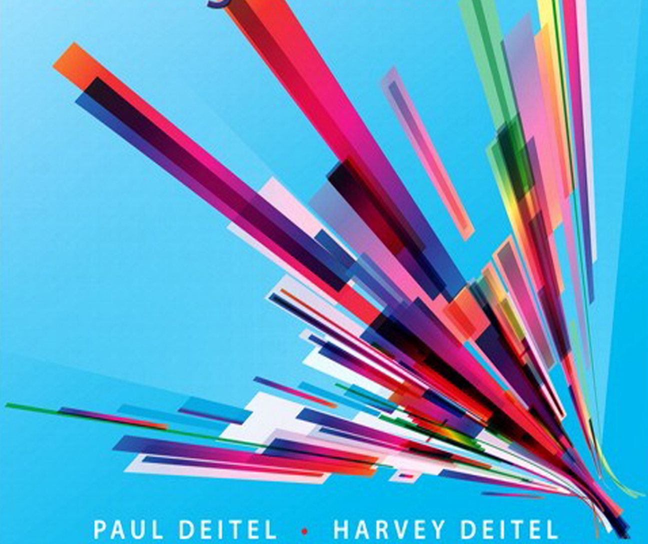
DEITEL® DEVELOPER SERIES

The New Language for iOS® and OS X® Developers

Swift™ for Programmers



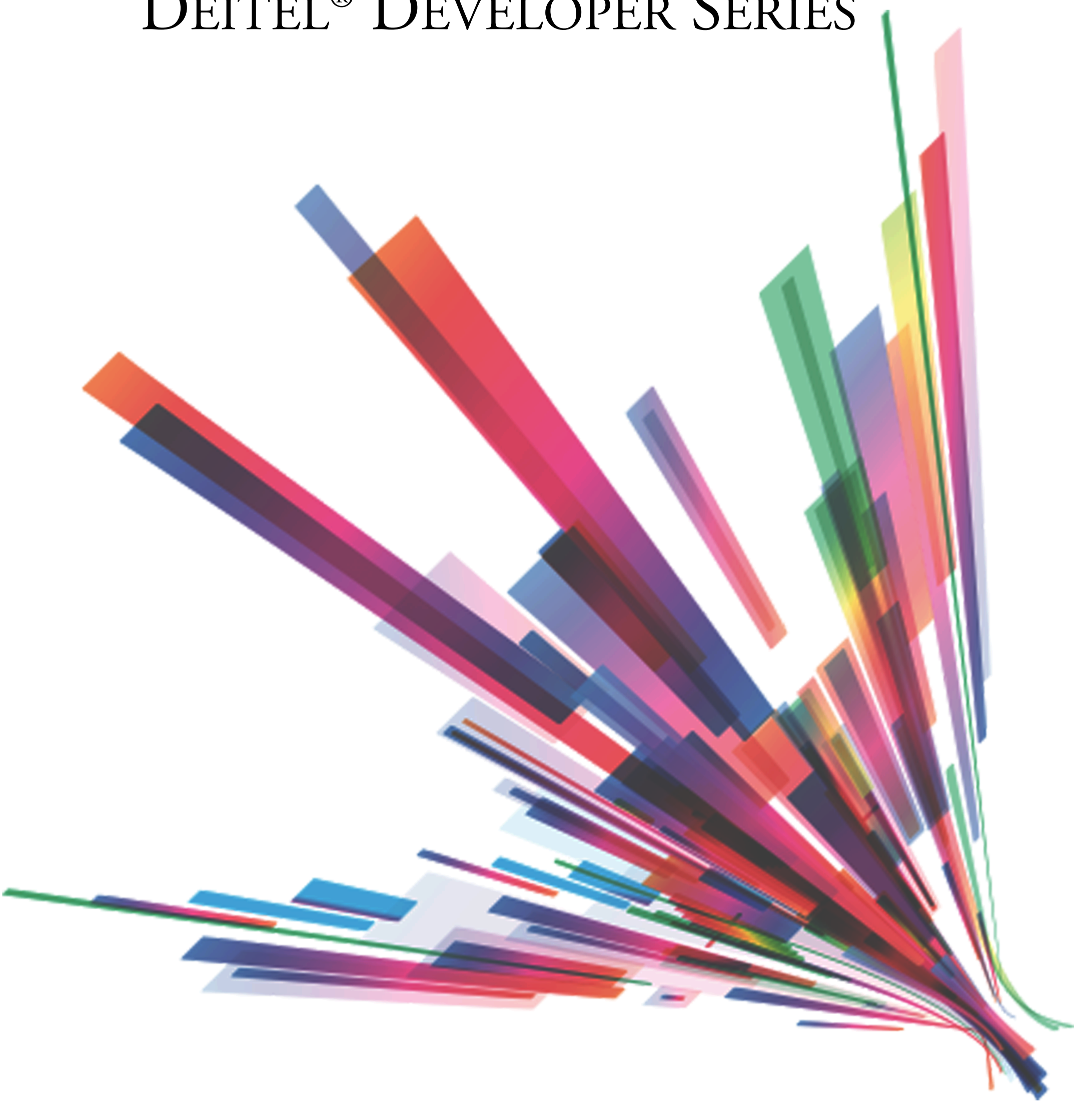
PAUL DEITEL • HARVEY DEITEL

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Swift™ for Programmers

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: 978-0-13402136-2
ISBN-10: 0-13-402136-3

Text printed in the United States at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, January 2015

SWIFT™ FOR PROGRAMMERS

DEITEL® DEVELOPER SERIES

Paul Deitel • Harvey Deitel
Deitel & Associates, Inc.

DEITEL®

PRENTICE HALL

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™, Volume 1
Java™ for Programmers, 3/E
JavaScript for Programmers
Swift™ 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 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® 8 App Development Fundamentals, 3/e
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 Loving Memory of Aunt Rochelle Deitel:

The most positive person we ever knew.
You brought joy to our lives.

Harvey, Barbara, Paul and Abbey

Trademarks

Contents

This page intentionally left blank

Preface

Welcome to Apple's new Swift programming language and *Swift for Programmers*! This book presents leading-edge computing technologies for software developers. It's designed primarily for three audiences of developers who already know object-oriented programming and are considering using Swift:

- Objective-C programmers who are developing *new* iOS and/or OS X apps and who want to quickly begin using Swift in their apps.
- Objective-C programmers who are enhancing *existing* iOS and/or OS X apps and who want to quickly begin using Swift in their apps.
- Java, C++ and C# programmers who are new to iOS and OS X development and who want to start developing iOS and/or OS X apps in Swift.

Chapters 1 through 12 focus on Swift programming, then Chapters 13 and 14 briefly introduce iOS 8 app development. The iOS 8 chapters are condensed versions of Chapters 2 and 3 of our book, *iOS® 8 for Programmers: An App-Driven Approach with Swift*™, in which we focus on building many complete iPhone® and iPad® apps.[1]

We emphasize software engineering best practices. At the heart of the book is the Deitel signature "live-code approach." Rather than using only code snippets, we present most concepts in the context of complete working Swift programs that run on OS X® and— in the last two chapters—iOS® 8. Each complete code example is accompanied by one or more live sample executions. In the few cases where we use code snippets, we always extract them from compiled, correctly executing, live-code examples. All of the book's source code is available at

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

Some complete live-code programs might appear to be code snippets—this is because Swift eliminates various items that are common in many C-based languages, such as the need for a `main` method. For example, the following is actually a complete Swift program:

```
println("Welcome to Swift Programming!")
```

Swift Programming Language

Swift was a surprise announcement at Apple's WWDC (Worldwide Developer Conference) in June 2014. Because the language is so new, it's likely to evolve quickly over the next few years. Here's some key aspects of Swift:

1. Swift is a young language that's evolving rapidly. We plan to post bonus content covering important new features as they emerge. See `http://www.deitel.com/books/SwiftFP` for details.

- *Apple's Language of the Future*—Apple is the most valuable technology company in the world, and they've declared that Swift is their language of the future for app and systems programming.
- *Popular Language Features*—Swift is a contemporary language with simpler syntax than Objective-C. Because Swift is new, its designers were able to include popular features like those in Objective-C, Java, C#, Ruby, Python and many others. These features (which are listed in Fig. 1.1) include type inference, tuples, closures (lambdas), generics, operator overloading, functions with multiple return values, optionals, `String` interpolation, `switch` statement enhancements and more. We've found it easier and faster to develop iOS and OS X apps in Swift than in Objective-C.
- *Performance*—Swift was designed for better performance than Objective-C. Apple has observed that Swift code is about 1.5 times faster than Objective-C code on today's multi-core systems.
- *Error Prevention*—Swift eliminates many common programming errors, making your code more robust and secure. Some of these error prevention features (which are listed in Fig. 1.2) include automatic memory management, no pointers, required braces around every control statement's body, assignment operators that do not return values, requiring initialization of all variables and constants before they're used, array bounds checking, automatic checking for overflow of integer calculations, and more.
- *Interoperability with Objective-C*—You can combine Swift and Objective-C in the same app. This enables you to enhance existing Objective-C apps without having to rewrite all the code. Your apps will easily be able to interact with the Cocoa/Cocoa Touch frameworks, which are largely written in Objective-C.
- *Playgrounds*—A playground is an Xcode window in which you can enter Swift code that compiles and executes as you type it. This allows you to see and hear your code's results as you write it, to quickly find and fix errors, and to experiment with features of Swift and the Cocoa/Cocoa Touch frameworks.

