Objective-C 2.0 Essentials

Third Edition

Objective-C 2.0 Essentials – Third Edition

ISBN-13: 978-1480262102

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

Rev. 3.0

Table of Contents

Chapter 1

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.

Fully updated for Modern Objective-C syntax, 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, though if your ultimate objective is to develop for iOS or Mac OS X, 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. Another benefit of learning Objective-C on a Mac OS X system is that as new features are added to the language, those improvements are typically made available on Mac OS X before they make it onto other operating system platforms. That being said, other than references to Xcode in early chapters, the remainder of this book is intended to be as platform agnostic as possible.

Chapter 2

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.

2.5 Modern Objective-C

As the decades passed by, aspects of Objective-C such as some elements of language syntax and memory management began to appear somewhat dated, particularly when compared to more contemporary languages such Java and C#. In recognition of this fact Objective-C has continued to evolve and, in recent years in particular, a number of additions and improvements have been made to the language to make the task of writing code easier and less error prone for the programmer. These improvements have combined to create what is typically referred to as “Modern Objective-C”.

Chapter 3

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

Although Objective-C is available on a range of platforms via the GNUstep environment, if you are planning to develop iOS or Mac OS X applications 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 an integrated development environment (IDE) within which you will code, compile, test and debug your iOS and Mac OS X applications.

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 Identifying if you have an Intel or PowerPC based Mac

Only Intel based Mac OS X systems can be used to run the latest versions of Xcode. If you have an older, PowerPC based Mac then you will need to purchase a new system before you can begin your app development project. If you are unsure of the processor type inside your Mac, you can find this information by clicking on the Apple logo in the top left hand corner of the screen and selecting the *About This Mac* option from the menu. In the resulting dialog, check the *Processor* line. Figure 3-1 illustrates the results obtained on an Intel based system.

If the dialog on your Mac does not reflect the presence of an Intel based processor then your current system is, sadly, unsuitable as a platform for running latest versions of Xcode.

In addition, the current edition of the Xcode environment requires that the version of Mac OS X running on the system be version 10.7.4 or later. If the “About This Mac” dialog does not indicate that Mac OS X 10.7.4 or later is running, click on the *Software Update…* button to download and install the appropriate operating system upgrades.

Figure 3-1

3.2 Installing the Xcode Development Environment

The best way to obtain the latest version of Xcode development environment is to download it from the Apple Developer Center web site at:

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

The download is over 1.6GB in size and may take a number of hours to complete depending on the speed of your internet connection.

3.3 Starting Xcode

Having successfully installed Xcode, the next step is to launch it so that we can write and then run a sample Objective-C application. To start up Xcode, open the Finder and search for *Xcode.app*. 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:

Figure 3-2

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:

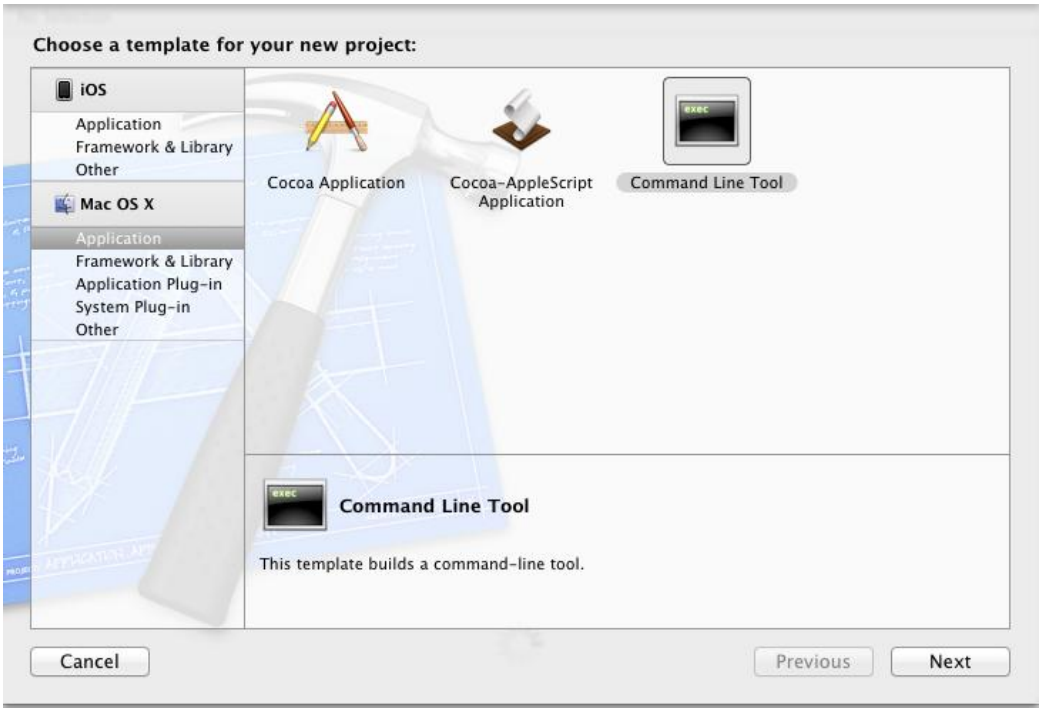


Figure 3-3

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.

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. Throughout the remainder of this book the assumption will be made that automatic reference counting is enabled when sample code is compiled.

