Kevin Hoffman

In **Full Color**

Figures and code appear as they do in Xcode

Sams **Teach Yourself**

Mac OS® X Lion™ App Development

in **24 Hours**

SAMS

Kevin Hoffman

Sams Teach Yourself Mac OS® X Lion™ App Development in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

ISBN-13: 978-0-672-33581-5
ISBN-10: 0-672-33581-6

The Library of Congress Cataloging-in-Publication data is on file.

Printed in the United States of America

First Printing March 2012

Editor-in-Chief
Greg Wiegand

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Barbara Hacha

Indexer
Larry Sweazy

Proofreader
Christal White, Language Logistics, LLC

Technical Editor
Mike Givens

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Compositor
Nonie Ratcliff

Contents at a Glance

Table of Contents

About the Author

Kevin Hoffman has been programming since he was 10 years old, when he got his start writing BASIC programs on a Commodore VIC-20. Since then, he has been obsessed with building software and doing so with clean, elegant, simple code in as many languages as he can learn. He has presented twice at Apple's Worldwide Developer Conference, guest lectured at Columbia University on iPhone programming, and built several iOS Apps available in the App Store. He is currently Vice President, Global Information Technology for Barclays Capital, where he is involved in all aspects of the software life cycle—from testing standards to coding standards and from architecture to implementation of powerful, high-volume, low-latency systems.

Dedication

I would like to dedicate this book to Angelica and Isabella, two of the most amazing women I have ever met who put up with me as I complain that my code samples won't work and tolerate me pacing back and forth trying to figure out how to finish a chapter. Most importantly, they encourage me to follow my dreams and do the things that make me happy.

Acknowledgments

This book would never have happened were it not for the efforts of Apple Developer Evangelists like Matt Drance and Mike Jurewitz. One day in New York City, I went into the basement conference rooms of a building a prejudiced man, convinced that no development platform was better than C# and WPF. I left NYC that day enlightened, my horizons broadened and realizing that there is a beauty that lies between the shades of gray among the many development platforms and environments.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can e-mail or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone number or e-mail address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail:	consumer@samspublishing.com
Mail:	Greg Wiegand Editor-in-Chief Sams Publishing 800 East 96th Street Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, and errata that might be available for this book.

This page intentionally left blank

Introduction

Building applications for Mac OS X has never been easier and—more important—has never before enabled such a wide distribution audience or such potential to make money. Because of the new App Store, the time has never been better to be a Mac OS X developer.

This book was written for those with some programming background who want to get into building Mac OS X Lion applications. Each of the 24 hours in this book is designed to quickly introduce you to a new topic that builds on the information you've learned in the previous hours. By the time you finish, you should have a firm grasp on the most important topics for Mac OS X Lion application development, and you should know enough to supplement that learning with additional references, such as Apple's documentation and more in-depth books.

Using this book, you will learn the basics of building Mac OS X Lion applications. To do this, you'll need a computer that has the latest version of Mac OS X Lion installed, and you will need access to the Internet to download and install the developer tools (introduced in Hour 2, "Introduction to the Developer Tools") and access other online resources, such as code downloads and documentation.

Audience and Organization

This book is targeted at those who have some basic familiarity with programming concepts,—but you don't have to be a professional programmer nor do you have to have any prior exposure to Mac OS X or Cocoa development.

This book gradually introduces you to the Objective-C programming language and provides you with the object-oriented programming fundamentals that you will need to proceed through all 24 hours.

The following is a list of the hours in this book with a short description of each.

Part I—Mac OS X Lion Programming Basics

Part 1 of the book provides you with the core knowledge you'll need, the foundation on which all the other hours are based. In this part you'll get an introduction to Objective-C and the development tools used to build Mac OS X applications.

1. Introduction to Mac OS X Lion

This hour provides an introduction to the new features in the Mac OS X Lion operating system, many of which you will be manipulating with code throughout this book. Some are subtle UX changes that make Mac OS X Lion the best version of Mac OS X yet.

2. Introduction to the Developer Tools

This hour introduces you to Xcode, the main development tool for Mac OS X developers. This tool is not only used for writing and compiling code, but now includes the main design tool, Interface Builder, directly within the Xcode tool.

3. Introducing Objective-C

This hour provides you with a quick and easy introduction to the Objective-C programming language, its history, philosophy, and how it differs from other C-like languages. Don't let the C lineage scare you; Objective-C is a very easy language to use and learn.

4. Object-Oriented Programming with Objective-C

This hour introduces you to some basic object-oriented programming (OOP) concepts such as classes, contracts, and polymorphism and uses Objective-C code to illustrate those concepts. After you start adding OOP to Objective-C, things really start to get fun.

5. Understanding Memory Management

This hour provides you with an overview of the various memory management techniques and technologies available to Objective-C programmers, including a discussion of manual reference counting and the newer, easier way of managing memory: Automatic Reference Counting (ARC).