Software Used in *Swift for Programmers*

To execute our Swift examples and write your own Swift code, you must install Xcode 6, which is available 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
```

Swift Fundamentals: Parts I, II and III LiveLessons Video Training

Our *Swift Fundamentals: Parts I, II and III* LiveLessons video training product shows you what you need to know to start building robust, powerful software with Swift. It includes approximately 20 hours of expert training synchronized with *Swift for Programmers*. For additional information about Deitel LiveLessons video products, visit

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

or contact us at `deitel@deitel.com`.

You also can access our books and LiveLessons videos on Safari Books Online

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

if you have an appropriate subscription. A limited free-trial is available. Safari is popular with large companies, colleges, libraries and individuals who would like access to video training and electronic versions of print publications.

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.[2] The iPhone 5s and the iPhone 5c, released simultaneously in September 2013, sold over nine million combined in the first three days of availability.[3] 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 iPhone 5 pre-sales in its first day of pre-order availability.[4] Apple sold 10 million iPhone 6 and iPhone 6 Plus units combined in their first weekend of availability.[5]

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[6] and over 40 million worldwide by September 2011.[7] 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.[8]

There are over 1.3 million apps in the App Store[9] and over 75 billion iOS apps have been downloaded.[10] The potential for iOS app developers is enormous. It's likely that most new iOS and OS X development soon will be done in Swift, so there are great opportunities for Swift programmers.

Our Research Sources

Due to Swift's similarities with many of today's popular programming languages, we were able to repurpose and customize examples from many of our other programming textbooks and professional books. Because Swift is new, we performed most of our research using the Apple resources listed on the next page.

---

2. http://www.apple.com/pr/library/2009/07/21results.html.
3. https://www.apple.com/pr/library/2013/09/23First-Weekend-iPhone-Sales-Top-Nine-Million-Sets-New-Record.html.
4. http://techcrunch.com/2014/09/15/apple-sells-4m-iphone-6-and-6-plus-pre-orders-in-opening-24-hours/.
5. http://www.apple.com/pr/library/2014/09/22First-Weekend-iPhone-Sales-Top-10-Million-Set-New-Record.html.
6. http://www.ipadinsider.com/tag/ipad-sales-figures/.
7. http://www.statista.com/statistics/180656/sales-of-tablets-and-ipads-in-the-us-until-2012/.
8. http://www.theverge.com/2014/1/27/5350106/apple-q1-2014-earnings.
9. http://mashable.com/2014/09/09/apple-1-3-million-apps-app-store/.
10. http://techcrunch.com/2014/06/02/itunes-app-store-now-has-1-2-million-apps-has-seen-75-billion-downloads-to-date/.

- *The Swift Programming Language*—available in the iBooks store and at:

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

- *Using Swift with Cocoa and Objective-C*—available in the iBooks store and at:

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

- The *Swift Standard Library Reference*:

```
https://developer.apple.com/library/ios/documentation/General/
    Reference/SwiftStandardLibraryReference
```

- The Swift Blog:

```
https://developer.apple.com/swift/blog/
```

- World Wide Developers Conference (WWDC) 2014 videos:

```
https://developer.apple.com/videos/wwdc/2014/
```

Teaching Approach

Swift for Programmers contains numerous complete working code examples. We stress program clarity and concentrate on building well-engineered, high-performance software.

Syntax Coloring. For readability, we syntax color all the Swift code, similar to the syntax coloring in the Xcode 6 integrated-development environment. Our conventions are:

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

Code Highlighting. We place colored rectangles around key code segments.

Using Fonts for Emphasis. We place key terms and the index's page reference for each term's defining occurrence in **bold colored** 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, `println()`).

Objectives/Outline. Each chapter begins with a list of objectives and a chapter outline.

Illustrations/Figures. Abundant tables, programs and program outputs are included.

Programming Tips. We include programming tips to help you focus on important aspects of program development. These tips and practices represent the best we've gleaned from a combined eight decades of programming experience.

Good Programming Practices

The Good Programming Practices *call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

Common Programming Errors

Pointing out these Common Programming Errors *reduces the likelihood that you'll make them.*

Error-Prevention Tips
These tips contain suggestions for exposing bugs and removing them from your programs; many describe aspects of Swift that prevent bugs from getting into programs in the first place.

Performance Tips
These tips highlight opportunities for making your programs run faster or minimizing the amount of memory they occupy.

Software Engineering Observations
The Software Engineering Observations *highlight design patterns and architectural issues that affect the construction of software systems, especially large-scale systems.*

Index. We've included an extensive index. Each key term's defining occurrence is highlighted with a **bold colored** page number.

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 from `pearsonhighered.com` 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 from `pearsonhighered.com` using the following ISBNs (email `deitel@deitel.com` if you have questions):

- *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™, Volume 1, 3/e* (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.

Acknowledgments

Deitel Team

We'd like to thank Abbey Deitel and Barbara Deitel of Deitel & Associates, Inc. for long hours devoted to this project. Abbey co-authored Chapter 1 and this Preface, and she and Barbara painstakingly researched the world of Swift. Our Art Director, Jessica Deitel (age 10) chose the cover color.

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 20-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, OS X and emerging Swift communities 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 production.

Reviewers

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

- Scott Bossack, Lead iOS Developer, Thrillist Media Group
- René Cacheaux, iOS Architect, Mutual Mobile
- Ash Furrow, iOS Developer, Artsy
- Rob McGovern, Independent Contractor
- Abizer Nasir, Freelance iOS and OS X Developer, Jungle Candy Software Ltd.
- Rik Watson, Technical Team Lead for HP Enterprise Services (Applications Services)
- Jack Watson-Hamblin, Programming Writer and Teacher, MotionInMotion (`https://motioninmotion.tv/`)

A Special Thank You to Reviewer Charles Brown

When Swift was announced in June 2014, within days our publisher, Prentice Hall/Pearson, agreed to publish our Swift book, which at the time was just an idea. One key prob-

lem—where would we find Swift reviewers when the language was so new? We asked for help from our 75,000 social media and newsletter followers. Charles E. Brown, Independent Contractor affiliated with Apple and Adobe, was the first to respond and became the core member of our review team. He mentored us throughout the project, providing insights, encouragement, answers to our technical questions and appropriate cautions.

Keeping in Touch with the Authors

As you read the book, if you have questions, comments or suggestions, send an e-mail to us at

