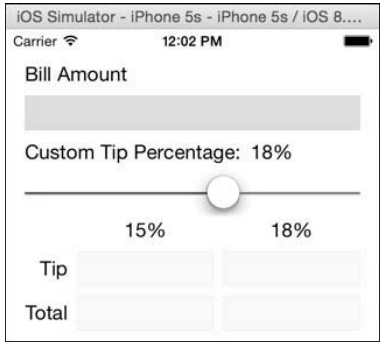


Fig. 3.31 | App with its final UI running in the simulator.

3.4 Creating Outlets with Interface Builder

You'll now use Interface Builder to create the *outlets* for the UI components that the app interacts with programmatically. Figure 3.32 shows the outlet names that we specified when creating this app. A common naming convention is to use the UI component's class name without the `UI` class prefix at the end of an outlet property's name—for example,

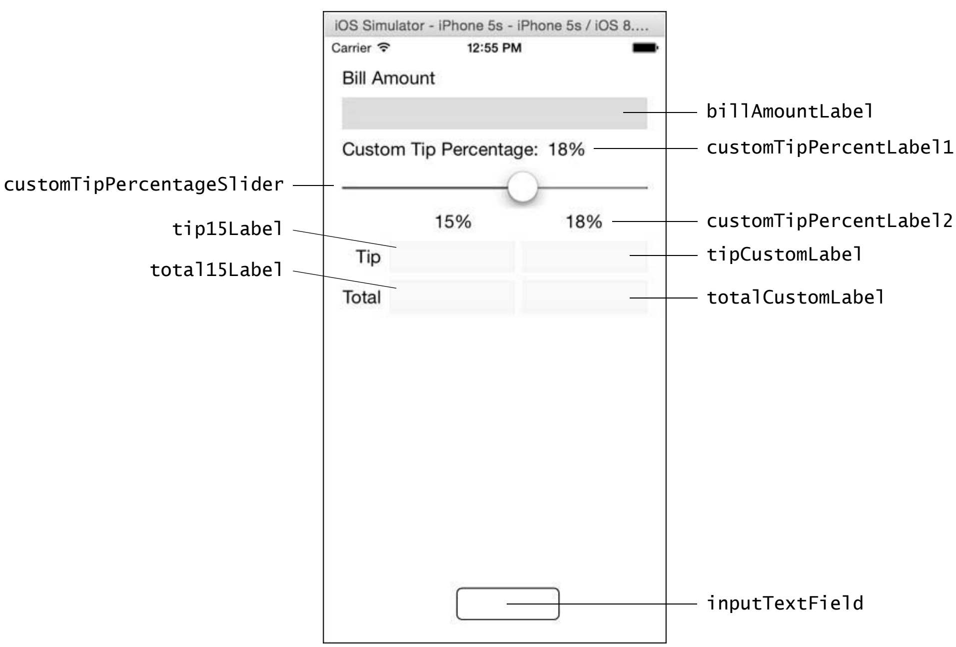


Fig. 3.32 | **Tip Calculator**'s UI components labeled with their outlet names.

`billAmountLabel` rather than `billAmountUILabel`. (At the time of this writing, Apple had not yet published their Swift coding guidelines.) Interface Builder makes it easy for you to create outlets for UI components by *control* dragging from the component into your source code. To do this, you'll take advantage of the Xcode **Assistant** editor.

Opening the Assistant Editor

To create outlets, ensure that your scene's storyboard is displayed by selecting it in the **Project** navigator. Next, select the **Assistant** editor button (◎) on the Xcode toolbar (or select **View > Assistant Editor > Show Assistant Editor**). Xcode's **Editor** area splits and the file `ViewController.swift` (Fig. 3.33) is displayed to the right of the storyboard. By default, when viewing a storyboard, the **Assistant** editor shows the corresponding view controller's source code. However, by clicking **Automatic** in the jump bar at the top of the **Assistant** editor, you can select from options for previewing the UI for different device sizes and orientations, previewing localized versions of the UI or viewing other files that you'd like to view side-by-side with the content currently displayed in the editor. The comments in lines 1–7 are autogenerated by Xcode—later, we delete these comments and replace them with our own. Delete the method `didReceiveMemoryWarning` in lines 18–21 as we will not use it in this app. We'll discuss the details of `ViewController.swift` and add code to it in Section 3.6.

Fig. 3.33 | `ViewController.swift` displayed in the **Assistant** editor.

Creating an Outlet

You'll now create an outlet for the blue **Label** that displays the user's input. You need this outlet to programmatically change the **Label**'s text to display the input in currency format. Outlets are declared as properties of a view controller class. To create the outlet:

1. *Control* drag from the blue **Label** to below line 11 in `ViewController.swift` (Fig. 3.34) and release. This displays a popover for configuring the outlet (Fig. 3.35).