Part II—Cocoa Application Programming Basics

Now that you've completed 5 hours of basics and introductory material, it's time to start building actual Cocoa applications. In this part of the book, you'll learn about Cocoa and its scope and history, how to use controls and lay out your controls within windows, use data binding, display interactive collections and lists, and support new multitouch devices and gestures.

6. Introducing Cocoa

In this hour, you will learn the basics of building Cocoa applications and get an overview of exactly what Cocoa is and how to use it.

7. Using Controls and Layout with Cocoa

This hour provides an overview of the controls available to Cocoa application developers as well as the various layout mechanisms available for organizing and displaying those controls.

8. Creating Interactive Applications

This hour builds on the previous hours and adds some interactivity to your applications by showing you how to invoke code in response to user actions and how to programmatically manipulate UI elements.

9. Creating Data-Bound Interfaces

This hour shows you how to declaratively bind properties of UI elements to underlying data, enabling more powerful and robust user interfaces that require less code to create and maintain.

10. Working with Tables and Collections

In this hour, you learn how to extend your data binding knowledge to working with lists of data that appear as tables or collections.

11. Working with Multitouch and Gestures

In this hour, you learn how Mac OS X (and especially Lion) provide developers with powerful, easy-to-use mechanisms for recognizing and responding to multitouch gestures on any number of devices, including trackpads and magic mice.

Part III—Working with Data

This section of the book deals with something that virtually every application needs to have in some form: data. It covers user defaults, core data, documents, and even Apple's new ubiquitous, synchronized data storage system called iCloud.

12. Working with User Defaults

This hour shows you how (and when) you can read and write data that is stored alongside your application as user-specific application preferences.

13. Working with Core Data

This hour shows you the power of Core Data, a framework that enables your application to store, retrieve, and query relational data in XML, binary, and SQLite.

14. Working with Documents, Versions, and Autosave

This hour extends your knowledge of data binding, collections, and Core Data to show you how to build document-oriented applications and utilize some new Lion features such as versioning and autosave.

15. Working with Apple's iCloud

This hour shows you how to extend your knowledge of document-oriented applications to illustrate how your application can store, query, and maintain documents in the cloud using iCloud.

Part IV—Building Advanced Mac Applications

In this section, you start exploring some of the more advanced features of Mac OS X Lion applications as well as some user interface elements and patterns that differentiate sample applications from real-world, commercial applications.

16. Using Alert Panels and Sheets

This hour shows you how to alert your users to important information or prompt them to supply information using Alert Panels and Sheets.

17. Working with Images

The use of images can enrich virtually any application when done tastefully. This hour shows you how to read, write, display, and manipulate images within your Mac OS X Lion application. You'll be surprised at how much you can do with images in just a few lines of code!

18. Using Popovers

This hour shows you how to display secondary windows that are anchored contextually to other windows or controls. The iPad made popover windows a familiar paradigm to millions of people, and now you can use popovers in your Mac OS X Lion application.

19. Building Animated User Interfaces

This hour shows you how to create and use animation sequences to give your applications the additional life, energy, and reactivity that people have come to expect from Mac OS X and iOS applications over the years.

20. Consuming Web Services

The Internet provides a wealth of data and functionality to enhance or support your application. This hour shows you how to take advantage of web services and how your Mac OS X application can connect to them to send and receive data.

21. Building Full-Screen Applications

This hour shows you how to take advantage of the new full-screen functionality available to all Mac OS X Lion applications.

22. Supporting Drag-and-Drop Behavior

This hour shows you how to add drag-and-drop support to your application, which can dramatically increase user satisfaction and overall user experience.

23. Building Apps for the Mac App Store

This hour introduces you to the new Mac App Store, how to create a developer account, and the process through which you can submit your application to the App Store.

24. Supporting In-App Purchases

Building on the information contained in the previous hour, this hour shows you how you can give your users the ability to buy extended functionality and features directly within your application using the StoreKit framework.

Conventions Used in This Book

The following styles are found throughout the book to help the reader with important points of interest.

Watch Out!

This is the “Watch Out” style. These boxes present important information about the subject that the reader may find helpful to avoid potential problems.

Did You Know?

This is the “Did You Know” style. These boxes provide additional information about a subject that may be of interest to the reader.

By the Way

This is a “By The Way” style. These boxes usually refer the reader to an off-topic subject of interest.

Resource Files

All the code used in this book is available for download online. In addition to being easier than attaching a CD-ROM to the book, this gives us the ability to update the code samples as necessary.

You can download the code samples for this book from the SAMS/Pearson website. Additional information on the code downloads for this book may be available from the author’s website at http://www.kotancode.com.

Keep an eye on informit.com/title/9780672335815 for updates regarding this book.

