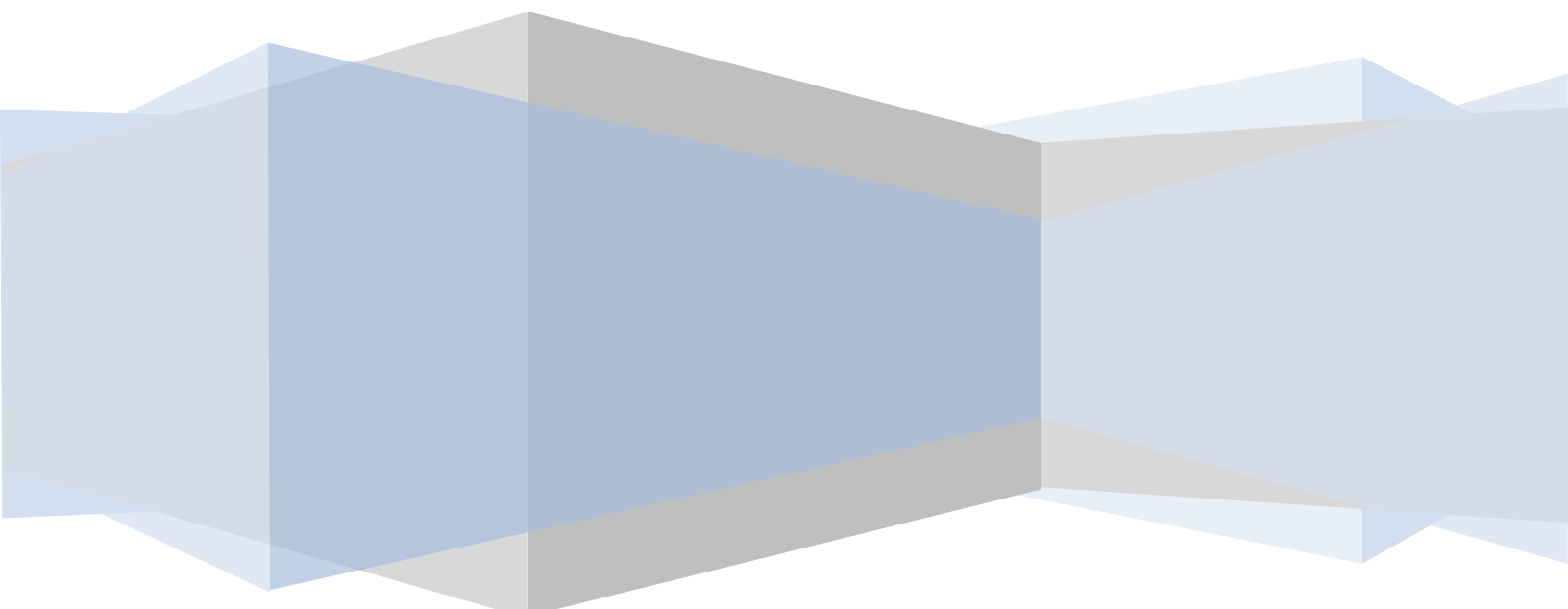


Objective-C



Essentials

Objective-C 2.0 Essentials



Objective-C 2.0 Essentials

© 2012 Payload Media. This eBook is provided for personal use only. Unauthorized use, reproduction and/or distribution strictly prohibited. All rights reserved.

The content of this book is provided for informational purposes only. Neither the publisher nor the author offers any warranties or representation, express or implied, with regard to the accuracy of information contained in this book, nor do they accept any liability for any loss or damage arising from any errors or omissions.

Find more eBooks online at <http://www.eBookFrenzy.com>.

Rev. 2.0

Table of Contents

Chapter 1. About Objective-C Essentials	13
1.1 Why are you reading this?	13
1.2 Supported Platforms	13
Chapter 2. The History of Objective-C	15
2.1 The C Programming Language	15
2.2 The Smalltalk programming Language.....	15
2.3 C meets Smalltalk	15
2.4 Objective-C and Apple.....	16
Chapter 3. Installing Xcode and Compiling Objective-C on Mac OS X.....	17
3.1 Installing Xcode on Mac OS X.....	17
3.2 Starting Xcode	17
3.3 Writing an Objective-C Application with Xcode.....	19
3.4 Compiling Objective-C from the Command Line	20
3.5 Summary	21
Chapter 4. Objective-C 2.0 Data Types	23
4.1 int Data Type	24
4.2 char Data Type	25
4.2.1 Special Characters/Escape Sequences	25
4.3 float Data Type	26
4.4 double Data Type	26
4.5 id Data Type.....	26
4.6 BOOL Data Type	26
4.7 Objective-C Data Type Qualifiers	27
4.7.1 long	27
4.7.2 long long	27
4.7.3 short.....	27
4.7.1 signed / unsigned.....	28

4.8 Summary	28
Chapter 5. Working with Variables and Constants in Objective-C	29
5.1 What is an Objective-C Variable.....	29
5.2 What is an Objective-C Constant?.....	30
5.3 Type Casting Objective-C Variables.....	30
Chapter 6. Objective-C Operators and Expressions.....	32
6.1 What is an Expression?	32
6.2 The Basic Assignment Operator	32
6.3 Objective-C Arithmetic Operators.....	33
6.4 Compound Assignment Operators.....	33
6.5 Increment and Decrement Operators.....	34
6.6 Comparison Operators	35
6.7 Boolean Logical Operators	36
6.8 The Ternary Operator.....	37
6.9 Bitwise Operators.....	37
6.9.1 Bitwise AND	38
6.9.2 Bitwise OR.....	39
6.9.3 Bitwise XOR.....	39
6.9.4 Bitwise Left Shift	40
6.9.5 Bitwise Right Shift.....	40
6.10 Compound Bitwise Operators.....	41
Chapter 7. Objective-C 2.0 Operator Precedence	42
7.1 An Example of Objective-C Operator Precedence	42
7.2 Objective-C Operator Precedence and Associativity	42
7.3 Overriding Operator Precedence	44
Chapter 8. Commenting Objective-C Code.....	45
8.1 Why Comment your Code?	45
8.2 Single Line Comments	45