```
deitel@deitel.com
```

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

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

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

Well, there you have it! 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. We hope you enjoy working with *Swift for Programmers* as much as we enjoyed writing it!

Paul and Harvey 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 (now Oracle), 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 M. 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.

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 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 39-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, e-mail `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 to the information presented here will be posted at:

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

Conventions

Font and Naming

We use fonts to distinguish between on-screen components (such as menu names and menu items) and Swift code or commands. 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, `println()`). 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 Swift examples and write your own Swift code, 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 and OS X 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 a prerelease DMG (disk image) file, double click it to launch the installer, then follow the on-screen instructions.

Fee-Based iOS Developer Programs

In Chapters 13–14, you'll build two iOS apps and test them on your Mac using the iOS simulator that's bundled with Xcode. If you'd like to run iOS apps on actual iOS devices, you must be a member of one of the following iOS 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 and OS X Developer Program accounts 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 *Swift for Programmers* examples are available for download as a ZIP file from

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

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 `SwiftFPExamples` 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 Playgrounds and Projects for the Code Examples

Playgrounds are a new interactive coding capability in Xcode 6. They execute Swift code as you write it. They're particularly useful for learning and experimenting with Swift or the Cocoa and Cocoa Touch frameworks that are used to build iOS and OS X apps. Projects, on the other hand, are used to manage all the files for each app that you create.

For each example, we provide one of the following:

- an Xcode playground file with the `.playground` extension
- an Xcode project for an OS X **Command Line Tool** app that produces text output (such projects don't require you to develop a GUI or to run apps in the iOS simulator)
- an Xcode project for an iOS 8 app that runs in the iOS simulator bundled with Xcode.

An Xcode project is stored in a folder with the project's name. In that folder is a file with a `.xcodeproj` extension. You can double click a `.playground` or `.xcodeproj` file to open it in Xcode. Throughout this book, we use playgrounds for single-source-file examples and projects for multi-source-file examples.

Use Playgrounds for Learning

We recommend that as you learn Swift, you enter each example's code into an Xcode 6 playground so that you can immediately see the code in action as you write it. Sometimes you might need to restart the IDE if a playground stops working correctly. If you enter any of our multi-source-file examples into a playground, you must define any functions and types *before* they're used.

Viewing Output in a Playground

In a playground, the results of any output statements are visible only if the **Assistant Editor** is displayed. To open it in a playground, select **Assistant Editor > Show Assistant Editor** from Xcode's **View** menu. The **Assistant Editor** will appear at the playground window's right side.

Playground and Project Naming Conventions

Each project or playground is named based on its figure number(s) or the concept being presented. The comment in the first line of a source code file contains information to help you identifiy which playground or project to open from the chapter's examples folder:

- the project's or playground's base name—e.g., `fig02-01` and `fig03-01-11` correspond to `fig02-01.playground` and `fig03-01-11.xcodeproj`, respectively.
- the project's or playground's complete name—e.g., `CompoundInterest.playground` or `Inheritance.xcodeproj`.

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.

You're now ready to begin learning Swift with *Swift for Programmers*. We hope you enjoy the book! If you have any questions, please email us at `deitel@deitel.com`.

5

Functions and Methods: A Deeper Look; enums and Tuples

Objectives

In this chapter you'll:

- Learn about Swift modules (for software reuse).
- Define functions with multiple parameters.
- Use random-number generation to implement a game-playing app.
- Use **enum** types to create sets of named constants.
- Return multiple values from a function via a tuple, pass a tuple to a function and access a tuple's elements.
- Learn how an identifier's scope limits its visibility to specific parts of a program.
- Create overloaded functions.
- Learn how local and external parameter names are used in function and method calls.
- Use default parameter values in function calls.
- Pass method arguments by value and by reference.
- Define a recursive function.
- Define a nested function.

Outline

5.1 Introduction

We introduced functions and methods in Chapter 3. The key distinction between a function and a method is that any function defined *in a type* is a method.

In this chapter, we begin by discussing modules, which Swift uses to package related software components for reuse. We introduce Darwin—Apple's UNIX-based core of OS X and iOS—and import Darwin features (such as a C-based random-number-generation function) for use in apps.

We discuss random-number generation and develop a version of a popular casino dice game. That example demonstrates basic `enum` types for creating named constants that improve the readability of the code. You'll see that Swift's `enum` constants can have values, but that's not required. The example also presents tuples—collections of values of the same or different types. We return multiple values from a function via a tuple, pass a tuple to a function and access a tuple's elements via both names and indices.

Next, we discuss Swift's scope rules. Then, we introduce the concept of *overloading*. You'll frequently see identically named functions or, within a type, identically named methods. This overloading is used to implement functions or methods that perform similar tasks but with different types and/or different numbers of parameters. This chapter demonstrates overloading with functions, and you'll see examples of method overloading in later chapters.

We discuss the differences between calling functions and methods and present the concepts of local vs. external parameter names. As you'll see, external parameter names must be used in a function call to label all of the corresponding arguments. This is another distinction between functions and methods—by default, methods require their second and subsequent arguments to be labeled with parameter names. This has to do with the similarities between how methods are named in Objective-C and Swift, which we discuss in Section 5.9. We also mention how to disable this feature when calling methods. Parameter names are always required in initializer calls.

We use a default parameter value that the compiler inserts in a function call if you do not provide the corresponding argument when the function is called. We discuss how value- and reference-type arguments are passed to methods, then demonstrate how to pass arguments by reference using the keyword `inout`. You'll write recursive functions (functions that call themselves) and nested functions.

Many of the features presented as functions in this chapter also apply to methods and initializers in the new types you create. We'll point out key differences between functions, methods and initializers.

5.2 Modules in Swift

Swift apps are written by combining new functions and types, properties, methods, classes, `structs` (Chapter 9) and `enums` (introduced in Section 5.6 and discussed in more detail in Chapter 9) with predefined capabilities in the Swift Standard Library, the Cocoa and Cocoa Touch frameworks, and other class libraries. Figure 5.1 overviews some functions, types and protocols (similar to interfaces in other languages) from the Swift Standard Library. You can locate additional information about Swift Standard Library types and functions in the *Swift Standard Library Reference* at

```
http://bit.ly/SwiftStandardLibrary
```

At the time of this writing, the *Swift Standard Library Reference* is not yet complete. There are many other built-in free functions (sometimes called global functions), but only a few are currently listed. Similarly, there are other protocols not yet included in the reference, but mentioned in other Swift documentation (e.g., `Hashable` and `DebugPrintable`).

Feature	Description
Types	
`Array`	This type is used to represent arrays—collections of related data items. Type `Array` provides many initializers, properties, methods and operators for performing common array manipulations. Chapter 6 discusses type `Array` in detail.
`Dictionary`	A `Dictionary` maps unique *keys* to *values*—for example, an employee's ID number can be mapped to one employee's information. Type `Dictionary` provides many initializers, properties, methods and operators for performing common manipulations of key–value pairs. Chapter 7 discusses type `Dictionary` in detail.
Boolean and numeric types	As you've seen, Swift provides type `Bool` and integer and floating-point numeric types (Fig. 2.6). These are the equivalent of what many programming languages refer to as the built-in, primitive or fundamental types.
`String`	`String`s are collections of characters. Type `String` provides many initializers, properties, methods and operators for performing common `String` manipulations. We present details of type `String` throughout the book.
Protocols	
`Comparable`	An item that is `Comparable` can be compared with another item of the same type using the `<` operator. `String`s and all of Swift's integer and floating-point numeric types are `Comparable`. We discuss how to make your own types `Comparable` in Chapter 12, Operator Overloading and Subscripts.

Fig. 5.1 | Some Swift Standard Library features, (Part 1 of 2.)

Feature	Description
`Equatable`	An item that is `Equatable` can be compared with another item of the same type using the == operator. `Bool`s, `String`s and all of Swift's numeric types are `Equatable`. We discuss how to make your own types `Equatable` in Chapter 12.
`Printable`	Any item that is `Printable` has a `description` property that returns a `String` representation of the item—similar to some languages' `toString` or `ToString` methods. `Bool`s, `String`s and all of Swift's numeric types are `Printable`. We discuss how to make your own types `Printable` in Chapter 10.
Functions	
`print`, `println`	Functions that display text representations of `Printable` items.
`sort`, `sorted`	Functions that sort the contents of `Array`s—`sort` modifies the original `Array`'s contents and `sorted` returns a new `Array` containing the sorted contents. Chapter 6 uses these functions to sort `Array`s.

Fig. 5.1 | Some Swift Standard Library features. (Part 2 of 2.)

Modules

Related software components in Objective-C are grouped into frameworks (similar to namespaces or packages in other languages) so that they can be reused in Cocoa and Cocoa Touch apps. Swift's equivalent to a framework is a **module.** When you create a Swift project, Xcode places all of the project's Swift code in a module with the same name as your project. If you create a Swift-based Cocoa Framework project or Cocoa Touch Framework project, you can then reuse that framework in Cocoa and Cocoa Touch apps by importing it with the `import` keyword (as you did with the Foundation framework in Fig. 3.6).

Software Engineering Observation 5.1

Don't try to "reinvent the wheel." When possible, reuse capabilities of the Swift Standard Library, the Cocoa and Cocoa Touch frameworks, and other libraries. This reduces app development time, avoids introducing programming errors and contributes to good app performance.

5.3 Darwin Module—Using Predefined C Functions

Just as your Swift apps can reuse Cocoa and Cocoa Touch frameworks (written largely in Objective-C), they can also reuse C-based UNIX functions (such as `arc4random_uniform` in Section 5.5) and C Standard Library functions (such as the common C math functions listed in Fig. 5.2) that are built into OS X and iOS. These and many other features of UNIX and C are available via the **`Darwin` module**, which provides access to the C libraries in Darwin—Apple's open-source UNIX-based core on which the OS X and iOS operating systems are built. To import the `Darwin` module, use the following `import` declaration:

```
import Darwin
```

The `Darwin` module is imported by default into several Cocoa and Cocoa Touch frameworks—such as Foundation, AppKit and UIKit—so that various software components in those frameworks can interact with the underlying operating system.

Method	Description	Example
Throughout this table, x *and* y *are of type* `Double`		
`abs(x)`	absolute value of *x*	`abs(23.7)` is 23.7 `abs(0.0)` is 0.0 `abs(-23.7)` is 23.7
`ceil(x)`	rounds *x* to the smallest integer not less than *x*	`ceil(9.2)` is 10.0 `ceil(-9.8)` is -9.0
`cos(x)`	trigonometric cosine of *x* (*x* in radians)	`cos(0.0)` is 1.0
`exp(x)`	exponential method e^x	`exp(1.0)` is 2.71828 `exp(2.0)` is 7.38906
`floor(x)`	rounds *x* to the largest integer not greater than *x*	`floor(9.2)` is 9.0 `floor(-9.8)` is -10.0
`log(x)`	natural logarithm of *x* (base *e*)	`log(M_E)` is 1.0 `log(M_E * M_E)` is 2.0
`max(x, y)`	larger value of *x* and *y*	`max(2.3, 12.7)` is 12.7 `max(-2.3, -12.7)` is -2.3
`min(x, y)`	smaller value of *x* and *y*	`min(2.3, 12.7)` is 2.3 `min(-2.3, -12.7)` is -12.7
`pow(x, y)`	*x* raised to the power *y* (i.e., x^y)	`pow(2.0, 7.0)` is 128.0 `pow(9.0, 0.5)` is 3.0
`sin(x)`	trigonometric sine of *x* (*x* in radians)	`sin(0.0)` is 0.0
`sqrt(x)`	square root of *x*	`sqrt(900.0)` is 30.0
`tan(x)`	trigonometric tangent of *x* (*x* in radians)	`tan(0.0)` is 0.0

Fig. 5.2 | Some math functions from the C Standard Library.

5.4 Multiple-Parameter Function Definition

In previous chapters, you called functions, methods and initializers with varying numbers of arguments. You also defined functions and methods with only one parameter. In this section, we define and call a function with multiple parameters.

Figure 5.3 defines a function `maximum` (lines 4–18) that determines and returns the largest of three `Double` values. Lines 21–23 call `maximum` with the largest value (`3.3`) as the first, second or third argument, respectively, to show that the function always returns the largest of its three arguments.

```
1  // fig05-03: Function maximum with three Double parameters.
2
3  // returns the maximum of its three Double parameters
4  func maximum(x: Double, y: Double, z: Double) -> Double {
5      var maximumValue = x // assume x is the largest to start
6
```

Fig. 5.3 | Function `maximum` with three `Double` parameters. (Part 1 of 2.)

```
7       // determine whether y is greater than maximumValue
8       if y > maximumValue {
9           maximumValue = y
10      }
11
12      // determine whether z is greater than maximumValue
13      if z > maximumValue {
14          maximumValue = z
15      }
16
17      return maximumValue;
18  }
19
20  // test function maximum
21  println("Maximum of 3.3, 2.2 and 1.1 is: \(maximum(3.3, 2.2, 1.1))")
22  println("Maximum of 1.1, 3.3 and 2.2 is: \(maximum(1.1, 3.3, 2.2))")
23  println("Maximum of 2.2, 1.1 and 3.3 is: \(maximum(2.2, 1.1, 3.3))")
```

```
Maximum of 3.3, 2.2 and 1.1 is: 3.3
Maximum of 1.1, 3.3 and 2.2 is: 3.3
Maximum of 2.2, 1.1 and 3.3 is: 3.3
```

Fig. 5.3 | Function `maximum` with three `Double` parameters. (Part 2 of 2.)

Function `maximum`

Line 4 indicates that `maximum` requires three `Double` parameters (`x`, `y` and `z`) to accomplish its task and returns a `Double`. There must be one argument in the function call for each parameter in the function definition. Also, each argument must match the type of the corresponding parameter. Parameters are *constants* by default—if you need to modify a parameter's value in the function's body, you must place `var` before the parameter's name.

Common Programming Error 5.1

Declaring method parameters of the same type as `x, y: Double` *instead of* `x: Double, y: Double` *is a syntax error—a type is required for each parameter in the parameter list.*

Error-Prevention Tip 5.1

Making parameters constant by default ensures that you do not accidentally modify their values—you must explicitly opt for this functionality by declaring parameters as `var`.

Three Ways to Return Control from a Function

There are three ways to return control to the statement that calls a function. If the function's return type is `Void` (that is, it does not return a result), control returns when the function-ending right brace is reached or when the statement

```
return
```

is executed from the functions's body. If the function returns a result, the statement

`return` *expression*

evaluates the *expression*, then returns the result (and control) to the caller (as in line 17).

Swift Function **`max`**

Swift provides a **`max`** function that can be used to compare two values of the same `Comparable` type—all of Swift's numeric types and `String`s are `Comparable`. A second version of `max` takes a variable number of arguments and is used to compare three or more arguments of the same `Comparable` type. You'll create your own functions with variable-length parameter lists in Chapter 6, Arrays and an Introduction to Closures. There is no need for us to define our own `maximum` function, as we could have replaced the `maximum` calls in lines 21–23 with:

```
max(3.3, 2.2, 1.1)
max(1.1, 3.3, 2.2)
max(2.2, 1.1, 3.3)
```

5.5 Random-Number Generation

We now take a brief diversion into a popular type of programming application—simulation and game playing. In this and the next section, we develop a game-playing program with multiple functions.

The element of chance can be introduced in a program via the **`arc4random_uniform`** function (a C-based UNIX function from the `Darwin` module), which produces random unsigned 32-bit integers (`UInt32`; see Fig. 2.6) from 0 up to but not including an upper bound that you specify as an argument. There's also function **`arc4random`**, which takes no arguments and returns a random unsigned 32-bit integer in the range 0 (`UInt32.min`) to 4,294,967,295 (`UInt32.max`).

Both functions use the RC4 (also called ARCFOUR) random-number generation algorithm (`http://en.wikipedia.org/wiki/RC4`) and produce **nondeterministic random numbers** that cannot be predicted. To use these functions, you must import the `Darwin` module (Section 5.3).

Error-Prevention Tip 5.2

Functions `arc4random_uniform` and `arc4random` cannot produce repeatable random-number sequences. If you require repeatability for testing, use the `Darwin` module's C function `random` to obtain the random values and function `srandom` to seed the random-number generator with the same seed during each program execution. Once you've completed testing, use either `arc4random_uniform` or `arc4random` to produce random values.

Obtaining a Random Value with **`arc4random`**

The following statement generates a random `UInt32` value in the range 0 (`UInt32.min`) to 4,294,967,295 (`UInt32.max`):

```
let randomValue = arc4random()
```

Obtaining a Random Value in a Specific Range with **`arc4random_uniform`**

The range of values produced by `arc4random` generally differs from the range of values required in a particular app. For example, a program that simulates the rolling of a six-sided die might require random integers in the range 1–6. For cases like this, we'll use the function `arc4random_uniform`.

To demonstrate `arc4random_uniform`, let's develop a program that simulates 20 rolls of a six-sided die and displays the value of each roll. First, we use `arc4random_uniform` to produce random values in the range 0–5, as follows:

```
let face = arc4random_uniform(6)
```

The argument `6` is the upper bound of the values produced and represents the number of unique values to produce (in this case six—0, 1, 2, 3, 4 and 5).

A six-sided die has the numbers 1–6 on its faces, not 0–5. So we shift the range of numbers produced by adding 1 to our previous result, as in

```
let face = 1 + arc4random_uniform(6)
```

Rolling a Six-Sided Die 20 Times

Figure 5.4 shows two sample outputs which confirm that the results of the preceding calculation are integers in the range 1–6, and that each run of the program can produce a *different* sequence of random numbers. Line 2 imports the `Darwin` module to allow the program to access function `arc4random_uniform`—the Swift Standard Library does not have its own random-number-generation capabilities. Line 5 executes 20 times in a loop to roll the die. To run the program multiple times in a playground, simply press *Enter* on a blank line.

```
1  // fig05-04: Shifted and scaled random integers
2  import Darwin  // allow program to use C function arc4random_uniform
3
4  for i in 1...20 {
5      print("\(1 + arc4random_uniform(6)) ")
6  }
```

```
3 3 3 1 1 2 1 2 4 2 2 3 6 2 5 3 4 6 6 1
```

```
6 2 5 1 3 5 2 1 6 5 4 1 6 1 3 3 1 4 3 4
```

Fig. 5.4 | Shifted and scaled random integers.

5.6 Introducing Enumerations and Tuples

One popular game of chance is the dice game known as “craps.” In this section, we implement a simple version of the game and introduce Swift’s `enum` and tuple features.

The rules of the game are straightforward:

> *You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called “craps”), you lose (i.e., “the house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your “point.” To win, you must continue rolling the dice until you “make your point” (i.e., roll that same point value). You lose by rolling a 7 before making your point.*

The app in Fig. 5.5 simulates the game of craps. Lines 31–74 of the program play the game. The `rollDice` function (lines 19–23) is called to roll the two dice and compute their sum, and the `displayRoll` function (lines 26–28) is called to display the results of a roll. The four sample outputs show winning on the first roll, losing on the first roll, winning on a subsequent roll and losing on a subsequent roll, respectively.