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:

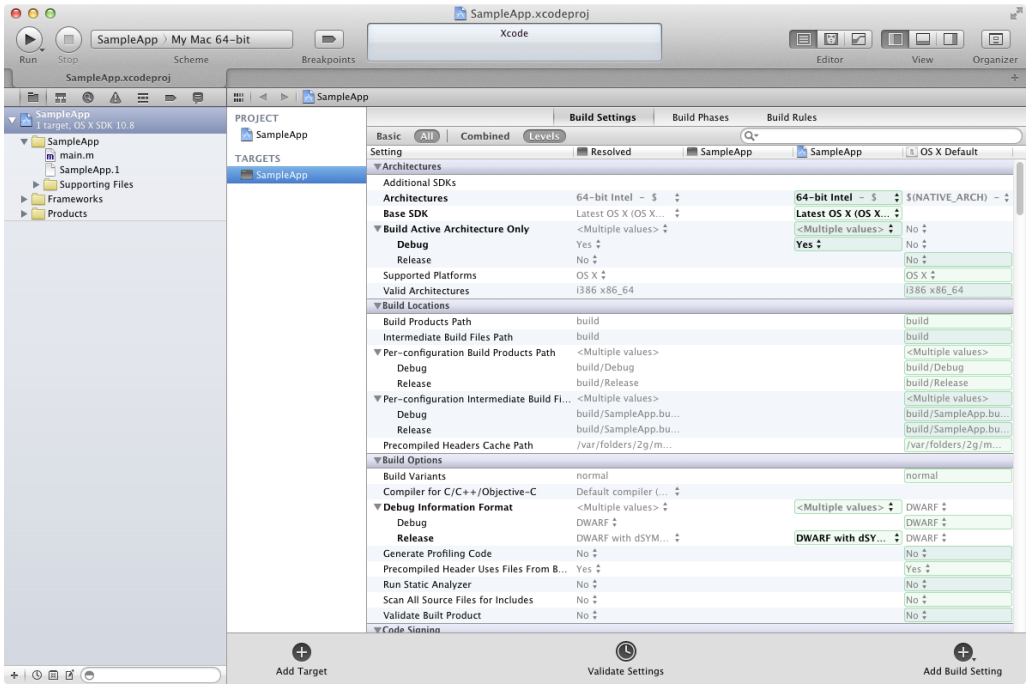


Figure 3-4

Before proceeding we should take some time to look at what Xcode has done for us. Firstly it has created a group of files that we will need to create our command-line based application. Some of these are Objective-C source code files (with a .m extension) where we will enter the

code to make our application work, whilst others are header or interface files (.h) that are included by the source files and are where we will also need to put our own declarations and definitions.

The list of files is displayed in the *Project Navigator* located in the left hand panel of the main Xcode project window. A toolbar at the top of this panel contains tabs to change the information displayed in this panel including options to display debugging information, log history and issues such as complication warnings and errors. In Figure 3-5, for example, the option has been selected to display the Issues Navigator (which in this case displays some complication errors):

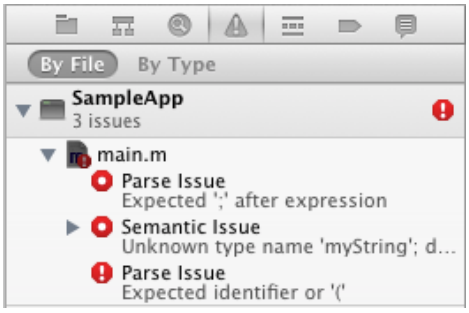


Figure 3-5

By default, the center panel of the window shows the build settings for the application. This includes, amongst other settings, the project identifier specified during the project creation process and the target architecture and operating system (in this case Mac OS X and Intel 64-bit).

In addition to the Build Settings screen, tabs are provided to view and modify additional settings consisting of Build Phases and Build Rules. To return to these panels at any future point in time, make sure the *Project Navigator* is selected in the left hand panel (selected using the folder icon in the toolbar at the top of the panel) and select the top item (the project name) in the navigator list.

When a source file is selected from the list in the navigator panel, the contents of that file will appear in the center panel where it may then be edited. To open the file in a separate editing window, simply double click on the file in the list.

3.4 Writing an Objective-C Application with Xcode

As previously outlined, 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 located in the Project Navigator so that the code appears in the editor area. The skeleton code reads as follows:

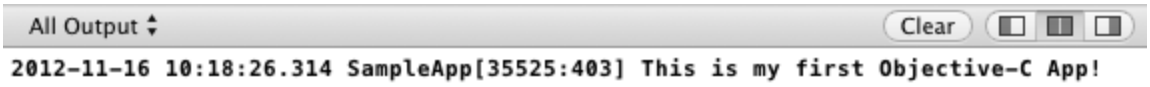
```
#import <Foundation/Foundation.h>

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:



3.5 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 -fobjc-arc -framework Foundation hello.m -o hello
```

This instructs the clang *compiler front end* to initiate the compilation of the source code located in *hello.m* and output the resulting executable binary to a file named *hello*. The clang tool is referred to as a compiler "front end" because it does not perform that actual compilation work. Instead it calls a *back end* compiler to perform the compilation work. In the case of Objective-C, this role is filled by the LLVM compiler.

The *–framework* directive (also referred to as a *build flag* or *compilation option*) instructs the compiler to include the Foundation framework in the compilation process. It is this framework on which all Objective-C applications are built. Failure to include this directive will result in compilation errors relating to undefined symbols.

Finally, the *–fobjc_arc* option instructs the compiler to use automatic reference counting (ARC).

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. In addition, Xcode includes a powerful user interface design tool called Interface Builder that will be essential when developing either iOS apps or Mac OS X applications that require a graphical user interface.

3.6 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 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

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 x 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 (named after famed mathematician George Boole). 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

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, however, 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.