HOUR 3

Introducing Objective-C

What you'll learn in this hour:

- History of Objective-C
- Doing Some Basic Math
- Using Strings
- Understanding Pointers and Equality
- Passing Messages
- Controlling Program Flow with Loops and Conditionals

It used to be that developers who wrote code in Objective-C were a small, elite group. These programmers wrote code for the Mac, building applications for a desktop OS that had very small market share and even smaller profit potential.

Today, thanks to the increasing popularity of the iPhone, iPod Touch, and the iPad, Objective-C's popularity is on the rise. Everyone from hobbyists to scientists to commercial developers and consultants are building applications for iOS using Objective-C.

During this hour you will be introduced to the Objective-C programming language, learn the basics of using variables, doing arithmetic, and building algorithms with larger blocks of code involving conditionals, looping, and the use of objects such as strings.

Overview and History of Objective-C

At its core, Objective-C is an ANSI standard version of the C programming language. Wrapped around this ANSI C core is a Smalltalk-inspired set of extensions that give the language its object-oriented capabilities, as well as several other enhancements that you don't get from the regular version of C.

Brad Cox and Tom Love created the Objective-C programming language in the early 1980s in an effort to get people to write cleaner, more modular, and clearly separated code. Contrary to popular belief, Objective-C wasn't invented by, nor is it exclusively owned by, Apple. It's actually an open standard; in the past, implementations of the Objective-C compiler existed that even ran on Windows.

If you have had any experience with C, learning Objective-C should be a breeze. Most developers find that learning the Objective-C syntax takes very little time at all, and the rest of the learning curve is devoted to learning about all the tools and controls available in Cocoa for Mac OS X.

By the Way

Objective-C isn't just like C, it *is* C. If you have had any experience with C, C#, or Java, much of Objective-C's syntax should be easy to pick up. Don't let this discourage you if you are new to C-inspired languages, because Objective-C is easy to pick up for new and veteran developers alike.

Creating a New Cocoa Application

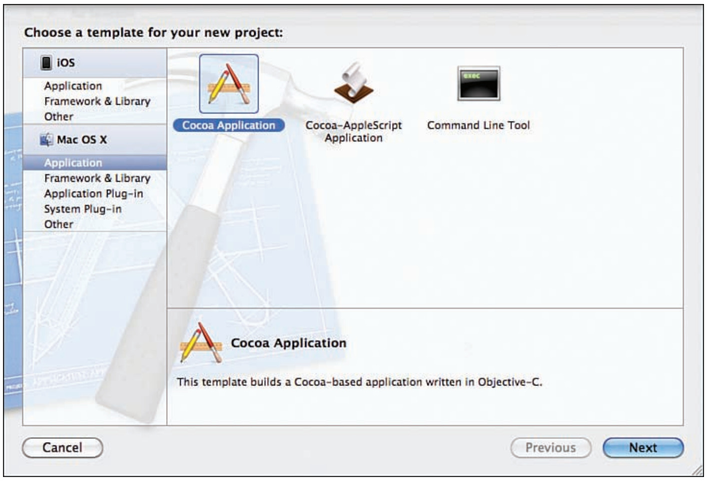
The first few steps of this section should seem familiar to you because in the previous hour we did some exercises to become familiar with the Xcode IDE and the various functions it performs. Throughout this section we'll be writing some code, predicting the output, and running the code to verify the results.

To get started with a new Cocoa application, follow these steps:

1. First, open up Xcode and when prompted create a new project. You should see a screen similar to the one shown in Figure 3.1.
2. Choose Cocoa Application as the template for the project and click Next. On the next screen (shown in Figure 3.2) you need to supply some information for your product. Use the information in Table 3.1 to fill out this form. Leave all the other values as defaults.

TABLE 3.1 Form Values to Create a New Cocoa Application

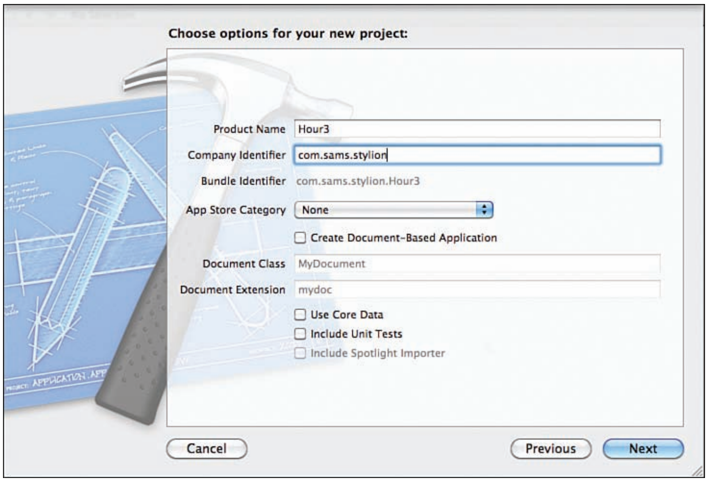
Form Field	Value
Product Name	Hour3
Company Identifier	com.sams.stylion
App Store Category	None



Cocoa Application

This template builds a Cocoa-based application written in Objective-C.