8.3 Multi-line Comments	46
8.4 Summary	47
Chapter 9. Objective-C Flow Control with <i>if</i> and <i>else</i>	48
9.1 Using the <i>if</i> Statement.....	48
9.2 Using <i>if ... else ...</i> Statements	49
9.3 Using <i>if ... else if ...</i> Statements	49
9.4 Summary	50
Chapter 10. The Objective-C <i>switch</i> Statement.....	51
10.1 Why Use a <i>switch</i> Statement?	51
10.2 Using the <i>switch</i> Statement Syntax	52
10.3 A <i>switch</i> Statement Example	53
10.4 Explaining the Example	53
10.5 Combining case Statements.....	54
Chapter 11. Objective-C Looping - The <i>for</i> Statement	55
11.1 Why Use Loops?	55
11.2 Objective-C Loop Variable Scope	57
11.3 Creating an Infinite <i>for</i> Loop	58
11.4 Breaking Out of a <i>for</i> Loop	58
11.5 Nested <i>for</i> Loops	58
11.6 Breaking from Nested Loops.....	59
11.7 Continuing <i>for</i> Loops	59
11.8 Using <i>for</i> Loops with Multiple Variables	60
Chapter 12. Objective-C Looping with <i>do</i> and <i>while</i> Statements.....	62
12.1 The Objective-C <i>while</i> Loop.....	62
12.2 Objective-C <i>do ... while</i> loops.....	62
12.3 Breaking from Loops	63
12.4 The <i>continue</i> Statement.....	64
Chapter 13. An Overview of Objective-C Object Oriented Programming	65

13.1 What is an Object?	65
13.2 What is a Class?	65
13.3 Declaring an Objective-C Class Interface	65
13.4 Adding Instance Variables to a Class	66
13.5 Define Class Methods	67
13.6 Declaring an Objective-C Class Implementation	68
13.7 Declaring, Initializing and Releasing a Class Instance	70
13.8 Calling Methods and Accessing Instance Data	71
13.9 Creating the Program Section	71
13.10 Bringing it all Together	72
13.11 Structuring Object-Oriented Objective-C Code	74
Chapter 14. Writing Objective-C Class Methods	78
14.1 Instance and Class Methods	78
14.2 Creating a New Class Method	78
14.3 The @interface Section	79
14.4 The @implementation Section	79
14.5 The main() Function	80
Chapter 15. Objective-C - Data Encapsulation, Synthesized Accessors and Dot Notation	82
15.1 Data Encapsulation	82
15.2 Synthesized Accessor Methods	82
15.3 Direct Access to Encapsulated Data	84
15.4 Objective-C and Dot Notation	84
15.5 Controlling Access to Instance Variables	85
Chapter 16. Objective-C Inheritance	87
16.1 Inheritance, Classes and Subclasses	87
16.2 An Objective-C Inheritance Example	87
16.3 Extending the Functionality of a Subclass	89
16.4 Overriding Inherited Methods	90

Chapter 17. Pointers and Indirection in Objective-C.....	93
17.1 How Variables are Stored.....	93
17.2 An Overview of Indirection	94
17.3 Indirection and Objects.....	96
17.4 Indirection and Object Copying.....	96
Chapter 18. Objective-C Dynamic Binding and Typing with the id Type.....	98
18.1 Static Typing vs. Dynamic Typing	98
18.2 Dynamic Binding.....	99
18.3 Polymorphism	100
Chapter 19. Objective-C Variable Scope and Storage Class	101
19.1 Variable Scope.....	101
19.2 Block Scope	101
19.3 Function Scope	103
19.4 Global Scope.....	105
19.5 File Scope.....	107
19.6 Variable Storage Class	107
Chapter 20. An Overview of Objective-C Functions	109
20.1 What is a Function?.....	109
20.2 How to Declare an Objective-C Function	109
20.3 The main() Function	110
20.4 Calling an Objective-C Function	110
20.5 Function Prototypes	111
20.6 Function Scope and the static Specifier	114
20.7 Static Variables in Functions	114
Chapter 21. Objective-C Enumerators.....	116
21.1 Why Use Enumerators	116
21.2 Declaring an Enumeration.....	116
21.3 Creating and Using an Enumeration	117

21.4 Enumerators and Variable Names	117
Chapter 22. An Overview of the Objective-C Foundation Framework.....	119
22.1 The Foundation Framework.....	119
22.2 Including the Foundation Headers.....	119
22.3 Finding the Foundation Framework Documentation	120
Chapter 23. Working with String Objects in Objective-C.....	121
23.1 Strings without NSString	121
23.2 Declaring Constant String Objects	122
23.3 Creating Mutable and Immutable String Objects	122
23.4 Getting the Length of a String	123
23.5 Copying a String.....	124
23.6 Searching for a Substring	125
23.7 Replacing Parts of a String.....	126
23.8 String Search and Replace.....	127
23.9 Deleting Sections of a String	127
23.10 Extracting a Subsection of a String.....	127
23.11 Inserting Text into a String	128
23.12 Appending Text to the End of a String	128
23.13 Comparing Strings	129
23.14 Checking for String Prefixes and Suffixes.....	129
23.15 Converting to Upper or Lower Case.....	130
23.16 Converting Strings to Numbers	131
23.17 Converting a String Object to ASCII.....	131
Chapter 24. Understanding Objective-C Number Objects	132
24.1 Creating and Initializing NSNumber Objects.....	132
24.2 Getting the Value of a Number Object	133
24.3 Comparing Number Objects.....	134
24.4 Getting the Number Object Value as a String.....	135