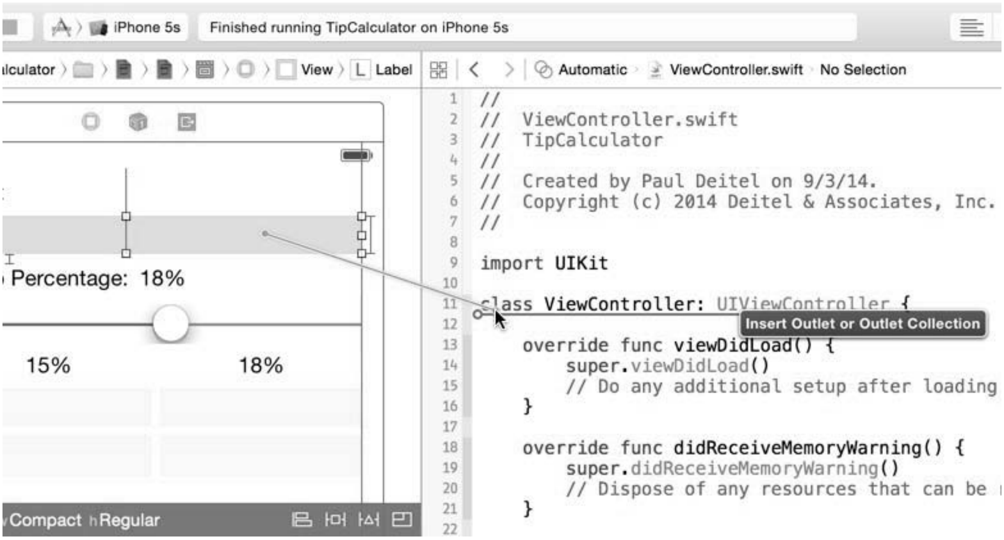


Fig. 3.34 | *Control* dragging from the scene to the **Assistant** editor to create an outlet.

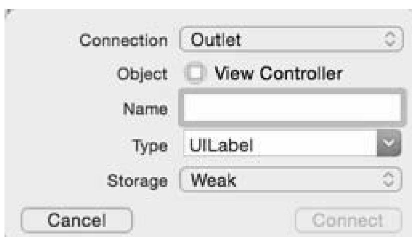


Fig. 3.35 | Popover for configuring an outlet.

2. In the popover, ensure that **Outlet** is selected for the **Connection** type, specify the name `billAmountLabel` for the outlet's **Name** and click **Connect**.

Xcode inserts the following property declaration in class `ViewController`:

```
@IBOutlet weak var billAmountLabel: UILabel!
```

We'll explain this code in Section 3.6.3. You can now use this property to programmatically modify the **Label**'s text.

Creating the Other Outlets

Repeat the steps above to create outlets for the other labeled UI components in Fig. 3.32. Your code should now appear as shown in Fig. 3.36. In the gray margin to the left of each outlet property is a small bullseye (◉) symbol indicating that the outlet is connected to a UI component. Hovering the mouse over that symbol highlights the connected UI component in the scene. You can use this to confirm that each outlet is connected properly.

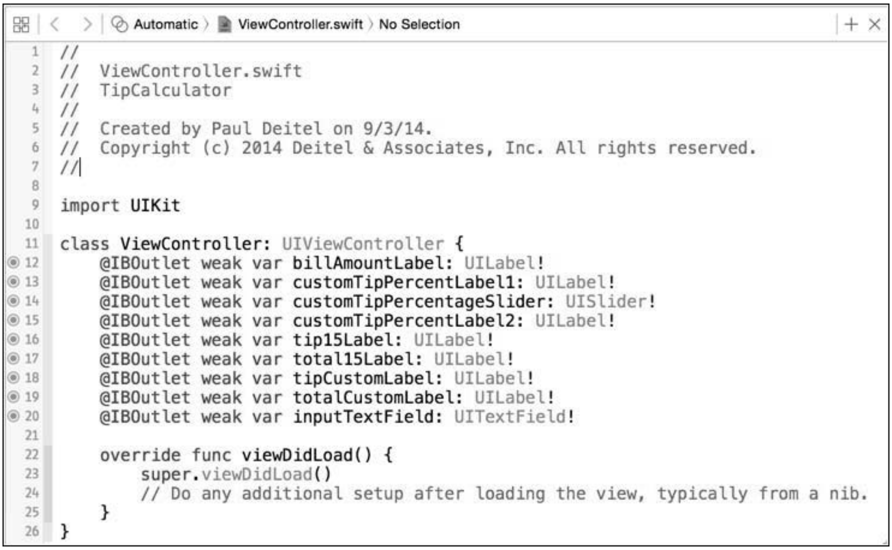


Fig. 3.36 | Code after adding outlets for the programmatically manipulated UI components.

3.5 Creating Actions with Interface Builder

Now that you've created the outlets, you need to create actions (i.e., event handlers) that can respond to the user-interface events. A **Text Field**'s **Editing Changed** event occurs every time the user changes the **Text Field**'s contents. If you connect an action to the **Text Field** for this event, the **Text Field** will send a message to the view-controller object to execute the action each time the event occurs. Similarly, the **Value Changed** event repeatedly occurs for a **Slider** as the user moves the thumb. If you connect an action method to the **Slider** for this event, the **Slider** will send a message to the view controller to execute the action each time the event occurs.