```
// fig05-05: Simulating the dice game craps
import Darwin

// enum representing game status constants (no raw type)
enum Status {
    case Continue, Won, Lost
}

// enum with Int constants representing common dice totals
enum DiceNames: Int {
    case SnakeEyes = 2
    case Trey = 3
    case Seven = 7
    case YoLeven = 11
    case BoxCars = 12
}

// function that rolls two dice and returns them and their sum as a tuple
func rollDice() -> (die1: Int, die2: Int, sum: Int) {
    let die1 = Int(1 + arc4random_uniform(6)) // first die roll
    let die2 = Int(1 + arc4random_uniform(6)) // second die roll
    return (die1, die2, die1 + die2)
}

// function to display a roll of the dice
func displayRoll(roll: (Int, Int, Int)) {
    println("Player rolled \(roll.0) + \(roll.1) = \(roll.2)")
}

// play one game of craps
var myPoint = 0 // point if no win or loss on first roll
var gameStatus = Status.Continue // can contain Continue, Won or Lost

var roll = rollDice() // first roll of the dice
displayRoll(roll) // display the two dice and the sum

// determine game status and point based on first roll
switch roll.sum {
    // win on first roll
    case DiceNames.Seven.rawValue, DiceNames.YoLeven.rawValue:
        gameStatus = Status.Won
    // lose on first roll
    case DiceNames.SnakeEyes.rawValue, DiceNames.Trey.rawValue,
         DiceNames.BoxCars.rawValue:
        gameStatus = Status.Lost
    // did not win or lose, so remember point
    default:
        gameStatus = Status.Continue // game is not over
        myPoint = roll.sum // remember the point
        println("Point is \(myPoint)")
}
```

Fig. 5.5 | Simulating the dice game craps. (Part 1 of 2.)

```
// while game is not complete
while gameStatus == Status.Continue
{
    roll = rollDice() // first roll of the dice
    displayRoll(roll) // display the two dice and the sum

    // determine game status
    if roll.sum == myPoint { // won by making point
        gameStatus = Status.Won
    } else {
        if (roll.sum == DiceNames.Seven.rawValue) { // lost by rolling 7
            gameStatus = Status.Lost
        }
    }
}

// display won or lost message
if gameStatus == Status.Won {
    println("Player wins")
} else {
    println("Player loses")
}
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 2 + 1 = 3
Player loses
```

```
Player rolled 2 + 4 = 6
Point is 6
Player rolled 3 + 1 = 4
Player rolled 5 + 5 = 10
Player rolled 6 + 1 = 7
Player loses
```

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 1 + 3 = 4
Player rolled 1 + 3 = 4
Player rolled 2 + 3 = 5
Player rolled 4 + 4 = 8
Player rolled 6 + 6 = 12
Player rolled 4 + 4 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 6 = 8
Player rolled 6 + 6 = 12
Player rolled 6 + 4 = 10
Player wins
```

Fig. 5.5 | Simulating the dice game craps. (Part 2 of 2.)

The Game's Logic

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Lines 31–74 contain the logic for one complete game of craps. Variable `myPoint` (line 31) stores the "point" if the player does not win or lose on the first roll. Variable `gameStatus` (line 32) maintains the game status. Variable `roll` (created at line 34 and assigned a new value at line 56) stores the most recent roll of the dice. Variable `myPoint` is initialized to `0` so the program can compile. If you do not initialize `myPoint`, the compiler issues an error, because `myPoint` is not assigned a value in every `case` of the `switch` statement—thus, the app could try to use `myPoint` before it's assigned a value. By contrast, `gameStatus` does not require initialization because it's assigned a value in every branch of the `switch` statement—thus, it's guaranteed to be initialized before it's used.

Error-Prevention Tip 5.3

Initialize every variable when it's defined.

The First Roll

Line 34 calls function `rollDice`, which picks two random values from 1 to 6 and returns both values and their sum. Line 35 calls function `displayRoll` to display the value of the first die, the value of the second die and the sum of the dice. We explain the details of `rollDice`'s return value and `displayRoll`'s argument in Sections 5.6.2 and 5.6.3, respectively. Next, the program enters the `switch` statement at lines 38–51, which uses the sum of the dice to determine whether the game has been won or lost, or whether it should continue with another roll.

Additional Rolls of the Dice

If we're still trying to "make our point" (i.e., the game is continuing from a prior roll), the loop in lines 54–67 executes. Line 56 rolls the dice again. Lines 60–66 determine whether the game was won or lost on the most recent roll—if not, the game continues. When the game completes, lines 70—74 display a message indicating whether the player won or lost, and the app terminates.

5.6.1 Introducing Enumeration (`enum`) Types

In this section, we introduce basic enumeration features—more details are presented in Chapter 9, Structures, Enumerations and Nested Types.

`Status` *Enumeration*

The `Status` type (lines 5–7) is an **enumeration** that declares a set of constants represented by identifiers. An enumeration is introduced by the keyword **`enum`** and a type name (in this case, `Status`). As with a class, braces (`{` and `}`) delimit the `enum`'s body. Inside the braces is a `case` containing a comma-separated list of **enumeration constants**. The `enum` constant names must be unique. Unlike `enum`s in other C-based programming languages, a Swift `enum`'s constants do not have values by default—the constants themselves are the values. Sometimes it's useful for each constant to have a so-called raw value, as in the `DiceNames` `enum` (lines 10–16) that we discuss momentarily.

Variables and constants of type `Status` can be assigned only constants defined in the `Status` `enum`. When the game is won, the app sets variable `gameStatus` to `Status.Won` (lines 41 and 61). When the game is lost, the app sets `gameStatus` to `Status.Lost` (lines 45 and 64). Otherwise, the app sets `gameStatus` to `Status.Continue` (line 48) to indicate that the dice must be rolled again. If a variable has an `enum` type, you can assign `enum` constants to the variable using the shorthand notation:

```
variableName = .EnumConstantName
```

Good Programming Practice 5.1

`enum` constant names should begin with a capital letter and use camel-case naming.

`DiceNames` Enumeration

The sums of the dice that would result in a win or loss on the first roll are declared in the `DiceNames` enumeration in lines 10–16. These are used in the `case`s of the `switch` statement (lines 38–51). The identifier names use casino parlance—such as snake eyes (2) and box cars (12)—for these sums. In `DiceNames` we explicitly assign a value to each constant's name. When an `enum`'s constants require values (known as **raw values**), you must specify the `enum`'s **raw type**—that is, the type used to represent each constant's value. Line 10 indicates that `DiceNames`'s raw type is `Int`, so each constant's type is also `Int`. The raw type can be any of Swift's numeric types, type `String` or type `Character`.

Constants that are assigned explicit values are typically defined in a separate `case`s for readability (as in lines 11–15), but this is not required. We could have written the `DiceNames` enumeration as:

```
enum DiceNames: Int {
    case SnakeEyes = 2, Trey = 3, Seven = 7, YoLeven = 11,
        BoxCars = 12
}
```

If an `enum` type's constants represent sequential integer values, they can be defined as a comma-separated list in one `case`, as in:

```
enum Months: Int {
    case January = 1, February, March, April, May, June, July,
        August, September, October, November, December
}
```

In `Months`, each subsequent constant after `January` has a value one higher than the value of the previous constant, so `February` is `2`, `March` is `3`, etc. So, we could have defined the `DiceNames` constants `SnakeEyes` and `Trey` in one `case` as:

```
case SnakeEyes = 2, Trey
```

The raw values of an `enum`'s constants must be unique. In an `enum` with one of the integer numeric types, if the first constant is unassigned, the compiler gives it the value `0`.

Good Programming Practice 5.2

Using enumeration constants (like `Months.January`, `Months.February`, etc.) rather than literal integer values (such as 1, 2, etc.) makes code easier to read and maintain.

5.6.2 Tuples and Multiple Function Return Values

In the rules of the game, the player must roll two dice on the first roll and must do the same on all subsequent rolls. Function `rollDice` (lines 19–23) rolls the dice and computes their sum. Function `rollDice` is declared once, but it's called from two places (lines 34 and 56). The function takes no arguments. Each time it's called, `rollDice` returns *three values* (the two die values and the sum of the dice) as a **tuple**—an arbitrary collection of values that can be of the same or different types. In function `rollDice`'s return type

```
(die1: Int, die2: Int, sum: Int)
```

`die1`, `die2` and `sum` are names that can be used to access the returned tuple's elements.

Good Programming Practice 5.3

You're not required to specify names for each element of a tuple, but doing so makes the code more readable.

The sum of the dice can be calculated using the values of the tuple elements `die1` and `die2`. We chose to include `sum` in the tuple because there are multiple locations in the program where we use the sum of the dice. Rather than recalculating the sum each time, we calculate it once in `rollDice`, return it as part of the tuple, then simply use the tuple's `sum` element as necessary in the rest of the code.

Composing a Tuple

To return a tuple containing multiple values from a function, you **compose** it by wrapping the values in parentheses, as in the `return` statement (line 22).

Accessing a Tuple's Elements

When a tuple specifies names for its elements, you can access them by name using the dot (`.`) syntax. Line 34 assigns the tuple returned by `rollDice` to the variable `roll`, which is inferred to have the tuple type `(Int, Int, Int)`. The `switch` statement's control expression (line 38) uses `roll.sum` to get the sum of the dice from the returned tuple.