FIGURE 3.1 Creating a new Cocoa Application.



Product Name	Hour3
Company Identifier	com.sams.stylion
Bundle Identifier	com.sams.stylion.Hour3
App Store Category	None
	☐ Create Document-Based Application
Document Class	MyDocument
Document Extension	mydoc
	☐ Use Core Data
	☐ Include Unit Tests
	☐ Include Spotlight Importer

FIGURE 3.2 Providing product information for a new Cocoa Application.

3. When Xcode finishes building your new Cocoa Application, run it to make sure that you see a standard, empty window.

This is where the review starts, and we move on to covering new material and writing some actual code. If you don't feel comfortable using Xcode to create new applications, you might want to go back to review the work we did in Hour 2, "Introduction to the Developer Tools," before continuing on.

By the Way

> If you're curious, take a look at some of the files created automatically for you in the starter project. See what happens when you click to select each of the different files in the project navigator on the left.

Exploring the Language with Some Basic Math

Regardless of what kind of application you plan to build, you will invariably find yourself working with numbers or strings (text). Whether you want to allow your users to create a budget, report their high score in a game, or see the average of their test scores, you'll need to use numbers for that.

You have probably performed some basic math using some programming language in the past and, as mentioned earlier, because Objective-C is a superset of C, anything you can do in C you can do in Objective-C, including math.

Follow these steps to create some code that illustrates the basic use of numbers and math in Objective-C:

1. With Xcode open to the project you created in the preceding task, click the `Hour3AppDelegate.m` file. Locate the `applicationDidFinishLaunching:` method and add the following lines of code between the two curly braces:

   ```
   int a=5;
   int b=10;
   int c = a*b;
   NSLog(@"a*b = %d", c);
   ```

2. In the top-right corner of the Xcode IDE, click the button that shows a bottom tray icon. If you hover over this button, it will indicate that it hides or displays the debug area.

3. With the debug area now visible, run your application. In the debug area, below the information about the version of the debugger being used, you should see some text that looks like this:

   ```
   2011-05-29 12:16:44.916 Hour3[640:903] a*b = 50
   ```

`NSLog` is a function that you can call that will log text to a standard output device—in our case, the debug area of the application.

You may have noticed the little yellow triangles that appeared in the margin to the left of your code as you typed. Even before you build and run your application, as you type, the compiler is checking your code for errors or possible problems (warnings). The first warning you saw was the compiler informing you that you had declared the variable a but hadn't yet used it. As soon as you used it, that warning disappeared. This kind of instant, interactive feedback from Xcode can come in extremely handy and save you a lot of troubleshooting time in the future.

By the Way

You may have noticed that there are a couple of panels you can open in the editor. Click the icon for the third (utilities) panel. You'll see a panel with a file inspector and Quick Help. Keep Quick Help open as you enter code throughout the rest of this hour and watch how it automatically figures out what information to display based on what you're typing.

4. Now add the following lines of code where you left off:

```
double d = 123.00;
double e = 235.00;
double f = d * e;
NSLog(@"d*e = %f", f);
NSLog(@"d/e = %f", d / e);
NSLog(@"d/0 = %f", d / 0);
```

By the Way

Note that as you type the last line of code, Xcode instantly throws a yellow caution triangle into the left margin. If you click that triangle, you'll see that the warning message says that division by zero is undefined.

5. Now try to run the application, even with that warning still in place. Your output in the debug area should look like this:

```
2011-05-29 12:29:31.407 Hour3[745:903] a*b = 50
2011-05-29 12:29:31.410 Hour3[745:903] d*e = 28905.000000
2011-05-29 12:29:31.411 Hour3[745:903] d/e = 0.523404
2011-05-29 12:29:31.412 Hour3[745:903] d/0 = inf
```

Notice here that dividing by zero didn't crash our application. In many programming languages, including modern, managed runtime environments such as Java and C#, any attempt to divide by zero will crash the running application. Here, Objective-C kept on going and even managed to give the mathematically correct answer here of *infinity* rather than a crash.

When you are done with this hour, feel free to go back to this code and play around with more basic math. Remember that the data types we've used thus far are basic C types. As Cocoa programmers using Objective-C, you have the primitive numeric data types shown in Table 3.2 (as well as several others) available to you.