Chapter 25. Working with Objective-C Array Objects	136
25.1 Mutable and Immutable Arrays	136
25.2 Creating an Array Object	136
25.3 Finding out the Number of Elements in an Array	137
25.4 Accessing the Elements of an Array Object	137
25.5 Accessing Array Elements using Fast Enumeration	138
25.6 Adding Elements to an Array Object	138
25.7 Inserting Elements into an Array	138
25.8 Deleting Elements from an Array Object	139
25.9 Sorting Array Objects	140
Chapter 26. Objective-C Dictionary Objects	142
26.1 What are Dictionary Objects?	142
26.2 Creating Dictionary Objects	142
26.3 Initializing and Adding Entries to a Dictionary Object	142
26.4 Getting an Entry Count	143
26.5 Accessing Dictionary Entries	144
26.6 Removing Entries from a Dictionary Object	144
Chapter 27. Working with Directories in Objective-C	145
27.1 The Objective-C NSFileManager, NSFileHandle and NSData Classes	145
27.2 Understanding Pathnames in Objective-C	145
27.3 Obtaining a Reference to the Default NSFileManager Object	146
27.4 Identifying the Current Working Directory	146
27.5 Changing to a Different Directory	146
27.6 Creating a New Directory	147
27.7 Deleting a Directory	147
27.8 Renaming or Moving a File or Directory	148
27.9 Getting a Directory File Listing	148
27.10 Getting the Attributes of a File or Directory	149

Chapter 28. Working with Files in Objective-C	151
28.1 Getting the NSFileManager Reference	151
28.2 Checking if a File Exists	151
28.3 Comparing the Contents of Two Files	151
28.4 Checking if a File is Readable/Writable/Executable/Deletable	152
28.5 Moving/Renaming a File	152
28.6 Copying a File	153
28.7 Removing a File	153
28.8 Creating a Symbolic Link	154
28.9 Reading and Writing Files with NSFileManager	154
28.10 Working with Files using the NSFileHandle Class	155
28.11 Creating an NSFileHandle Object	155
28.12 NSFileHandle File Offsets and Seeking.....	155
28.13 Reading Data from a File	156
28.14 Writing Data to a File	157
28.15 Truncating a File	158
Chapter 29. Constructing and Manipulating Paths with NSPathUtilities	159
29.1 The Anatomy of a Path.....	159
29.2 Finding a Temporary Directory	159
29.3 Getting the Current User's Home Directory	160
29.4 Getting the Home Directory of a Specified User.....	160
29.5 Extracting the Filename from a Path	160
29.6 Extracting the Filename Extension.....	161
29.7 Standardizing a Path.....	161
29.8 Extracting the Components of a Path	162
Chapter 30. Copying Objects in Objective-C.....	163
30.1 Objects and Pointers	163
30.2 Copying an Object in Objective-C using the <NSCopying> Protocol.....	163

30.3 <NSCopying> Protocol and copyWithZone Method Implementation	164
30.4 Performing a Deep Copy	166
Chapter 31. Using Objective-C Preprocessor Directives.....	169
31.1 The #define Statement.....	169
31.2 Creating Macros with the #define Statement.....	170
31.3 Changing the Objective-C Language with #define	170
31.4 undefining a Definition with #undef.....	172
31.5 Conditional Compilation	172
31.6 The #import Directive	173
Index	175

Chapter 1. About Objective-C Essentials

1.1 Why are you reading this?

On the surface this sounds like an odd opening sentence for a programming book. After all, if this were a book about JavaScript or PHP it would be safe to assume that you planned to develop some kind of web site or web application. Similarly, if this were a Visual Basic book it would be a good bet that you had plans to write a Windows application. Indeed, had this question been asked a few years ago, it could have been guessed with a reasonable level of confidence that you wanted to learn Objective-C in order to develop some software to run on Apple's Mac OS X operating system. Now, however, there is a greater likelihood that you plan to develop an application to run on the iPhone or iPad.

The iPhone and the iPad, after all, run a special version of Mac OS X called iOS. Given that Objective-C is the programming language of choice for this operating system it should come as no surprise that before you can develop iOS applications you first need to learn how to program in Objective-C.

The objective of this book is to teach the skills necessary to program in Objective-C using a style that is easy to follow, rich in examples and accessible to those who have never used Objective-C before. Topics covered include the fundamentals of Objective-C such as variables, looping and flow control. Also included are details of object-oriented programming, working with files and memory and the Objective-C Foundation framework.

Those who have developed using other programming languages such as C, C++, C# or Java will find much about Objective-C that is familiar. That said, there are aspects of the language syntax that are unique to Objective-C. Even experienced programmers should therefore expect to spend some time transitioning to this increasingly popular programming language before embarking on a major development project.

Whatever your background and experience, we have worked hard to make this book as useful and helpful as possible as you traverse the Objective-C learning curve.

1.2 Supported Platforms

After all this talk about Mac OS X and iOS, it is important to note that Objective-C is not confined to Apple's operating systems. In fact, Objective-C is available on a wide range of platforms including Linux, NetBSD, OpenBSD, FreeBSD, Solaris and Windows in the form of the open source *GNUstep* environment. This means that anyone with access to a GNUstep supported platform can learn Objective-C for free, though if your ultimate objective is to develop for iOS, you will at some point need access to an Intel based Mac computer system.

Perhaps one key advantage to using a Mac OS X system for learning Objective-C comes in the form of access to Apple's Xcode development environment. Other than references to Xcode in early chapters, however, the remainder of this book is intended to be as platform agnostic as possible.