In this app, you'll create one action method that's called for each of these events. You'll connect the **Text Field** and the **Slider** to this action using the **Assistant** editor. To do so, perform the following steps:

1. *Control* drag from the **Text Field** in the scene to `ViewController.swift` between the right braces (`}`) at lines 25 and 26 (Fig. 3.37), then release. This displays a popover for configuring an outlet. From the **Connection** list in the popover, select **Action** to display the options for configuring an action (Fig. 3.38).

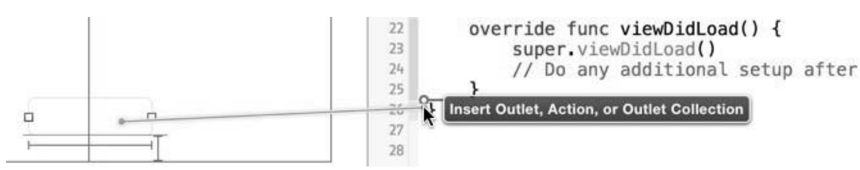


Fig. 3.37 | *Control* dragging to create an action for the **Text Field**.

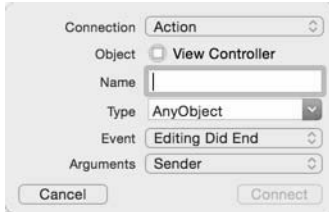


Fig. 3.38 | Popover for configuring an action.

2. In the popover, specify `calculateTip` for the action's **Name**, select **Editing Changed** for the **Event** and click **Connect**.

Xcode inserts the following empty method definition in the code:

```
@IBAction func calculateTip(sender: AnyObject) {
}
```

and displays a small bullseye (◉) symbol (Fig. 3.39) in the gray margin to the left of the method indicating that the action is connected to a UI component. Now, when the user edits the **Text Field**, a message will be sent to the `ViewController` object to execute `calculateTip`. You'll define the logic for this method in Section 3.6.6.

Connecting the Slider to Method `calculateTip`

Recall that `calculateTip` should also be called as the user changes the custom tip percentage. You can simply connect the **Slider** to this existing action to handle the **Slider**'s **Value Changed** event. To do so, select the **Slider** in the scene, then hold the *control* key and drag from the **Slider** to the `calculateTip:` method (Fig. 3.39) and release. This connects the **Slider**'s **Value Changed** event to the action. You're now ready to implement the app's logic.

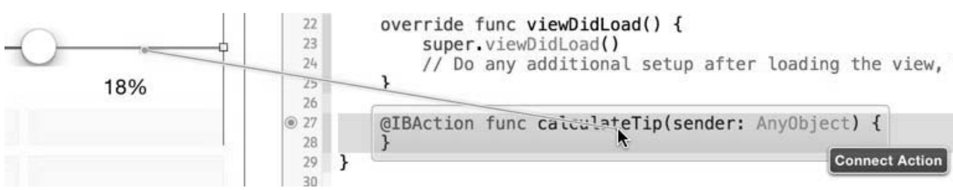


Fig. 3.39 | *Control* dragging to connect an existing `@IBAction` to the **Slider**.

3.6 Class `ViewController`

Sections 3.6.1–3.6.7 present `ViewController.swift`, which contains class `ViewController` and several global utility functions that are used throughout the class to format `NSDecimalNumbers` as currency and to perform calculations using `NSDecimalNumber` objects. We modified the autogenerated comments that Xcode inserted at the beginning of the source code file.

3.6.1 `import` Declarations

Recall that to use features from the iOS 8 frameworks, you must *import* them into your Swift code. Throughout this app, we use the UIKit framework's UI component classes. In Fig. 3.46, line 3 is an `import` declaration indicating that the program uses features from the UIKit framework. All `import` declarations must appear *before* any other Swift code (except comments) in your source-code files.

```
1  // ViewController.swift
2  // Implements the tip calculator's logic
3  import UIKit
4
```

Fig. 3.40 | `import` declaration in `ViewController.swift`.

3.6.2 `ViewController` Class Definition

In Fig. 3.41, line 5—which was generated by the IDE when you created the project—begins a **class definition** for class `ViewController`.

```
5  class ViewController: UIViewController {
```

Fig. 3.41 | `ViewController` class definition and properties.

Keyword `class` *and Class Names*

The **`class` keyword** introduces a class definition and is immediately followed by the class name (`ViewController`). Class name *identifiers* use *camel-case* naming in which each word in the identifier begins with a capital letter. Class names (and other type names) begin with an initial uppercase letter and other identifiers begin with lowercase letters. Each new class you create becomes a new type that can be used to declare variables and create objects.

Class Body

A **left brace** (at the end of line 5), **`{`**, begins the body of every class definition. A corresponding **right brace** (at line 82 in Fig. 3.45), **`}`**, ends each class definition. By convention, the contents of a class's body are indented.