TABLE 3.2 Some Basic Objective-C Numeric Data Types

Type	Size	Description
int	32 or 64 bits	Basic integer. The size varies whether you're building a 32 or 64-bit application.
float	32 bits	A data type used for storing floating-point values.
double	64 bits	A data type used for storing large floating-point values.
long	64 bits	A data type used for storing large integers.
uint	2x(int)	A data type used for storing integers without a sign bit, allowing for the storage of numbers twice as big as in integer.

More numeric data types are available, but these are the ones that get used most often.

6. Add the following lines of code to the sample you've been working with:

```
float floaty = d*e;

NSLog(@"size of float: %lu", sizeof(floaty));
NSLog(@"size of double: %lu", sizeof(f));
NSLog(@"size of int: %lu", sizeof(a));
NSLog(@"size of long: %lu", sizeof(long));
NSLog(@"size of uint: %lu", sizeof(uint));
```

7. Now run the application to see the various sizes of the data types. Note that the `sizeof` function will work on a variable (such as `floaty` or `f`) or a data type (such as `long` or `int`).

One possible problem that you may already be thinking about is that different users have different computers, and as such, an `int` on the developer's computer might not always be the same as an `int` on the end-user's computer. There is a way to deal with these issues (and many more not-so-obvious problems with numbers and math), and you'll see one solution later in this hour when we get to a discussion of pointers.

Using Strings

Next to numbers, the other type of data that you will spend a large amount of time with is *strings*. Strings may just be pieces of text, but they are used extensively throughout every kind of application. Knowing how to use them efficiently and properly is essential to developing applications for the Mac or any other platform.

In the preceding section, you were actually using strings. Every time you wrote code that used the `NSLog` function, you were using strings. The `@` operator in Objective-C is a convenient shortcut for the creation of a string. In addition to being a convenient shortcut, it also provides a way of distinguishing between a classic C-style string and a true Objective-C string object. Strings in Objective-C are objects, not simple values. The difference between the two will become clear in the next section on pointers and equality.

There are two different kinds of strings in Objective-C: `NSString` and `NSMutableString`. As their names imply, the former represents an instance of an immutable string, whereas the latter represents an in-memory string that can be manipulated. Many languages, including C# and Java, enforce that all strings are immutable and that anytime you "modify" a string, you are actually creating a new copy containing the modification.

By the Way

The concept of immutability is a tough one to grasp. For many languages, immutable strings are better for performance and even prevent catastrophic failures. If you can avoid using mutable strings in Objective-C, you should. Typically, you use `NSMutableString` when you need a single pointer to a string that will change frequently throughout your application's lifetime, and you don't want to create new strings each time it changes.

Creating and Manipulating Strings

Now that you've had a brief introduction to strings, let's write some code to experiment with them. Follow these steps to create the sample code:

1. Add the following lines of code to the sample you've been working on:

```
NSString *favoriteColor = @"green";
NSLog(@"%@", favoriteColor);

NSString *friendlyOutput =
   [NSString stringWithFormat:
      @"Kevin loves the color %@", favoriteColor];
NSLog(@"%@", friendlyOutput);

NSString *subString = [friendlyOutput
  substringWithRange:NSMakeRange(6, 15)];
NSLog(@"substring: %@", subString);
```

There's a lot going on here that you may not be familiar with, so let's walk through it line by line.

The first line creates a new variable called `favoriteColor`, which is a *pointer* to an object of type `NSString` (again, don't worry about pointers just yet, we're building up to them slowly but surely).

The second line uses the `%@` string format character to log the string object. Just like standard C format specifiers such as `%d` and `%f` are used to represent numbers, the `%@` format string in Objective-C is used to represent an *object*. All objects in Objective-C know how to describe themselves, and whenever an object is used as a parameter to the `%@` format specifier, that object is asked to describe itself. Conveniently, the way a string describes itself is by returning its contents.

The third line creates a new string using a format string. Don't worry if the square bracket syntax looks odd; we'll get to an explanation of those in the section on message passing. Until then, try to mentally replace the square brackets with the idea that a message is being sent to an object.

Next we create a new string by extracting a substring using a range of indices. In the preceding code, we're extracting by starting at string index 6 (strings are indexed starting at 0, just like C arrays) and pulling the following 15 characters.

2. Run the sample to see what kind of output you get. It should now look something like this:

```
2011-05-29 13:10:03.351 Hour3[1076:903] a*b = 50
2011-05-29 13:10:03.354 Hour3[1076:903] d*e = 28905.000000
2011-05-29 13:10:03.356 Hour3[1076:903] d/e = 0.523404
2011-05-29 13:10:03.358 Hour3[1076:903] d/0 = inf
2011-05-29 13:10:03.359 Hour3[1076:903] size of float: 4
2011-05-29 13:10:03.360 Hour3[1076:903] size of double: 8
2011-05-29 13:10:03.360 Hour3[1076:903] size of int: 4
2011-05-29 13:10:03.361 Hour3[1076:903] size of long: 8
2011-05-29 13:10:03.363 Hour3[1076:903] size of uint: 4
2011-05-29 13:10:03.366 Hour3[1076:903] green
2011-05-29 13:10:03.367 Hour3[1076:903] Kevin loves the color green
2011-05-29 13:10:03.367 Hour3[1076:903] substring: loves the color
```

All applications use strings. They need them to display information to users and to gather information from users. As a Mac developer, you will find that you spend a great deal of time creating, manipulating, and comparing strings.