Decomposing a Tuple

You can also **decompose** a tuple into individual variables or constants. For example, the statement

```
let (die1, die2, sum) = rollDice()
```

assigns the three values in the tuple to the constants `die1`, `die2` and `sum`, respectively. When decomposing a tuple, if you need only some of the values, you can ignore individual values with the underscore character (_), as in:

```
let (_, _, sum) = rollDice()
```

Explicit Casts Are Required for Numeric Conversions

Unlike many other programming languages, Swift does *not* allow implicit conversions between numeric types. To prevent a compilation error when you use a value of one numeric type where a different numeric type is expected, the compiler requires you to cast the value to the required type to force the conversion. This enables you to "take control" from the compiler. You essentially say, "I know this conversion might lose information, but for my purposes here, that's fine."

Function `rollDice` returns a tuple containing `Int` values; however, the random numbers returned by function `arc4random_uniform` are of type `UInt32`. To convert these to type `Int`, you must use an `Int` cast as shown in line 20:

```
let die1 = Int(1 + arc4random_uniform(6)) // first die roll
```

The cast `Int(1 + arc4random_uniform(6))` creates a *temporary* `Int` copy of the argument in parentheses.

Error-Prevention Tip 5.4

Each numeric type represents a different range of values. Disallowing implicit conversions—thus forcing you to use explicit casts for numeric conversions—prevents unintentional conversions between types. This is another Swift feature that eliminates errors.

Common Programming Error 5.2

Converting a numeric-type value to a value of another numeric type may change the value. For example, converting a `Double` value to an `Int` value may introduce truncation errors (loss of the fractional part) in the result.

5.6.3 Tuples as Function Arguments

After each call to `rollDice`, the program calls function `displayRoll` (lines 35 and 57) to display the two die values and the sum of the dice. The function (lines 26–28) receives one parameter (roll) which has the tuple type `(Int, Int, Int)`. In this case, we did not specify names for the elements in the tuple, so that we could show accessing a tuple's members using indices and dot syntax, as in line 27. The first tuple element has index `0`, so `roll.0` evaluates to the first die's value, `roll.1` evaluates to the second die's value and `roll.2` evaluates to their sum.

5.6.4 Accessing the Raw Value of an enum Constant

The `switch` statement at lines 38–51 performs its tasks based on the sum of the dice. Swift does not provide implicit conversions between `enum` constants and numeric types. However, each `enum` constant has a **`rawValue`** property that returns the constant's raw value. Lines 40, 43 and 44 compare the `Int` sum of the dice to the raw `Int` values of several `DiceNames` constants to determine whether the game was won or lost on the first roll. We use the raw `enum` constant values in this case because there are several sums (4, 5, 6, 8, 9 and 10) that don't correspond to the `DiceName` `enum` constants.

Converting a Value to an `enum` Constant

You can use an `enum`'s initializer to get the `enum` constant that corresponds to a raw value. For example, using the `Months` `enum` discussed in Section 5.6.1, the expression

```
Months(rawValue: 2)
```

returns the `enum` constant `Months.February`. In a program that receives a month as a value in the range 1–12, you could use the `Months` `enum`'s initializer to convert those values to the corresponding `Months` `enum` constants for use in a `switch`'s `cases`. Because the argument could be invalid, the actual value returned by the initializer is a `Months?`—an optional value of type `Months`. We discuss this in more depth in Section 9.3.3.

5.7 Scope of Declarations

You've seen declarations of Swift entities, such as classes, methods, properties, variables and parameters. Declarations introduce names that can be used to refer to such Swift entities. The **scope** of a declaration is the portion of the code that can refer to the declared entity by its unqualified name. Such an entity is said to be "in scope" for that portion of the app. This section introduces several important scope issues. The basic scope rules are:

1. The scope of a parameter is the body of the method in which the declaration appears.
2. The scope of a local variable or constant is from the point at which it's defined to the closing right brace (`}`) of the block containing the definition.
3. The scope of a local variable that appears in the initialization section of a `for` statement's header is the body of that `for` statement and the other expressions in the header.
4. The scope of a local variable that receives each value in a `for…in` statement is the body of that `for…in` statement.
5. The scope of a method or property of a class is the entire body of the class.
6. A type, function, variable or constant defined outside any other language element has global scope from its point of definition to the end of the file in which the type, function, variable or constant is defined. Types and functions also have module scope—by default, they can be used from other files in the same module or in other apps that import that module, unless they're declared `private`.

Any block may contain variable declarations. If a local variable, constant or parameter in a method has the same name as a property of a class, the property is hidden until the block terminates. In Chapter 8, we discuss how to access hidden properties via the keyword `self`. The app in Fig. 5.6 demonstrates the scopes for a global variable, a property of a class and local variables in methods.

```
// fig05-06: Demonstrating scopes
var x = 5 // global variable x

class Scope {
    var x = 1 // property hides global variable x in class Scope

    // create and initialize local variable x during each call
    func useLocalVariable()
    {
        var x = 25 // initialized each time useLocalVariable is called

        println("\nlocal x on entering useLocalVariable is \(x)")
        ++x // modifies this method's local variable x
        println("local x before exiting useLocalVariable is \(x)")
    }

```

Fig. 5.6 | Demonstrating scopes. (Part 1 of 2.)

```
17      // modify class Scope's property x during each call
18      func useProperty() {
19          println("\nproperty x on entering useProperty is \(x)")
20          x *= 10 // modifies class Scope's property x
21          println("property x before exiting useProperty is \(x)")
22      }
23  }
24
25  var scope = Scope() // create a Scope object
26
27  println("global variable x when program begins execution is \(x)")
28
29  scope.useLocalVariable()
30  scope.useProperty()
31  scope.useLocalVariable()
32  scope.useProperty()
33
34  println("\nglobal variable x before program terminates is \(x)")
```

```
global variable x when program begins execution is 5

local x on entering useLocalVariable is 25
local x before exiting useLocalVariable is 26

property x on entering useProperty is 1
property x before exiting useProperty is 10

local x on entering useLocalVariable is 25
local x before exiting useLocalVariable is 26

property x on entering useProperty is 10
property x before exiting useProperty is 100

global variable x before program terminates is 5
```

Fig. 5.6 | Demonstrating scopes. (Part 2 of 2.)

Line 2 defines and initializes the global variable `x` to `5`. This variable is hidden in any block or method that declares local variable named `x` and in any class that defines a property named `x`. Class `Scope` (lines 4–23) defines a property `x` with the value 1 (line 5). We defined the class after the global variable `x` at line 2 to show that the class's property `x` hides the global variable.

Line 25 defines an object of class `Scope` named `scope`. Line 27 outputs the value of global variable `x` (whose value is `5`). Next, lines 29–32 call `Scope` methods `useLocalVariable` (lines 8–15) and `useProperty` (lines 18–22) that each take no arguments and do not return results. We call each method twice. Method `useLocalVariable` declares local variable `x` (line 10). When `useLocalVariable` is first called (line 29), it creates local variable `x` and initializes it to `25` (line 10), outputs the value of `x` (line 12), increments `x` (line 13) and outputs the value of `x` again (line 14). When `useLocalVariable` is called a second time (line 31), it re-creates local variable `x` and reinitializes it to `25`, so the output of each `useLocalVariable` call is identical.

Method `useProperty` does not declare any local variables. Therefore, when it refers to `x`, class `Scope`'s property `x` (line 5) is used. When method `useProperty` is first called (line 30), it outputs the value (1) or property x (line 19), multiplies the property x by 10 (line 20) and outputs the value (10) of property x again (line 21) before returning. The next time method useProperty is called (line 32), the property has its modified value, 10, so the method outputs 10, then 100. The app outputs the value of global variable x again (line 34) to show that none of the method calls modified the global variable x, because the methods all referred to variables or properties named x in other scopes.

5.8 Function and Method Overloading

You can define functions of the same name, as long as they have different sets of parameters (determined by the number, types and order of the parameters). This is called **function overloading** and can be used with a type's methods and initializers as well. When an overloaded function is called, the Swift compiler selects the appropriate function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions with the *same name* that perform the same or similar tasks, but on *different types* or *different numbers of arguments*. For example, Swift function max is overloaded with two versions—one that returns the maximum of two values and one that returns the maximum of three or more values. Our next example demonstrates declaring and invoking overloaded functions. You'll see examples of overloaded initializers in Chapter 8, Classes: A Deeper Look and Extensions.

Declaring Overloaded Functions

In Fig. 5.7, we define overloaded versions of function square—one that calculates the square of an Int (and returns an Int) and one that calculates the square of a Double (and returns a Double). Although these functions have the same name and similar parameter lists and bodies, you can think of them simply as *different* methods. It may help to think of the functions names as "square of Int" and "square of Double," respectively.

```
// fig05-07: Overloaded function definitions

// square function with Int argument
func square(value: Int) -> Int
{
    println("Called square with Int argument: \(value)")
    return value * value
}

// square function with Double argument
func square(value: Double) -> Double
{
    println("Called square with Double argument: \(value)")
    return value * value
}

// test overloaded square functions
println("Square of Int 7 is \(square(7))\n")
println("Square of Double 7.5 is \(square(7.5))")
```

Fig. 5.7 | Overloaded function definitions. (Part 1 of 2.)

```
Called square with Int argument: 7
Square of Int 7 is 49

Called square with Double argument: 7.5
Square of Double 7.5 is 56.25
```

Fig. 5.7 | Overloaded function definitions. (Part 2 of 2.)