> **Error-Prevention Tip 3.1**
> *A class must be defined before you use it in a given source-code file. In an Xcode project, if you define a class in one* `.swift` *file, you can use it in the project's other source-code files—which is typical of other object-oriented languages, such as Objective-C, Java, C# and C++.*

Inheriting from Class `UIViewController`

The notation `: UIViewController` in line 5 indicates that class `ViewController` **inherits** from class `UIViewController`—the UIKit framework superclass of all view controllers. *Inheritance* is a form of software reuse in which a new class is created by absorbing an existing class's members and enhancing them with new or modified capabilities. This relationship indicates that a `ViewController` *is a* `UIViewController`. It also ensures that `ViewController` has the basic capabilities that iOS expects in all view controllers, including methods like `viewDidLoad` (Section 3.6.5) that help iOS manage a view controller's lifecycle. The class on the left of the `:` in line 5 is the *subclass* (derived class) and one on the right is the *superclass* (base class). Every scene has its own `UIViewController` subclass that defines the scene's event handlers and other logic. Unlike some object-oriented programming languages, Swift classes are not required to directly or indirectly inherit from a common superclass.

3.6.3 `ViewController`'s `@IBOutlet` Properties

Figure 3.42 shows class `ViewController`'s nine `@IBOutlet` property declarations that were created by Interface Builder when you created the outlets in Section 3.4. Typically, you'll define a class's *properties* first followed by the class's *methods*, but this is not required.

```
    // properties for programmatically interacting with UI components
    @IBOutlet weak var billAmountLabel: UILabel!
    @IBOutlet weak var customTipPercentLabel1: UILabel!
    @IBOutlet weak var customTipPercentageSlider: UISlider!
    @IBOutlet weak var customTipPercentLabel2: UILabel!
    @IBOutlet weak var tip15Label: UILabel!
    @IBOutlet weak var total15Label: UILabel!
    @IBOutlet weak var tipCustomLabel: UILabel!
    @IBOutlet weak var totalCustomLabel: UILabel!
    @IBOutlet weak var inputTextField: UITextField!

```

Fig. 3.42 | `ViewController`'s `@IBOutlet` properties.

`@IBOutlet` Property Declarations

The notation `@IBOutlet` indicates to Xcode that the property references a UI component in the app's storyboard. When a scene loads, the UI component objects are created, an object of the corresponding view-controller class is created and the connections between the view controller's outlet properties and the UI components are established. The connection information is stored in the storyboard. `@IBOutlet` properties are declared as *variables* using the `var` keyword, so that the storyboard can assign each UI component object's reference to the appropriate outlet once the UI components and view controller object are created.

Automatic Reference Counting (ARC) and Property Attributes

Swift manages the memory for your app's reference-type objects using **automatic reference counting (ARC)**, which keeps track of how many references there are to a given object. The runtime can remove an object from memory only when its *reference count* becomes 0.

Property attributes can specify whether a class maintains an ownership or nonownership relationship with the referenced object. By default, properties in Swift create **strong references** to objects, indicating an ownership relationship. Every strong reference increments an object's reference count by 1. When a strong reference no longer refers to an object, its reference count decrements by 1. The code that manages incrementing and decrementing the reference counts is inserted by the Swift compiler.

The `@IBOutlet` properties are declared as **`weak`** references, because the view controller *does not own* the UI components—the view defined by the storyboard that created them does. A `weak` reference does *not* affect the object's reference count. A view controller does, however, have a strong reference to its view.

Type Annotations and Implicitly Unwrapped Optional Types

A **type annotation** specifies a variable's or constant's type. Type annotations are specified by following the variable's or constant's identifier with a colon (:) and a type name. For example, line 7 (Fig. 3.42) indicates that `billAmountLabel` is a `UILabel!`. Recall from Section 3.2.12 that the exclamation point indicates an implicitly unwrapped optional type and that variables of such types are initialized to `nil` by default. This allows the class to compile, because these `@IBOutlet` properties are initialized—they'll be assigned actual UI component objects once the UI is created at runtime.

3.6.4 Other `ViewController` Properties

Figure 3.43 shows class `ViewController`'s other properties, which you should add below the `@IBOutlet` properties. Line 18 defines the constant `decimal100` that's initialized with an `NSDecimalNumber` object. Identifiers for Swift constants follow the same camel-case naming conventions as variables. Class `NSDecimalNumber` provides many **initializers**—this one receives a `String` parameter containing the initial value (`"100.0"`), then returns an `NSDecimalNumber` representing the corresponding numeric value. We'll use `decimal100` to calculate the custom tip percentage by dividing the slider's value by 100.0. We'll also use it to divide the user's input by 100.0 for placing a decimal point in the bill amount that's displayed at the top of the app. Initializers are commonly called constructors in many other object-oriented programming languages. Line 19 defines the constant `decimal15Percent` that's initialized with an `NSDecimalNumber` object representing the value 0.15. We'll use this to calculate the 15% tip.

```
17      // NSDecimalNumber constants used in the calculateTip method
18      let decimal100 = NSDecimalNumber(string: "100.0")
19      let decimal15Percent = NSDecimalNumber(string: "0.15")
20
```

Fig. 3.43 | `ViewController` class definition and properties.