This leads to the next section on pointers and equality, which is *very* important to understand, especially when comparing things like strings.

Understanding Pointers and Equality

When most people think of a *pointer*, they think of an arrow, a sign, or even a human finger pointing in a certain direction. This is exactly what a pointer is in C or Objective-C: an indicator pointing at another location.

When your code executes, that code resides in memory. Available to your code are two types of memory: the *stack* and the *heap*. The stack, as its name implies, is a contiguous block of memory. Values can be *pushed* into the stack and they can be *popped* off of the stack. This mechanism is how functions are called: The parameters to a function are pushed onto the stack and then execution resumes within the function where the parameters are popped off the stack to make them available to the code within the function.

This works great when function parameters are small pieces of datalike integers or floats or even smaller values. What if you have a very large object (we'll cover creating classes and instantiating them in the next hour) that takes up hundreds—or even thousands—of bytes? Now the mere presence of this object on the stack is creating a headache for whatever runtime is executing your code. If you had to shovel things from one bucket to another bucket before you could do your work, would you rather shovel small fish tank pebbles or enormous boulders?

Enter pointers. Rather than placing all of the data belonging to the large object (the boulder) on the stack, we can instead write down the location of the boulder onto a small pebble and place that small pebble on the stack instead. This way, we can now quickly and easily pass all of our parameters to a function using the stack. When the function needs to access the large object/boulder, it can read the location from the pebble we put on the stack and go get the object on its own.

Obviously the Objective-C runtime doesn't write down locations on pebbles. It does, however, store memory addresses in integers and place *those* on the stack.

Far more so than in other languages like C# or Java, you will be dealing with pointers on a daily basis in Objective-C. Just remembering that pointers are merely numbers on the stack that point to a different location on the heap will serve you well as you go through the rest of this book.

To see an example of pointers in action, follow these steps:

1. Type the following code in after the last line of code you wrote for the previous section:

```
int targetNum = 42;
int bigTargetNum = 75;
int *targetNum_ptr = &targetNum;
int *targetNum_ptr2 = &targetNum;
NSLog(@"targetNum = %d, ptr points to = %d", targetNum,
             *targetNum_ptr);
                 NSLog(@"are two pointers equal? %d", (targetNum_ptr ==
             targetNum_ptr2));
                 targetNum_ptr2 = &bigTargetNum;
                 NSLog(@"are two pointers equal 2nd time? %d", (targetNum_ptr ==
             targetNum_ptr2));
                 *targetNum_ptr2 = 42; // what did this really do?
                 NSLog(@"are two pointers equal 3rd time? %d", (targetNum_ptr ==
             targetNum_ptr2));

                 NSString *stringA = @"Hello World";
                 NSString *stringB = [NSString stringWithFormat:@"%@ %@", @"Hello",
             @"World"];
                 NSLog(@"Is %@ == %@ : %d", stringA, stringB, (stringA == stringB));
                 NSLog(@"What about with isEqualToString? %d",

                    [stringA isEqualToString:stringB]);
```

2. Before running the application, try to predict what the output might be. On the first two lines, we create two standard integers. These are standard integers and we are storing their actual values.

On the next two lines we create two pointers. First, take note of the `*` operator. In this case, we are not multiplying. The presence of an asterisk before a variable name in a declaration statement indicates that we are declaring a *pointer*. This informs Objective-C that the data being stored in this variable is a *memory location* (pointer to real data), and *not the actual data*.

On these same lines of code, we use the *address-of* operator (&) to obtain the memory address of the number stored in the location called `targetNum`. At this point, both `targetNum_ptr` and `targetNum_ptr2` *point to the address of* `targetNum`.

On the next line, we use the asterisk again, but this time as a *dereferencing operator*. This means that when Objective-C expects a value and you give it a pointer variable preceded by an asterisk, Objective-C will go fetch the *value* from the memory location currently stored within that pointer.

The next set of statements attempts to get a definitive answer to the question: *Are two pointers equal if they both point to the same memory location?* Running the application shows you that this is indeed true. For example, suppose the memory address holding the value for `targetNum` was 100080. If we set `targetNum_ptr` and `targetNum_ptr2` to `&targetNum`, we set *both* of those variables to 100080, which makes them equivalent.