Line 18 invokes method `square` with the argument `7`. Literal integer values are treated as type `Int`, so the method call in line 18 invokes the version of `square` at lines 4–8 that specifies an `Int` parameter. Similarly, line 19 invokes `square` with the argument `7.5`. Literal floating-point values are treated as type `Double`, so the method call in line 19 invokes the version of `square` at lines 11–15 that specifies a `Double` parameter. Each function first outputs a line of text to prove that the proper function was called in each case.

The overloaded functions in Fig. 5.7 perform the same calculation, but with two different types. Swift's generics feature provides a mechanism for writing a single "generic function" that can perform the same tasks as an entire set of overloaded functions. We discuss generic functions in Chapter 11, Generics.

Distinguishing Between Overloaded Functions

The compiler distinguishes overloaded functions by their **signature**—a combination of the function's name and the number, types and order of its parameters. The signature also includes the way those parameters are passed, which can be modified by the `inout` keyword (discussed in Section 5.11). If the compiler looked only at method names during compilation, the code in Fig. 5.7 would be *ambiguous*—the compiler would not know how to distinguish between the `square` functions. Internally, the compiler uses signatures to determine whether functions are unique, whether a class's methods are unique and whether a class's initializers are unique.

For example, in Fig. 5.7, the compiler will use the function signatures to distinguish between the "`square` of `Int`" function (the `square` function that specifies an `Int` parameter) and the "`square` of `Double`" function (the `square` function that specifies a `Double` parameter). If a function `someFunction`'s declaration begins as

```
func someFunction(a: Int, b: Double)
```

then that function will have a different signature than the function declared as

```
func someFunction(a: Double, b: Int)
```

The order of the parameter types is important—the compiler considers the preceding two functions to be distinct.

Return Types of Overloaded Functions

In discussing the logical names of functions used by the compiler, we did not mention the return types of the functions. This is because function calls cannot be distinguished by return type. Overloaded functions can have the *same* or *different* return types if the functions have *different* parameter lists. Also, overloaded functions need not have the same number of parameters.

5.9 External Parameter Names

By default, the parameter names you specify in a function definition are local to that function—they're used only in the body of that function to access the function's argument values. You can also define **external parameter names** that the caller is required to use when a function is called—as is the case for all the arguments to an initializer and any arguments after the first argument in a method call. This can help make the meaning of each argument clear to the programmer calling the function.

For each parameter, you can specify both an external name and a local name by placing the external name before the local name as in:

externalName localName: *type*

or you can specify that the local parameter name should also be used as the external parameter name by placing a # before the local parameter name, as we demonstrate in Fig. 5.8 (line 4). The function `power` (lines 4–12) calculates the value of its `base` argument raised to its `exponent` argument. The two calls to `power` (lines 15 and 16) each specify the parameter name before each argument. Once you expose an external parameter name, you must label the corresponding argument in a function call with a parameter name and a colon (:); otherwise, a compilation error occurs.

```
1  // fig05-08: External parameter names
2
3  // use iteration to calculate power of base raised to the exponent
4  func power(#base: Int, #exponent: Int) -> Int {
5      var result = 1;
6
7      for i in 1...exponent {
8          result *= base
9      }
10
11     return result
12 }
13
14 // call power with and without default parameter values
15 println("power(base: 10, exponent: 2) = \(power(base: 10, exponent: 2))")
16 println("power(base: 2, exponent: 10) = \(power(base:2, exponent: 10))")
```

```
power(base: 10, exponent: 2) = 100
power(base: 2, exponent: 10) = 1024
```

Fig. 5.8 | External parameter names.

Changing the Default External Parameter Names for an Initializer or Method

By default, the names of an initializer's parameters and the names of a method's parameters for every parameter after the first are used as their external names. You can customize a method's or initializer's external parameter names by specifying your own, using the same syntax we discussed for functions earlier in this section.

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

In Objective-C, method calls read like sentences. The method name refers to the first parameter, and each subsequent parameter has a name that's specified as part of the method call. In addition, method and parameter names often include prepositions to help make function calls read like sentences.

Apple wants Swift programmers to use similar naming conventions in their methods. Because the method name should refer to the first parameter, Swift provides only a local parameter name for the first method parameter, then provides local and external parameter names for all subsequent parameters. Using this naming convention, we could reimplement the `power` function as

```
func raiseBase(base: Int, #toExponent: Int) -> Int
```

In this case, we'd call the function as:

```
raiseBase(10, toExponent: 2)
```

which reads like the sentence, "Raise the base 10 to the exponent 2."

Requiring an External Parameter Name for a Method's First Argument

You can require a method's caller to provide an external parameter name for the method's first argument. To do so, simply precede the parameter name with `#` to use the local parameter name as the external parameter name or specify an external parameter name.

Passing Method Arguments Without Parameter Names

You can allow a method to be called without labeling its arguments by using an underscore (_) as each parameter's external name.

5.10 Default Parameter Values

Methods can have **default parameters** that allow the caller to vary the number of arguments to pass. A default parameter specifies a **default value** that's assigned to the parameter if the corresponding argument is omitted.

You can create functions with one or more default parameters. *All default parameters must be placed to the right of the function's nonoptional parameters*—that is, at the end of the parameter list. Each default parameter must specify a default value by using an equal (=) sign followed by the value.

When a parameter has a default value, the caller can optionally pass that particular argument. For example, the function

```
func power(base: Int, exponent: Int = 2) -> Int
```

specifies a default second parameter. Any call to `power` must pass at least an argument for the parameter `base`, or a compilation error occurs. Optionally, a second argument (for the `exponent` parameter) can be passed to `power`. Consider the following calls to `power`:

```
power() // compilation error--first argument is required
power(10) // calls power with 2 as the second argument
power(10, exponent: 3) // explicitly specifying both arguments
```

The first call generates a compilation error because this function requires a minimum of one argument. The second call is valid because the one required argument (`10`) is being passed explicitly—the optional `exponent` is not specified in the method call, so `2` is passed by default. The last call is also valid—`10` is passed as the required argument and `3` is passed as the optional argument. A function's default parameter names are automatically external parameter names—when you provide an argument for a default parameter, you *must* specify the default parameter's name with that argument in the function call.

Figure 5.9 demonstrates a default parameter. The program reimplements the `power` function of Fig. 5.8 without external parameter names and with a default value for its second parameter. Lines 15–16 call function `power`. Line 15 calls it without the second argument. In this case, the compiler provides the second argument, `2`, using the default value specified in line 4, which is not visible to you in the call. Notice that the call to `power` at line 16 requires the parameter name for the second argument.

```
1   // fig05-09: Default parameter values
2
3   // use iteration to calculate power of base raised to the exponent
4   func power(base: Int, exponent: Int = 2) -> Int {
5       var result = 1;
6
7       for i in 1...exponent {
8           result *= base
9       }
10
11      return result
12  }
13
14  // call power with and without default parameter values
15  println("power(10) = \(power(10))")
16  println("power(2, exponent: 10) = \(power(2, exponent: 10))")
```

```
power(10) = 100
power(2, exponent: 10) = 1024
```

Fig. 5.9 | Default parameter values.

5.11 Passing Arguments by Value or by Reference

Swift allows you to pass arguments to functions by value or by reference. When an argument is passed by *value* (the default for value types in Swift), a *copy* of its value is made and passed to the called function. Changes to the copy do *not* affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument that's been passed in the programs in this chapter so far has been passed by value. When an argument is passed by *reference*, the caller gives the function the ability to access and modify the caller's original variable.

To pass an object of a class type by reference into a function, simply provide as an argument in the function call the variable that refers to the object. Then, in the function body, reference the object using the parameter name. The parameter refers to the original object in memory, so the called function can access the original object directly.

We've considered value types and reference types. A major difference between them is that value-type variables store *values*, so specifying a value-type variable in a function call passes a *copy* of that value to the method. Reference-type variables store *references to objects*, so specifying a reference-type variable as an argument passes the function a *copy of the reference* that refers to the object. Even though the reference itself is passed by value, the function can still use the reference it receives to interact with—and possibly modify—the original object. Similarly, when returning information from a function via a `return` statement, the function returns a *copy* of the value stored in a value-type variable or a copy of the reference stored in a reference-type variable. When a reference is returned, the caller can use that reference to interact with the returned reference-type object.