Chapter 2. The History of Objective-C

Before learning the intricacies of a new programming language it is often worth taking a little time to learn about the history and legacy of that language. In this chapter of *Objective-C 2.0 Essentials* we will provide a brief overview of the origins of Objective-C and the business history that ultimately led to it becoming the programming language of choice for both Mac OS X and iOS.

2.1 The C Programming Language

Objective-C is based on a programming language called, quite simply, *C*. The origins of the C programming language can be traced back nearly 40 years to two engineers named Dennis Ritchie and Ken Thompson working at what is now known as AT&T Bell Labs. At the time, the two were working on developing the UNIX operating system on PDP-7 and PDP-11 systems. After attempts to write this operating system using assembly language (essentially using sequences of instruction codes understood by the processor), it was decided that a higher level, more programmer friendly programming language was required to handle the complexity of an operating system such as UNIX. The first attempt was a language called *B*. The *B* language, which was based on a language called *BCPL*, was found to be lacking. Taking the next initial from the *BCPL* name, the *C* language was created and subsequently used to write much of the UNIX operating system kernel and infrastructure. As far as we can tell, *C* was so successful that new languages named *P* and *L* never needed to be created.

2.2 The Smalltalk programming Language

The C programming language is what is known as a *procedural* language. As such, this means that it lacks features such as object oriented programming. Object oriented programming advocates the creation of small, clearly defined code objects that can be assembled and reused to create more complex systems.

An early attempt at an object-oriented programming language was developed by a team including Alan Kay (who later went to work for Apple) and Dan Ingalls at Xerox PARC (Palo Alto Research Center) in the 1970s. This language is known as Smalltalk.

2.3 C meets Smalltalk

An interesting history lesson so far, but what does this have to with Objective-C? Well, in the 1980s, two developers named Brad Cox and Tom Love extended the C programming language to support the object oriented features of Smalltalk. This melding of languages ultimately culminated in the creation of Objective-C. Objective-C was subsequently adopted by the Free Software Foundation and released under the terms of the GNU Public License (GPL).

2.4 Objective-C and Apple

To understand how Objective-C, a language based on two 40 year old programming languages, ended up being the language of choice on Mac OS X and the latest cutting edge smart phones and tablets from Apple it is necessary to move away from technology for a while and talk about business.

In the 1980s Steve Jobs and Steve Wozniak founded Apple Computer. After many years of success, Steve Jobs hired a marketing wizard from PepsiCo called John Sculley to help take Apple to the next level of business success. To cut a long story short, a boardroom battle ensued and Steve Jobs got pushed out of the company (for the long version of the story pick up a used copy of John Sculley's book *Odyssey: From Pepsi to Apple*) leaving John Sculley in charge.

After leaving Apple, Jobs started a new company called NeXT to design an entirely new generation of computer system. The operating system developed by NeXT to run on these computers was called NeXTstep. In order to develop NeXTstep, NeXT licensed Objective-C. NeXT subsequently joined forces with Sun Microsystems to create a standardized version of NeXTstep named OPENstep which the Free Software Foundation then adopted as GNUstep.

During the 1990s, John Sculley left Apple and a procession of new CEOs came and went. During this time, Apple had been losing market share and struggling to come out with a new operating system to replace the aging Mac OS. After a number of failed attempts and partnerships, it was eventually decided that rather than try to write a new operating system, Apple should acquire a company that already had one. During Gil Amelio's brief reign as CEO, a shortlist of two companies was drawn up. One was a company called Be, Inc. founded by a former Apple employee named Jean-Louis Gassée, and the other was NeXT.

Ultimately, NeXT was selected and Steve Jobs once again joined Apple. In another boardroom struggle (another long story as outlined in Gil Amelio's book *On the Firing Line: My 500 Days at Apple*) Steve Jobs pushed out Gil Amelio and once again became CEO of the company he had founded all those years ago.

The rest, as they say, is history. NeXTStep formed the foundation of what became Mac OS X, bringing with it Objective-C. Mac OS X was subsequently modified to provide the iOS operating system for the spectacularly successful iPhone and iPad devices.

Chapter 3. Installing Xcode and Compiling Objective-C on Mac OS X

In later chapters we will look at how to install and use Objective-C on Windows and Linux systems for those that do not have access to Mac OS X. If you are planning to develop iOS applications (or Mac OS X applications for that matter), however, you are going to need to use an Intel based Mac OS X system at some point in the future.

Perhaps the biggest advantage of using Mac OS X as your Objective-C learning platform (aside from the ability to develop iOS and Mac OS X applications) is the fact that you get to use Apple's Xcode development tool. Xcode is a powerful and easy to use development environment that is available for a minimal charge to anyone fortunate enough to own an Apple computer running Mac OS X.

In this chapter we will cover the steps involved in installing Xcode and writing and compiling a simple Objective-C program in this environment..

3.1 Installing Xcode on Mac OS X

Xcode may or may not be pre-installed on your Mac OS X system. To find out if you already have it, open the Finder and look for it in the *Developer* subfolder of the *Applications* folder. If the *Developer* folder does not exist, or does not contain Xcode then you will need to install it.

The best way to obtain Xcode is to download it from the Apple web site. The URL to download Xcode is <http://developer.apple.com/technology/xcode.html>.

In order to download Xcode 4.3 with the iOS 5 SDK, you will either need to be a member of the iOS Developer programs or purchase a copy from the Mac App Store. The download is over 3.5GB in size and may take a number of hours to complete depending on the speed of your internet connection.