The next question is a little more tricky: *If two pointers point to two different locations, but each location contains the same value, are those pointers equal?*

This is the key to understanding pointers and equality. When you compare pointers, you are essentially asking if the address of the underlying data is the same. If you want to compare whether the data itself is the same, you need to either dereference the pointers and compare the underlying data or, better yet, use classes that know how to compare themselves.

This leads us to the `NSString` class. The first variable, `stringA`, is assigned to a string literal “Hello World”. Remember that the `@` operator is actually a shortcut for creating an instance of an immutable string. If I had assigned `stringB` to `@"Hello World"`, both `stringA` and `stringB` would be considered equivalent via the == operator because both would point to the single memory location on the heap where the immutable string “Hello World” resides.

Because we used two different methods to create “Hello World”, one constant and one via concatenation, neither `stringA` nor `stringB` points to the same memory location. However, because `NSString` knows how to do string comparison via the `isEqualToString` method (there is an abundance of other related string-comparison tools for more involved scenarios), you can compare the two strings and verify that the *content* is equivalent, even though both copies of “Hello World” are sitting in two difference places on the heap.

3. When you run the application you’ve been working on this hour, the output related to pointers and string comparison should look like the following:

```
2011-05-29 13:31:04.788 Hour3[1325:903] targetNum = 42, ptr points to = 42
2011-05-29 13:31:04.789 Hour3[1325:903] are two pointers equal? 1
2011-05-29 13:31:04.790 Hour3[1325:903] are two pointers equal 2nd time? 0
2011-05-29 13:31:04.792 Hour3[1325:903] are two pointers equal 3rd time? 0
2011-05-29 13:31:04.793 Hour3[1325:903] Is Hello World == Hello World : 0
2011-05-29 13:31:04.794 Hour3[1325:903] What about with isEqualToString? 1
```

Finally, before moving on to the next section, it is worth noting that Objective-C strings are in fact Unicode strings. This means that these strings are not limited to storing text in English or similar 7- and 8-bit languages. Unicode strings can be used to store text in Chinese, Japanese, Hindi, Arabic, and any other language that uses text characters whose ordinal value has a value higher than 255 (often referred to as long characters or multibyte characters).

By the Way

Not only do non-US versions of Mac OS X allow people to type Unicode characters directly with special keyboards, but there are also phonetic Input Method Editors (IMEs) that allow people to type phonetically and autoconvert text into Unicode. For example, someone might type ‘ohayou gozaimasu’ into an IME and get the Japanese Unicode: おはひょうございます。Knowing that Objective-C strings are Unicode strings can come in handy in the future, especially for languages that will support multiple cultures.

Passing Messages

If you've never seen Objective-C before, the square brackets ([and]) might seem a little unusual. This is what is known as *message passing syntax*. Within the two square brackets are the *target* of a message and the *parameters* of that message.

The capability to send messages to objects is what lifts Objective-C above its roots in ANSI C and provides the robust, object-oriented functionality that Mac and iOS developers love today.

Let's take an example from some code that we've already written:

```
    NSString *stringB = [NSString stringWithFormat:@"%@ %@", @"Hello", @"World"];
```

By the Way

If you've had Quick Help active throughout the hour, notice what happens when you click the first letter of the method name; documentation for that method appears instantly. Now see what happens when you hit the escape key with the cursor still in that same spot—a list of all the possible messages that can be sent to an object or class! This list isn't always as well-informed as the developer, so take its contents with a grain of salt.

In this line of code we are sending the `NSString` class the `stringWithFormat:` message. Note that the colon is considered part of the message name. In fact, all subsequent parameters are considered part of the name of the method being called. For example, if I had an ice cream object and I wanted to top it using a specific method, it might look something like this:

```
[iceCream topWithSauce:SauceHotFudge withCherry:YES withWhippedCream:NO];
```

In this line of code, what's the name of the method we are invoking? It isn't just `topWithSauce:` because that doesn't convey what we're actually doing. The method name is actually called `topWithSauce:withCherry:withWhippedCream:`

It might take some getting used to for programmers used to calling methods with a comma-delimited list of arguments as shown in the following line of C# code:

```
iceCream.Top(Toppings.HotFudge, true, false);
```

However, this syntax makes for highly readable, self-documenting code, and in many cases it removes all ambiguity about the name and purpose of a method's parameters without needing to consult an external document.

As an exercise, go back through the code you've written so far this hour and for every use of square brackets, determine the target being sent the message and the name of the method being called.

By the Way

When you say the names of Objective-C methods out loud, you don't pronounce the colons. Instead, insert a little pause between the parameter names as you read aloud.

Controlling Program Flow

So far everything you've written has been linear. Each line of code you wrote executed immediately after the preceding line and it executed only once. What if you want to execute the same block of code 50 times? What if you want to execute a block of code over and over until you reach the end of an input file? What if you want to decide whether to execute a block of code based on some condition?