Initializer Parameter Names Are Required

When initializing an object in Swift, you must specify each parameter's name, followed by a colon (:) and the argument value. As you type your code, Xcode displays the parameter names for initializers and methods to help you write code quickly and correctly. Required parameter names in Swift are known as **external parameter names**.

Type Inference

Neither constant in Fig. 3.43 was declared with a type annotation. Like many popular languages, Swift has powerful **type inference** capabilities and can determine a constant's or variable's type from its initializer value. In lines 18–19, Swift infers from the initializers that both constants are `NSDecimalNumber`s.

3.6.5 Overridden `UIViewController` method `viewDidLoad`

Method `viewDidLoad` (Fig. 3.44)—which Xcode generated when it created class `ViewController`—is inherited from superclass `UIViewController`. You typically *override* it to define tasks that can be performed only *after* the view has been initialized. You should add lines 25–26 to the method.

```
21     // called when the view loads
22     override func viewDidLoad() {
23         super.viewDidLoad()
24
25         // select inputTextField so keypad displays when the view loads
26         inputTextField.becomeFirstResponder()
27     }
28
```

Fig. 3.44 | Overridden `UIViewController` method `viewDidLoad`.

A method definition begins with the keyword **`func`** (line 22) followed by the function's name and parameter list enclosed in required parentheses, then the function's body enclosed in braces (`{` and `}`).The parameter list optionally contains a comma-separated list of parameters with type annotations. This function does not receive any parameters, so its parameter list is empty—you'll see a method with parameters in Section 3.6.6. This method does not return a value, so it does not specify a return type—you'll see how to specify return types in Section 3.6.7.

When overriding a superclass method, you declare it with keyword **`override`** preceding the keyword `func`, and the first statement in the method's body typically uses the **`super`** keyword to invoke the superclass's version of the method (line 23). The keyword `super` references the object of the class in which the method appears, but is used to access members inherited from the superclass.

Displaying the Numeric Keypad When the App Begins Executing

In this app, we want `inputTextField` to be the selected object when the app begins executing so that the numeric keypad is displayed immediately. To do this, we use property `inputTextField` to invoke the `UITextField` method `becomeFirstResponder`, which programmatically makes `inputTextField` the *active component* on the screen—as if the user touched it. You configured `inputTextField` such that when it's selected, the numeric keypad is displayed, so line 26 displays this keypad when the view loads.

3.6.6 `ViewController` Action Method `calculateTip`

Method `calculateTip` (Fig. 3.45) is the *action* (as specified by `@IBAction` on line 31) that responds to the **Text Field**'s **Editing Changed** event and the **Slider**'s **Value Changed** event. Add the code in lines 32–81 to the body of `calculateTip`. (If you're entering the Swift code as you read this section, you'll get errors on several statements that perform `NSDecimalNumber` calculations using overloaded operators that you'll define in Section 3.6.7.) The method takes one parameter. Each parameter's name must be declared with a type annotation specifying the *parameter's type*. When a view-controller object receives a message from a UI component, it also receives as an argument a reference to that component—the event's **sender**. Parameter `sender`'s type—the Swift type **`AnyObject`**—represents *any* type of object and does not provide any information about the object. For this reason, the object's type must be determined at runtime. This **dynamic typing** is used for actions (i.e., event handlers), because many different types of objects can generate events. In action methods that respond to events from multiple UI components, the `send-`

`er` is often used to determine which UI component the user interacted with (as we do in lines 42 and 57).

```
29      // called when the user edits the text in the inputTextField
30      // or moves the customTipPercentageSlider's thumb
31      @IBAction func calculateTip(sender: AnyObject) {
32          let inputString = inputTextField.text // get user input
33
34          // convert slider value to an NSDecimalNumber
35          let sliderValue =
36              NSDecimalNumber(integer: Int(customTipPercentageSlider.value))
37
38          // divide sliderValue by decimal100 (100.0) to get tip %
39          let customPercent = sliderValue / decimal100
40
41          // did customTipPercentageSlider generate the event?
42          if sender is UISlider {
43              // thumb moved so update the Labels with new custom percent
44              customTipPercentLabel1.text =
45                  NSNumberFormatter.localizedStringFromNumber(customPercent,
46                      numberStyle: NSNumberFormatterStyle.PercentStyle)
47              customTipPercentLabel2.text = customTipPercentLabel1.text
48          }
49
50          // if there is a bill amount, calculate tips and totals
51          if !inputString.isEmpty {
52              // convert to NSDecimalNumber and insert decimal point
53              let billAmount =
54                  NSDecimalNumber(string: inputString) / decimal100
55
56              // did inputTextField generate the event?
57              if sender is UITextField {
58                  // update billAmountLabel with currency-formatted total
59                  billAmountLabel.text = " " + formatAsCurrency(billAmount)
60
61                  // calculate and display the 15% tip and total
62                  let fifteenTip = billAmount * decimal15Percent
63                  tip15Label.text = formatAsCurrency(fifteenTip)
64                  total15Label.text =
65                      formatAsCurrency(billAmount + fifteenTip)
66              }
67
68              // calculate custom tip and display custom tip and total
69              let customTip = billAmount * customPercent
70              tipCustomLabel.text = formatAsCurrency(customTip)
71              totalCustomLabel.text =
72                  formatAsCurrency(billAmount + customTip)
73          }
74          else { // clear all Labels
75              billAmountLabel.text = ""
76              tip15Label.text = ""
```