`inout` *Parameters*

What if you would like to pass a variable by reference so the called function can modify the variable's value in the *caller*? To do this, Swift provides keyword **`inout`**. Applying `inout` to a parameter declaration allows you to pass a variable to a function *by reference*—the called function will be able to modify the original variable in the caller. It's a compilation error to pass a constant to an `inout` parameter. A function can use multiple `inout` parameters as another way to "return" multiple values to a caller. You can also pass a reference-type variable by reference, which allows you to modify it so that it refers to a *new* object.

Demonstrating an `inout` *Parameter*

The app in Fig. 5.10 uses the `inout` keyword to allow a function to modify its `Int` argument. Function `square` (lines 4–6) multiplies its parameter `value` by itself and assigns the result to `value`. The `Int` parameter is preceded with `inout`, which indicates that the argument passed to this method must be an `Int` and that it will be passed by reference. Because the argument is passed by reference, the assignment at line 5 modifies the original argument's value in the caller.

```
1  // fig05-10: Pass-by-reference with inout parameters
2
3  // square function that modifies its argument in the caller
4  func square(inout value: Int) {
5     value *= value // squares value of caller's variable
6  }
7
8  // test inout parameter
9  var x = 5
10 println("Original value of x is \(x)")
11 square(&x)
12 println("Value of x after calling square(&x) is \(x)")
```

```
Original value of x is 5
Value of x after calling square(&x) is 25
```

Fig. 5.10 | Pass-by-reference with `inout` parameters.

Passing an Argument by Reference

Line 9 initializes variable `x` to `5`. Line 10 displays `x`'s original value. When you pass a variable to a method with a reference parameter, you must precede the argument with an `&` (line 11)—similar to a pointer in languages like Objective-C, C and C++. After line 11 squares `x`'s value, line 12 displays the new value. Notice that `x` is now `25`.

Software Engineering Observation 5.2

By default, value types are passed by value. Objects of reference types are not passed to methods; rather, references to objects are passed to methods. The references themselves are passed by value. When a method receives a reference to an object, the method can manipulate the object directly, but the reference value cannot be changed to refer to a new object.

5.12 Recursion

Swift supports recursion. A **recursive function** calls itself, either *directly* or *indirectly through another function*.

Recursive Factorial Calculations

Figure 5.11 uses recursion to calculate and display the factorials of the integers from 0 to 10. The recursive function `factorial` (lines 4–11) first tests to determine whether a terminating condition (line 6) is `true`. If `number` is less than or equal to `1` (the base case), `factorial` returns `1`, no further recursion is necessary and the function returns. If `number` is greater than `1`, line 9 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`, which is a slightly simpler problem than the original calculation, `factorial(number)`.

```
1  // fig05-12: Recursive factorial function
2
3  // recursive factorial function
4  func factorial(number: Int64) -> Int64 {
5      // base case
6      if number <= 1 {
7          return 1
8      } else { // recursion step
9          return number * factorial(number - 1)
10     }
11 }
12
13 // calculate the factorials of 0 through 10
14 for counter in 0...10 {
15     println("\(counter)! = \(factorial(Int64(counter)))")
16 }
```

```
0! = 1
1! = 1
2! = 2
```

Fig. 5.11 | Recursive `factorial` function. (Part 1 of 2.)

```
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Fig. 5.11 | Recursive `factorial` function. (Part 2 of 2.)

Function `factorial` receives a parameter of type `Int64` and returns a result of type `Int64`. As you can see in Fig. 5.11, factorial values become large quickly. We chose `Int64` (which can represent relatively large integers) so that the app could calculate factorials up to 20!. Unfortunately, the function produces large values so quickly that 21! exceeds the maximum value that can be stored in an `Int64` variable, causing an overflow. Due to the restrictions on the integral types, variables of type `Float` or `Double` might ultimately be needed to calculate factorials of larger numbers.

A strength of object-oriented programming languages like Swift is that they can be extended with new types to meet your applications' needs. For example, you could create a type (e.g., `HugeInt`) that supports arbitrarily large integers for use in large-number factorial calculations.

Common Programming Error 5.3

Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause infinite recursion, eventually exhausting memory. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

5.13 Nested Functions

You can nest function definitions in other function definitions. This can be useful for organizing complex functions. Rather than defining at global scope a utility (helper) function that's called by only one other function, you can nest the utility function's definition in the scope of the function that uses it. This hides it from the rest of your code. For example, an array-sorting function could define a nested `swap` function for swapping elements into sorted order.

If necessary, an enclosing function can return a nested function so that it can be called from other scopes—for example, you could define a function that returns a nested function based on a value passed to the enclosing function (as we do in this section's example). A nested function also has access to the local variables and constants in its enclosing function's scope, including the enclosing function's parameters.

Figure 5.12 contains a mechanical nested-functions example. Function `sortOrder` (lines 4–16), based on the `Bool` parameter `increasingOrder`'s value, returns either the nested function `ascending` (defined at lines 6–8) or the nested function `descending` (defined at lines 11–13). To make the purpose of `sortOrder`'s argument clear, we specified that its parameter name (`increasingOrder`) should also be its external parameter name—thus, each call to `sortOrder` (lines 19 and 28) labels its argument with `increasingOrder`.

```
// fig05-12: Mechanical example of nested functions

// return a function that determines the ordering of two Ints
func sortOrder(#increasingOrder: Bool) -> (Int, Int) -> Bool {
    // return true if x and y are in ascending order
    func ascending(x: Int, y: Int) -> Bool {
        return x < y
    }

    // return true if x and y are in descending order
    func descending(x: Int, y: Int) -> Bool {
        return x > y
    }

    return (increasingOrder ? ascending : descending)
}

// get function for comparing Ints to see if they're in ascending order
var order = sortOrder(increasingOrder: true)

if order(7, 5) {
    println("7 and 5 are in ascending order")
} else {
    println("7 and 5 are not in ascending order")
}

// get function for comparing Ints to see if they're in descending order
order = sortOrder(increasingOrder: false)

if order(7, 5) {
    println("7 and 5 are in descending order")
} else {
    println("7 and 5 are not in descending order")
}
```

```
7 and 5 are not in ascending order
7 and 5 are in descending order
```

Fig. 5.12 | Mechanical example of nested functions.

Every Function Has a Type and Can Be Treated as Data

Each function you define has a type that's determined by the types of its parameters and by its return type. The return type of function `sortOrder` is specified as

```
(Int, Int) -> Bool
```

A function type consists of parentheses containing the parameter types, followed by -> and the return type. The preceding type indicates that the value returned by `sortOrder` is a function type for a function that receives two `Int` parameters and returns a `Bool`. Functions `ascending` and `descending` meet these requirements.

Because every function has a type, you can assign functions to variables, pass them to functions and methods, and return them from functions and methods. We'll discuss functions as data in more detail in Section 6.7.

Assigning a Function to a Variable and Using the Variable to Call the Function

Line 19 calls function `sortOrder` with the argument `true` to indicate that `sortOrder` should return the function that determines whether two `Int`s are in ascending order. The returned function is assigned to the variable `order`, which is inferred to have the type

```
(Int, Int) -> Bool
```

Once you've assigned a function to a variable, you can use the variable to call the function, as shown in line 21. Line 28 calls `sortOrder` with the argument `false` to get the function that determines whether two `Int`s are in descending order, then line 30 calls that function.

5.14 Wrap-Up

In this chapter, we continued our discussion of functions and methods. We discussed that Swift automatically creates modules for packaging reusable software components. We introduced Darwin—Apple's UNIX-based core of OS X and iOS—and imported the Darwin module so we could use the random-number generator.

We used `enum` types to create sets of named constants with and without values for the constants. You returned multiple values from a function via a tuple, passed a tuple to a function and accessed a tuple's elements via both names and indices.

We discussed the scope of identifiers. We used overloading to define multiple functions with the same name that performed similar tasks but with different types and/or different numbers of parameters.

We discussed differences in how functions and methods are called, and we presented the concepts of local parameter names vs. external parameter names. You saw that, when external parameter names are provided in a function definition, they must be used in the function call to label the corresponding arguments. You used # to expose a local parameter name as the external parameter name. We also showed how to disable this feature in methods—by placing an underscore (_) before the parameter's name—so that parameter names are not required in a method call.

You specified a default parameter value and saw that the compiler supplied that value in a function call when you did not explicitly provide an argument for that parameter.

We discussed how value-type and reference-type arguments are passed to methods and demonstrated how to pass arguments by reference by declaring the parameter as `inout` and providing an ampersand (&) before the corresponding argument in a function call. We demonstrated that Swift supports recursive functions and nested functions.

In Chapter 6, you'll use `Array`s to maintain lists and tables of data. You'll also create functions with variable-length argument lists. We'll continue our discussion of functions in Chapter 6 which also presents closures—anonymous functions that are typically defined in the scope of a function or method and commonly passed as arguments to other functions or methods. As you'll see, a function is a closure with a name. Swift's `Array` type provides methods `filter`, `map` and `reduce` that receive closures as arguments—which enable you to express complex operations in a more concise and elegant manner than with full function definitions. We'll present additional method and initializer concepts in Chapters 8–10.

Index

J

K