For these and many more scenarios, you have to turn to some Objective-C keywords that control your program's execution flow. In short, you will need to use looping or logic keywords.

The most common way to control program flow is through conditional branching. This can be done with the `if` statement and the `else` statement or the highly favored `switch` statement.

An `if` statement works by evaluating some Boolean expression (an expression that yields true or false or, in C terms, either zero or nonzero for false and true, respectively). If the Boolean expression evaluates to true, the body of the `if` statement is executed. If the expression evaluates to false, the body of the `if` statement is *not* executed and, optionally, an `if else` or `else` block is executed.

Follow these steps to experiment with branching and looping:

1. Enter the following code on a blank line below the code you've been writing so far:

```
if ([favoriteColor isEqualToString:@"blue"])
{
    NSLog(@"Your favorite color is BLUE!");
}
else if ([favoriteColor isEqualToString:@"purple"])
{
    NSLog(@"You're into purple.");
}
```

```
else
{
    NSLog(@"You don't like blue or purple!");
}
```

Remembering back to earlier in the hour, we set the value of `favoriteColor` to "green". This means that when you execute this code, it will not execute the body of the `if` statement or the body of the `else if` statement. It will execute the body of the `else` statement. Play around with the value of `favoriteColor` to get the different statements to execute. Note that every single Boolean expression evaluated by an `if` or `else if` statement *must* be wrapped in parentheses.

Next, let's see what a simple `for` loop looks like. As you may have guessed, an Objective-C `for` loop works the same as it does in ANSI C.

By the Way

Later in the book you'll see a special type of `for` loop called a *fast iterator*. This is an elegant, clean syntax that lets you iterate through the items in a collection with extremely good performance.

2. Enter the following code where you left off:

```
for (int x=0; x<10; x++)
{
    switch (x) {
        case 0:
            NSLog(@"First time through!");
            break;
        case 1:
        case 2:
        case 3:
        case 4:
            NSLog(@"Not halfway there yet...");
            break;
        default:
            NSLog(@"Working on the %dth iteration.", x);
            break;
    }
}
```

When executed, this produces output similar to the following:

```
2011-05-29 17:14:07.857 Hour3[1594:903] First time through!
2011-05-29 17:14:07.858 Hour3[1594:903] Not halfway there yet...
2011-05-29 17:14:07.859 Hour3[1594:903] Not halfway there yet...
2011-05-29 17:14:07.860 Hour3[1594:903] Not halfway there yet...
2011-05-29 17:14:07.861 Hour3[1594:903] Not halfway there yet...
2011-05-29 17:14:07.861 Hour3[1594:903] Working on the 5th iteration.
```

```
2011-05-29 17:14:07.862 Hour3[1594:903] Working on the 6th iteration.
2011-05-29 17:14:07.863 Hour3[1594:903] Working on the 7th iteration.
2011-05-29 17:14:07.864 Hour3[1594:903] Working on the 8th iteration.
2011-05-29 17:14:07.865 Hour3[1594:903] Working on the 9th iteration.
```

You can see that the `for` loop we just executed gave us 10 iterations, with indices ranging from 0 to 9. Other kinds of loops are available to us as well, including the `do` loop and the `while` loop. These loops both execute their bodies until the associated Boolean expression evaluates to false. The only difference is that a `do` loop will *always* execute at least once because it evaluates the Boolean condition *after* the body executes. A `while` loop evaluates the Boolean condition *before* the body executes, and if the expression evaluates to false, the body will never execute. Which loop you choose is entirely up to you and should be based on how readable you want the code to be and, of course, personal preference.

Summary

In this hour, you got your feet wet with Objective-C. You experimented a little with basic mathematics and string manipulation—the building blocks of any application. You took a look at pointers, how they work, and how they affect things like equality and comparisons. You took a look at message-passing syntax and finally spent a little time creating loops and using branching and conditional logic.

Hopefully, at this point, you are excited about Objective-C and what you might be able to do with it in the next hour, where we talk about object-oriented programming and creating and consuming your own classes.

Q&A

Q. ***How is Objective-C related to the ANSI C programming language?***

A. Objective-C isn't just similar to C; it is a superset of the C language. This means that any standard ANSI C code is compatible with an Objective-C application.

Q. ***What function do you call to write messages to the system log and console output?***

A. `NSLog()`. This function will become very familiar to you as you write more and more code.

Q. ***How does Objective-C message passing differ from calling methods in other programming languages?***

A. In Objective-C, the method parameters are actually part of the message name. In addition, message passing in Objective-C is dynamic, meaning you can compile code that sends messages to objects that may or may not respond to those messages at runtime.

Index

How can we make this index more useful? Email us at indexes@samspublishing.com