Fig. 3.45 | `ViewController` action method `calculateTip`. (Part 1 of 2.)

```
77                  total15Label.text = ""
78                  tipCustomLabel.text = ""
79                  totalCustomLabel.text = ""
80              }
81          }
82      }
83
```

Fig. 3.45 | `ViewController` action method `calculateTip`. (Part 2 of 2.)

Getting the Current Values of `inputTextField` *and* `customTipPercentageSlider`

Line 32 stores the value of `inputTextField`'s **`text`** property—which contains the user's input—in the local `String` variable `inputString`—Swift infers type `String` because `UITextField`'s `text` property is a `String`.

Lines 35–36 get the `customTipPercentageSlider`'s **`value`** property, which contains a `Float` value representing the **Slider**'s *thumb position* (a value from 0 to 30, as specified in Section 3.3.3). The value is a `Float`, so we could get tip percentages like, 3.1, 15.245, etc. This app uses only whole-number tip percentages, so we convert the value to an `Int` before using it to initialize the `NSDecimalNumber` object that's assigned to local variable `sliderValue`. In this case, we use the `NSDecimalNumber` initializer that takes an `Int` value named `integer`.

Line 39 uses the overloaded division operator function that we define in Section 3.6.7 to divide `sliderValue` by 100 (`decimal100`). This creates an `NSDecimalNumber` representing the custom tip percentage that we'll use in later calculations and that will be displayed as a *locale-specific* percentage `String` showing the current custom tip percentage.

Updating the Custom Tip Percentage Labels When the **Slider** *Value Changes*

Lines 42–48 update `customTipPercentLabel1` and `customTipPercentLabel2` when the **Slider** value changes. Line 42 determines whether the `sender` *is a* `UISlider` object, meaning that the user interacted with the `customTipPercentageSlider`. The **`is`** operator returns `true` if an object's class is the same as, or has an *is a* (inheritance) relationship with, the class in the right operand.

We perform a similar test at line 57 to determine whether the user interacted with the `inputTextField`. Testing the `sender` argument like this enables you to perform *different* tasks, based on the component that caused the event.

Lines 44–46 set the `customTipPercentLabel1`'s `text` property to a locale-specific percentage `String` based on the device's current locale. `NSNumberFormatter` class method `localizedStringFromNumber` returns a `String` representation of a formatted number. The method receives two arguments:

- The first is the `NSNumber` to format. Class `NSDecimalNumber` is a subclass of `NSNumber`, so you can use an `NSDecimalNumber` anywhere that an `NSNumber` is expected.
- The second argument (which has the external parameter name `numberStyle`) is a constant from the enumeration **`NSNumberFormatterStyle`** that represents the formatting to apply to the number—the `PercentStyle` constant indicates that the number should be formatted as a percentage. Because the second argument must be of type `NSNumberFormatterStyle`, Swift can infer information about the

method's argument. As such, it's possible to write the expression `NSNumberFormatterStyle.PercentStyle` with the shorthand notation:

```
.PercentStyle
```

Line 47 assigns the same `String` to `customTipPercentLabel2`'s `text` property.

Updating the Tip and Total Labels

Lines 51–80 update the tip and total **Label**s that display the calculation results. Line 51 uses the Swift `String` type's **`isEmpty`** property to ensure that `inputString` is not empty—that is, the user entered a bill amount. If so, lines 53–72 perform the tip and total calculations and update the corresponding **Label**s; otherwise, the `inputTextField` is empty and lines 75–79 clear all the tip and total **Label**s and the `billAmountLabel` by assigning the empty `String` literal (`""`) to their `text` properties.

Lines 53–54 use `inputString` to initialize an `NSDecimalNumber`, then divide it by 100 to place the decimal point in the bill amount—for example, if the user enters 5632, the amount used for calculating tips and totals is 56.32.