3.2 Starting Xcode

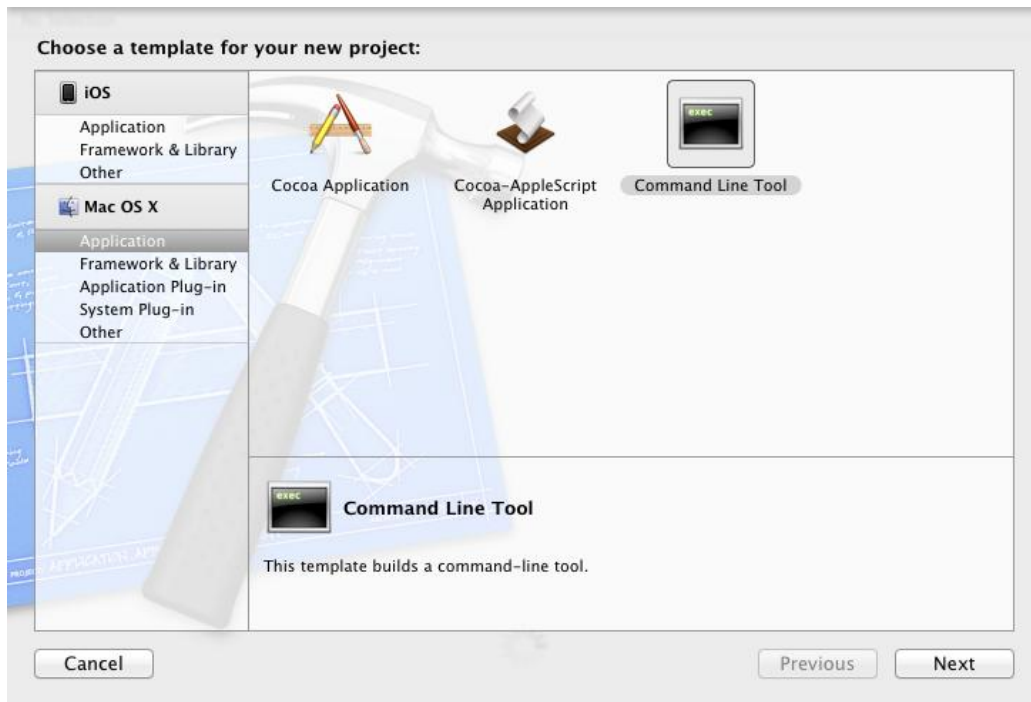
Having successfully installed the SDK and Xcode, the next step is to launch it so that we can write and then create a sample iPhone application. To start up Xcode, open the Finder and search for *Xcode*. Since you will be making frequent use of this tool, take this opportunity to drag and drop it into your dock for easier access in the future. Click on the Xcode icon in the dock to launch the tool.

Once Xcode has loaded, and assuming this is the first time you have used Xcode on this system, you will be presented with the *Welcome* screen from which you are ready to proceed:

Installing Xcode and Compiling Objective-C on Mac OS X



Click on the option to *Create a new Xcode project* to display the template selection screen. Within the template selection screen, select the *Application* entry listed beneath *MacOS X* in the left hand panel followed by *Command Line Tool* in the main panel:



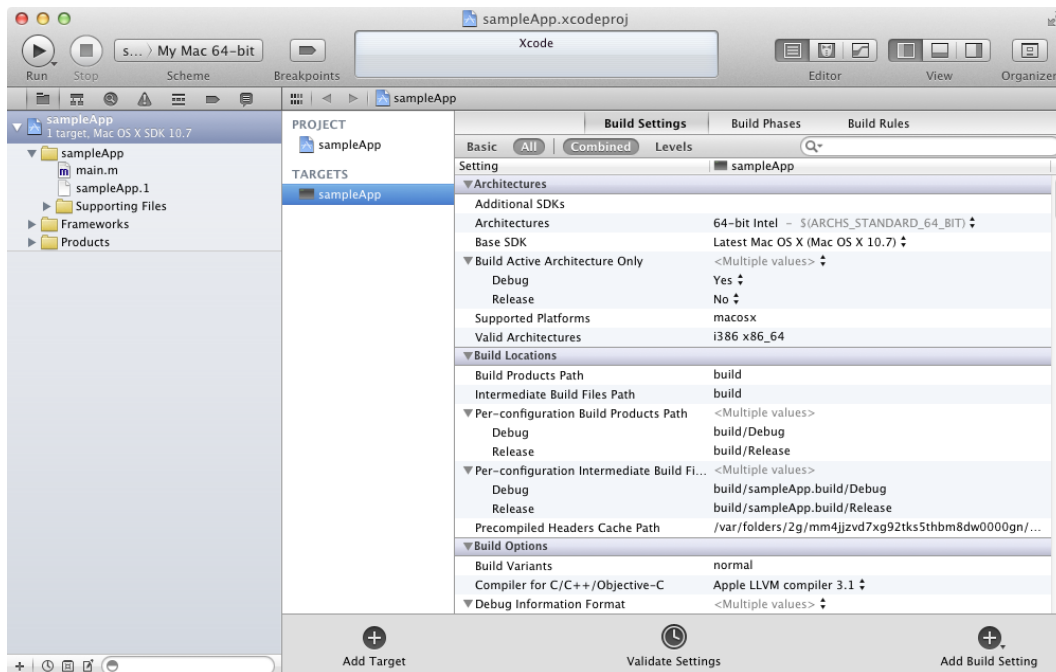
Click *Next* and on the resulting options panel name the project *sampleApp* and select *Foundation* from the *Type* menu. Also verify that the *Use Automatic Reference Counting* option is selected.

Installing Xcode and Compiling Objective-C on Mac OS X

Before Automatic Reference Counting (ARC) support was introduced to recent versions of Apple's compiler, an Objective-C programmer was responsible for retaining and releasing objects in application code. This typically entailed manually adding *retain* and *release* method calls to code in order to manage memory usage. Failing to release an object would result in memory leaks (whereby a running application's memory usage increases over time), whilst releasing an object too soon typically caused an application to crash. ARC is implemented by Apple's LLVM compiler which scans the source code and automatically inserts appropriate retain and release calls prior to compiling the code, thereby making the life of the Objective-C programmer much easier.