Lines 57–66 execute only if the event's `sender` was a `UITextField`—that is, the user tapped keypad buttons to enter or remove a digit in this app's `inputTextField`. Line 59 displays the currency-formatted bill amount in `billAmountLabel` by calling the `formatAsCurrency` method (defined in Section 3.6.7). Line 62 calculates the 15% tip amount by using an overloaded multiplication operator function for `NSDecimalNumbers` (defined in Section 3.6.7). Then line 63 displays the currency-formatted value in the `tip15Label`. Next, lines 64–65 calculates and displays the total amount for a 15% tip by using an overloaded addition operator function for `NSDecimalNumbers` (defined in Section 3.6.7) to perform the calculation, then passing the result to the `formatAsCurrency` function. Lines 69–72 calculate and display the custom tip and total amounts based on the custom tip percentage.

Why an External Name Is Not Required for a Method's First Argument

You might be wondering why we did not provide a parameter name for the first argument in the method call at lines 45–46. For method calls, Swift requires external parameter names for all parameters *after* the first parameter. Apple's reasoning for this is that they want method calls to read like sentences. A method's name should refer to the first parameter, and each subsequent parameter should have a name that's specified as part of the method call.

3.6.7 Global Utility Functions Defined in ViewController.swift

Figure 3.46 contains several global utility functions used throughout class `ViewController`. Add lines 84–103 after the closing right brace of class `ViewController`.

```
84  // convert a numeric value to localized currency string
85  func formatAsCurrency(number: NSNumber) -> String {
86      return NSNumberFormatter.localizedStringFromNumber(
87          number, numberStyle: NSNumberFormatterStyle.CurrencyStyle)
88  }
89
```

Fig. 3.46 | `ViewController.swift` global utility and overloaded operator functions. (Part 1 of 2.)

```
90  // overloaded + operator to add NSDecimalNumbers
91  func +(left: NSDecimalNumber, right: NSDecimalNumber) -> NSDecimalNumber {
92      return left.decimalNumberByAdding(right)
93  }
94
95  // overloaded * operator to multiply NSDecimalNumbers
96  func *(left: NSDecimalNumber, right: NSDecimalNumber) -> NSDecimalNumber {
97      return left.decimalNumberByMultiplyingBy(right)
98  }
99
100 // overloaded / operator to divide NSDecimalNumbers
101 func /(left: NSDecimalNumber, right: NSDecimalNumber) -> NSDecimalNumber {
102     return left.decimalNumberByDividingBy(right)
103 }
```

Fig. 3.46 | `ViewController.swift` global utility and overloaded operator functions. (Part 2 of 2.)

Defining a Function—`formatAsCurrency`

Lines 85–88 define the function `formatAsCurrency`. Like a method definition, a function definition begins with the keyword `func` (line 85) followed by the function's name and parameter list enclosed in required parentheses, then the function's body enclosed in braces (`{` and `}`). The primary difference between a method and a function is that a method is defined in the body of a class definition (or `struct` or `enum` definition). Function `formatAsCurrency` receives one parameter (`number`) of type `NSNumber` (from the Foundation framework).

A function may also specify a return type by following the parameter list with `->` and the type the function returns—this function returns a `String`. A function that does not specify a return type does not return a value—if you prefer to be explicit, you can specify the return type `Void`. A function with a return type uses a **`return`** statement (line 86) to pass a result back to its caller.

We use `formatAsCurrency` throughout class `ViewController` to format `NSDecimalNumber`s as locale-specific currency `String`s. `NSDecimalNumber` is a subclass of `NSNumber`, so any `NSDecimalNumber` can be passed as an argument to this function. An `NSNumber` parameter can also receive as an argument any Swift numeric type value—such types are automatically *bridged* by the runtime to type `NSNumber`.

Lines 86–87 invoke `NSNumberFormatter` class method `localizedStringFromNumber`, which returns a locale-specific `String` representation of a number. This method receives as arguments the `NSNumber` to format—`formatAsCurrency`'s `number` parameter—and a constant from the `NSNumberFormatterStyle` `enum` that specifies the formatting style—the constant `CurrencyStyle` specifies that a *locale-specific currency format* should be used. Once again, we could have specified the second argument as `.CurrencyStyle`, because Swift knows that the `numberStyle` parameter must be a constant from the `NSNumberFormatterStyle` enumeration and thus can infer the constant's type.

Defining Overloaded Operator Functions for Adding, Subtracting and Multiplying `NSDecimalNumber`s

Lines 91–93, 96–98 and 101–103 create global functions that overload the addition (+), multiplication (*) and division (/) operators, respectively. **Global functions** (also called free functions or just functions) are defined outside a type definition (such as a class). These functions enable us to:

- add two `NSDecimalNumber`s with the + operator (lines 65 and 72 of Fig. 3.45)
- multiply two `NSDecimalNumber`s with the * operator (lines 62 and 69 of Fig. 3.45)
- divide two `NSDecimalNumber`s with the / operator (lines 39 and 54 of Fig. 3.45)

Overloaded operator functions are defined like other global functions, but the function name is the symbol of the operator being overloaded (Fig. 3.46lines 91, 96 and 101). Each of these functions receives two `NSDecimalNumber`s representing the operator's left and right operands.

The addition (+) operator function (lines 91–93) returns the result of invoking `NSDecimalNumber` instance method **`decimalNumberByAdding`** on the left operand with the right operand as the method's argument—this adds the operands. The multiplication (*) operator function (lines 96–98) returns the result of invoking `NSDecimalNumber` instance method **`decimalNumberByMultiplyingBy`** on the left operand with the right operand as the method's argument—this multiplies the operands. The division (/) operator function (lines 101–103) returns the result of invoking `NSDecimalNumber` instance method **`decimalNumberByDividingBy`** on the left operand with the right operand as the method's argument—this divides the left operand by the right operand. Since each of these `NSDecimalNumber` instance methods receives only one parameter, the parameter's name is not required in the method call. Unlike initializers and methods, a global function's parameter names are not external parameter names and are not required in function calls unless they're are explicitly defined as external parameter names in the function's definition.

3.7 Wrap-Up

This chapter presented the **Tip Calculator** app that calculates and displays 15% and custom tip percentage tips and totals for a restaurant bill. The app uses **Text Field** and **Slider** UI components to receive user input and update suggested tips and bill totals in response to each user interaction.

We introduced Swift—Apple's programming language of the future—and several of its object-oriented programming capabilities, including objects, classes, inheritance, methods and properties. As you saw, the app's code required various Swift data types, operators, control statements and keywords.

You learned about strong and `weak` references and that only strong references affect an object's reference count. You also learned that iOS's automatic reference counting (ARC) removes an object from memory only when the object's reference count becomes 0.

You used Interface Builder to design the app's UI visually. We showed how to build your UI faster by duplicating UI components that had similar attribute settings. You learned that **Label**s (`UILabel`), **Slider**s (`UISlider`) and **Text Field**s (`UITextField`) are part of iOS's UIKit framework that's automatically included with each app you create.

We showed how to use `import` to give your code access to features in preexisting frameworks. You learned that a scene is managed by a view-controller object that determines what information is displayed and how user interactions with the scene's UI are processed. Our view-controller class inherited from class `UIViewController`, which defines the base capabilities required by view controllers in iOS.

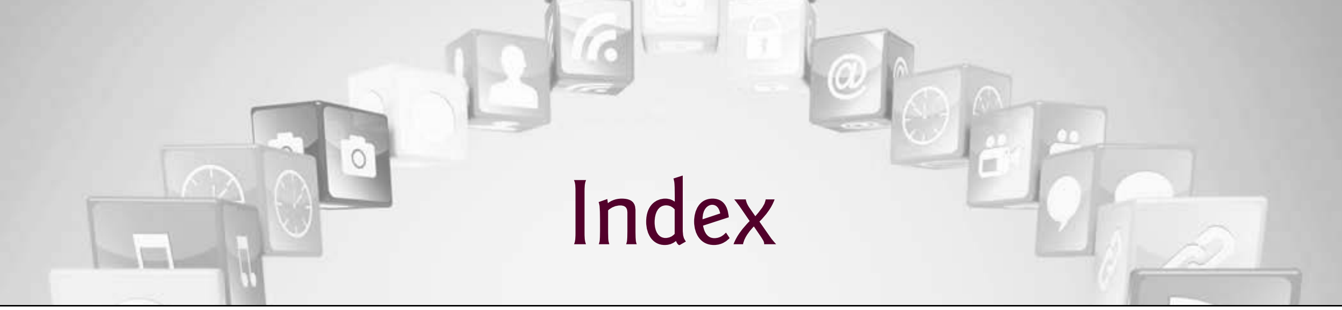
You used Interface Builder to generate `@IBOutlet` properties (outlets) in your view controller for programmatically interacting with the app's UI components. You used visual tools in Interface Builder to connect a UI control to a corresponding outlet in the view controller. Once a connection was made, the view controller was able to manipulate the corresponding UI component programmatically.

You saw that interacting with a UI component caused a user-interface event and sent a message from the UI component to an action (event-handling method) in the view controller. You learned that an action is declared in Swift code as an `@IBAction`. You used visual tools in Interface Builder to connect the action to specific user-interface events.

Next, you learned that after all the objects in a storyboard are created, iOS sends a `viewDidLoad` message to the corresponding view controller so that it can perform view-specific tasks that can be executed only after the UI components in the view exist. You also called the `UITextField`'s `becomeFirstResponder` method in `viewDidLoad` so that iOS would display this keypad immediately after the view loaded.

You used `NSDecimalNumbers` for precise financial calculations. You also used class `NSNumberFormatter` to create locale-specific currency and percentage string representations of `NSDecimalNumbers`. You used Swift's operator overloading capabilities to simplify `NSDecimalNumber` calculations.

In the next chapter, we present the **Twitter Searches** app, which allows you to save your favorite (possibly lengthy) Twitter search strings with easy-to-remember short tag names. You'll store the search strings and their short tag names in Foundation framework collections. You'll also use iCloud key–value pair storage so that you can sync your query between all your iOS devices that have the **Twitter Searches** app installed.



Index

Symbols

Numerics

A

B

I

M

N