Click *Next* and on the subsequent screen choose a suitable location on the local file system for the project to be created before clicking on the *Create* button.

Xcode will subsequently create the new project and open the main Xcode window:



3.3 Writing an Objective-C Application with Xcode

Xcode will create skeleton versions of the files needed to build a command-line based Objective-C application. Objective-C source files are identified by the *.m* filename extension. In the case of our example, Xcode has pre-created a main source file named *main.m* and populated it with some basic code ready for us to edit. To view the code, select *main.m* from the list of files so that the code appears in the editor area. The skeleton code reads as follows:

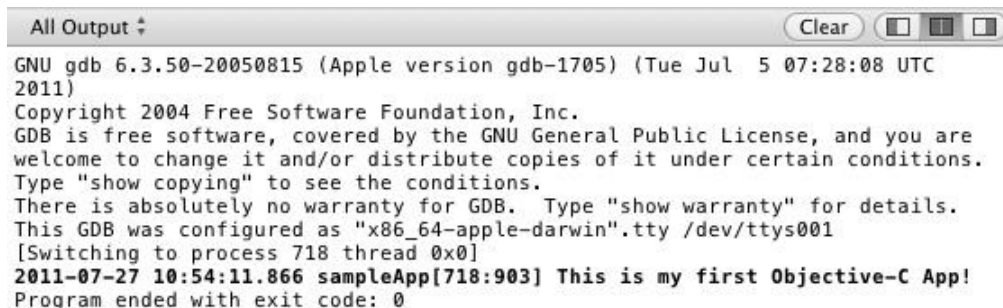
```
#import <Foundation/Foundation.h>
```

```
int main(int argc, const char * argv[])
{
    @autoreleasepool {
        // insert code here...
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

Modify the `NSLog` line so that it reads:

```
NSLog(@"This is my first Objective-C App!");
```

With the change made, the next step is to compile and run the application. Since the code is intended to display a message in the console, the first step is to make sure the Xcode console window is displayed by selecting the *View -> Debug Area -> Activate Console* menu option. Next, run the application by selecting the *Run* option located in the Xcode toolbar. Once this option has been selected, Xcode will compile the source code and run the application, displaying the message in the Xcode console panel:



```
All Output ↓ Clear [ ] [ ] [ ]
GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Tue Jul  5 07:28:08 UTC
2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys001
[Switching to process 718 thread 0x0]
2011-07-27 10:54:11.866 sampleApp[718:903] This is my first Objective-C App!
Program ended with exit code: 0
```

3.4 Compiling Objective-C from the Command Line

While Xcode provides a powerful environment that will prove invaluable for larger scale projects, for compiling and running such a simple application as the one we have been working with in this chapter it is a little bit of overkill. It is also a fact that some developers feel that development environments like Xcode just get in the way and prefer to use a basic editor and command line tools to develop applications. After all, in the days before integrated development environments came into favor, this was how all applications were developed.

Whilst we are not suggesting that everyone abandon Xcode in favor of the *vi* editor and GNU compiler, it is useful to know that the option to work from the command line is available.

Before attempting to compile code from the command line, the first step is to ensure that the *Xcode Command Line Tools* are installed. To achieve this, select the *Xcode -> Preferences* menu option and in the preferences dialog select the *Downloads* category. If the Command Line Tools are not yet installed, click on the *Install* button to begin the installation process.

Using your favorite text or programming editor, create a file named *hello.m* containing the following Objective-C code:

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

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        // insert code here...
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

Save the file and then open a Terminal window (if you aren't already working in one), change directory to the folder containing the source file and compile the application with the following command:

```
clang -framework Foundation hello.m -o hello
```

Assuming the code compiles without error it can be run as follows:

```
./hello
2012-04-18 13:27:11.621 hello[275:707] Hello, World!
```

Compared to using Xcode that seems much simpler, but keep in mind that the power of Xcode really becomes evident when you start developing larger scale projects.

3.5 Summary

The goal of this chapter has been to outline the steps involved in installing the Xcode development environment on Mac OS X. Objective-C programs can be written and compiled both from within Xcode and via the command prompt in a Terminal window. A brief overview

Installing Xcode and Compiling Objective-C on Mac OS X

of memory management and Automatic Reference Counting has been provided followed by the creation, compilation and execution of a simple Objective-C program.

Chapter 4. Objective-C 2.0 Data Types

When we look at the different types of software that run on computer systems, from financial applications to graphics intensive games, it is easy to forget that computers are really just binary machines. Binary systems work in terms of 0 and 1, true or false, set and unset. All the data sitting in RAM, stored on disk drives and flowing through circuit boards and buses are nothing more than sequences of 1s and 0s. Each 1 or 0 is referred to as a *bit* and bits are grouped together in blocks of 8, each group being referred to as a *byte*. When people talk about 32-bit and 64-bit computer systems they are talking about the number of bits that can be handled simultaneously by the CPU bus. A 64-bit CPU, for example, is able to handle data in 64-bit blocks, resulting in faster performance than a 32-bit based system.

Humans, of course, don't think in binary. We work with decimal numbers, letters and words. In order for a human to easily (easily being a relative term in this context) program a computer, some middle ground between human and computer thinking is needed. This is where programming languages such as Objective-C come into play. Programming languages allow humans to express instructions to a computer in terms and structures we understand, and then compile it down to a format that can be executed by a computer.

One of the fundamentals of any program involves data, and programming languages such as Objective-C define a set of *data types* that allow us to work with data in a format we understand when writing a computer program. For example, if we want to store a number in an Objective-C program we could do so with syntax similar to the following:

```
int mynumber = 10;
```

In the above example, we have created a variable named *mynumber* of data type *integer* by using the keyword *int*. We then assigned the value of 10 to this variable. Once we know that *int* means we are specifying a variable of integer data type we have an understanding of what is happening in this particular line of an Objective-C program. When we compile the source code down to the machine code used by the CPU, the number 10 is seen by the computer in binary as:

```
1010
```

Similarly, we can express a letter, the visual representation of a digit ('0' through to '9') or punctuation mark (referred to in computer terminology as *characters*) using the following syntax:

```
char myletter = 'c';
```

Once again, this is understandable by a human programmer, but gets compiled down to a binary sequence for the CPU to understand. In this case, the letter 'c' is represented by the decimal number 99 using the ASCII table (an internationally recognized standard that assigns numeric values to human readable characters). When converted to binary, it is stored as:

```
10101100011
```

Now that we have a basic understanding of the concept of data types and why they are necessary we can take a closer look at the different data types and qualifiers supported by Objective-C.

4.1 int Data Type

The Objective-C *int* data type can store a positive or negative whole number (in other words a number with no decimal places). The actual size or range of integer that can be handled by the *int* data type is machine and compiler implementation dependent. Typically the amount of storage allocated to *int* values is either 32-bit or 64-bit depending on the implementation of Objective-C on that platform or the CPU on which the compiler is running. It is important to note, however, that the operating system also plays a role in whether *int* values are 32 or 64-bit. For example the CPU in a computer may be 64-bit but the operating system running on it may only be 32-bit.

For example, on a 32-bit implementation, the maximum range of an unsigned *int* is 0 to 4294967295. On a 64-bit system this range would be 0 to 18,446,744,073,709,551,615. When dealing with *signed int* values, the ranges are $-2,147,483,648$ to $+2,147,483,647$ and $-9,223,372,036,854,775,808$ to $+9,223,372,036,854,775,807$ for 32-bit and 64-bit implementations respectively.

When writing an Objective-C program, the only guarantee you have is that an *int* will be *at least* 32-bits wide. To avoid future problems when compiling the code on other platforms it is safer to limit *int* values to the 32-bit range, rather than assume that 64-bit will be available.

By default, *int* values are decimal (i.e. based on number base 10). To express an *int* in Octal (number base 8) simply precede the number with a zero (0). For example:

```
int myoctal = 024;
```

Similarly, an *int* may be expressed in number base 16 (hexadecimal) by preceding the number with *0x*, for example:

```
int myhex = 0xFFA2;
```

4.2 char Data Type

The Objective-C char data type is used to store a single character such as a letter, numerical digit or punctuation mark or space character. For example, the following lines assign a variety of different characters to char type variables:

```
char myChar = 'w';  
char myChar = '2';  
char myChar = ':';
```

4.2.1 Special Characters/Escape Sequences

In addition to the standard set of characters outlined above, there is also a range of *special characters* (also referred to as *escape sequences*) available for specifying items such as a new line or tab. These special characters are identified by prefixing the character with a backslash (a concept referred to as *escaping*). For example, the following assigns a new line to the variable named newline:

```
char newline = '\n';
```

In essence, any character that is preceded by a backslash is considered to be a special character and is treated accordingly. This raises the question as to what to do if you actually want a backslash character. This is achieved by *escaping* the backslash itself:

```
char myslash = '\\'; Assign a backslash to a variable
```

Commonly used special characters supported by Objective-C are as follows:

\a - Sound alert

\b - Backspace

\f - Form feed

\n - New line

\r - Carriage return

\t - Horizontal tab

\v - Vertical tab

\\ - Backslash

\" - Double quote (used when placing a double quote into a string declaration)

\' - Single quote (used when placing a single quote into a string declaration)

4.3 float Data Type

The Objective-C *float* data type is used to store *floating point* values, in other words values containing decimal places. For example, 456.12 would be stored in a *float* data type. In practice, all floating point values are stored as a different data type (called *double*) by default. We will be covering the *double* data type next, but if you specifically want to use a *float* data type, you must append an *f* onto the end of the value. For example:

```
float myfloat = 123.432f
```

For convenience when working with exceptionally large numbers, both floating point and double data type values can be specified using scientific notation (also known as standard form or exponential notation). For example, we can express 67.7×10^4 in Objective-C as:

```
float myfloat = 67.7e4
```

4.4 double Data Type

The Objective-C *double* data type is used to store larger values than can be handled by the *float* data type. The term *double* comes from the fact that a *double* can handle values twice the size of a *float*. As previously mentioned, all floating point values are stored as double data types unless the value is followed by an 'f' to specifically specify a float rather than as a double.

4.5 id Data Type

As we will see in later chapters of this book, Objective-C is an object oriented language. As such much of the way a program will be structured is in the form of reusable objects. These objects are called upon to perform tasks and return results. Often, the information passed into an object and the results returned will be in the form of yet another object. The *id* data type is a general purpose data type that can be used to store a reference to any object, regardless of its type.

4.6 BOOL Data Type

Objective-C, like other languages, includes a data type for the purpose of handling true or false (1 or 0) conditions. Such a data type is declared using either the *_Bool* or *BOOL* keywords. Both of the following expressions are valid:

```
_Bool flag = 0;  
BOOL secondflag = 1;
```

4.7 Objective-C Data Type Qualifiers

So far we have looked at the basic data types offered within the context of the Objective-C programming language. We have seen that data types are provided for a number of different data declaration and storage needs and that each data type has associated with it some constraints in terms of what kind of data it can hold. In fact, it is possible to modify some of these constraints by using *qualifiers*. A number of such qualifiers are available and we will look at each one in turn in the remainder of this chapter.

4.7.1 long

The *long* qualifier is used to extend the value range of a data type. For example, to increase the range of an integer variable, the declaration is prefixed by the qualifier:

```
long int mylargeint;
```

The amount by which a data type's range is increased through the use of the long qualifier is system dependent, though on many modern systems *int* and *long int* both have the same range, making use of the qualifier unnecessary. The *long* qualifier may also be applied to the *double* data type. For example:

```
long double mydouble;
```

4.7.2 long long

It is safe to think of the *long long* qualifier as being equivalent to *extra long*. In the case of an *int* data type, the application of a *long long* qualifier typically will change the range from 32-bit up to 64-bit:

```
long long int mylargeint;
```

4.7.3 short

So far we have looked at qualifiers that increase the storage space, and thereby the value range, of data types. The *short* qualifier can be used to reduce the storage space and range of the *int* data type. This effectively reduces the integer to 16-bits in width, limiting the *signed* value range to $-32,768$ to $+32,767$:

```
short int myshort;
```

4.7.1 signed / unsigned

By default, an integer is assumed to be signed. In other words, the compiler assumes that an integer variable will be called upon to store either a negative or a positive number. This limits the extent that the range can reach in either direction. For example, a 32-bit *int* has a range of 4,294,967,295. In practice, because the value could be positive or negative the range is actually -2,147,483,648 to +2,147,483,647. If we know that a variable will never be called upon to store a negative value, we can declare it as unsigned, thereby extending the (positive) range to 0 to +4,294,967,295. An unsigned int is specified as follows:

```
unsigned int myint;
```

Qualifiers may also be combined, for example to declare an unsigned, short integer:

```
unsigned short int myint = 10;
```

Note that when using *unsigned*, *signed*, *short* and *long* with integer values, the *int* keyword is optional. The following are all valid:

```
short myint;  
long myint;  
unsigned myint;  
signed myint;
```

4.8 Summary

Data types are the basic building blocks of just about every programming language and Objective-C is no exception. Now that we have covered these basics we will move on to the next chapter and begin talking about the use of variables.

Chapter 5. Working with Variables and Constants in Objective-C

In the previous chapter we looked at the basic data types supported by Objective-C. Perhaps the second most basic aspect of programming involves the use of variables and constants. Even the most advanced and impressive programs use variables in one form or another. In this chapter of *Objective-C 2.0 Essentials* we will cover everything that an Objective-C programmer needs to know about variables.

5.1 What is an Objective-C Variable

Variables are essentially locations in computer memory reserved for storing the data used by an application. Each variable is given a name by the programmer and assigned a value. The name assigned to the variable may then be used in the Objective-C code to access the value assigned to the variable. This access can involve either reading the value of the variable, or changing the value. It is, of course, the ability to change the value of variables which gives them the name *variable*.

A variable must be declared as a particular type such as an integer, a character, a float or double. Objective-C is what is known as a *strongly typed* language in that once a variable has been declared as a particular type it cannot subsequently be changed to a different type. While this may come as a shock to those familiar with loosely typed languages such as Ruby it will be familiar to Java, C, C++ or C# programmers. Whilst it is not possible to change the type of a variable it is possible to disguise the variable as another type under certain circumstances. This involves a concept known as *type casting* and will be covered later in this chapter.

Variable declarations require a type, a name and, optionally a value assignment. The following example declares an integer variable called *interestRate* but does not initialize it:

```
int interestRate;
```

The following example declares and initializes a variable using the assignment operator (=):

```
int interestRate = 10;
```

Similarly, a new value may be assigned to a variable at any point after it has been declared.

```
double interestRate = 5.5456; //Declare the variable and initialize it to  
5.5456  
interestRate = 10.98; // variable now equals 10.98  
interestRate = 20.87; // variable now equals 20.87
```

5.2 What is an Objective-C Constant?

A constant is similar to a variable in that it provides a named location in memory to store a data value. Constants differ in one significant way in that once a value has been assigned to a constant it cannot subsequently be changed.

Constants are particularly useful if there is a value which is used repeatedly throughout the application code. Rather than use the value each time, it makes the code easier to read if the value is first assigned to a constant which is then referenced in the code. For example, it might not be clear to someone reading your Objective-C code why you used the value 5 in an expression. If, instead of the value 5, you use a constant named *interestRate* the purpose of the value becomes much clearer. Constants also have the advantage that if the programmer needs to change a widely used value it only needs to be changed once in the constant declaration and not each time it is referenced.

As with variables, constants have a type, a name and a value. Unlike variables, constants must be initialized at the same time that they are declared and must be prefixed with the `const` keyword:

```
const int interestRate = 10;
```

Once declared, it is not possible to assign a new value to the constant. The following code will cause the Objective-C compiler to report an error that reads "error: assignment of read-only variable":

```
const int interestRate = 10;
interestRate = 5; // invalid attempt to assign new value to read-only
const
```

Note that the value of a constant, unlike a variable, must be assigned at the point it is declared. For example, the following code will not compile:

```
const int interestRate;
interestRate = 10; // invalid attempt to initialize constant after
declaration
```

The above code will, once again, result in a compilation error.

5.3 Type Casting Objective-C Variables

As previously mentioned, Objective-C is a strongly typed language. In other words, once a variable has been declared as a specific data type, that type cannot be changed. It is possible, however, to make a variable behave as a different type using a concept known as *type casting*.

Suppose we have two variables declared as doubles. We need to multiply these together and display the result:

```
double balance = 100.54;
double interestRate = 5.78;
double result = 0;

result = balance * interestRate;

NSLog(@"The result is %f", result);
```

When executed, we will get the following output:

```
The result is 581.121200
```

Now, suppose that we wanted the result to the nearest whole number. We can achieve this by casting the type of both *double* values to type *int* within the arithmetic expression:

```
double balance = 100.54;
double interestRate = 5.78;
double result;

result = (int) balance * (int) interestRate;

NSLog(@"The result is %f", result);
```

When compiled and run, the output will now read:

```
The result is 500.000000
```

It is important to note that type casting only changes the way the value is read from the variable on that one occasion. It does not change the variable type or the value stored in any way. After the type cast, *balance* is still a *double* and still contains the value 100.54.