2

Programming in Objective-C

In this chapter, we dive right in and show you how to write your first Objective-C program. You won't work with objects just yet; that's the topic of the next chapter. We want you to understand the steps involved in keying in a program and compiling and running it. We give special attention to this process both under Windows and on a Macintosh computer.

To begin, let's pick a rather simple example: a program that displays the phrase "Programming is fun!" on your screen. Without further ado, Program 2.1 shows an Objective-C program to accomplish this task:

Program 2.1

```
// First program example

#import Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSLog (@"Programming is fun!");

   [pool drain];
   return 0;
}
```

Compiling and Running Programs

Before we go into a detailed explanation of this program, we need to cover the steps involved in compiling and running it. You can both compile and run your program using Xcode, or you can use the GNU Objective-C compiler in a Terminal window. Let's go through the sequence of steps using both methods. Then you can decide how you want to work with your programs throughout the rest of this book.

> **Note**
>
> These tools should be preinstalled on all Macs that came with OS X. If you separately installed OS X, make sure you install the Developer Tools as well.

Using Xcode

Xcode is a sophisticated application that enables you to easily type in, compile, debug, and execute programs. If you plan on doing serious application development on the Mac, learning how to use this powerful tool is worthwhile. We just get you started here. Later we return to Xcode and take you through the steps involved in developing a graphical application with it.

First, Xcode is located in the `Developer` folder inside a subfolder called `Applications`. Figure 2.1 shows its icon.

Start Xcode. Under the File menu, select New Project (see Figure 2.2).

Figure 2.1 Xcode Icon

A window appears, as shown in Figure 2.3.

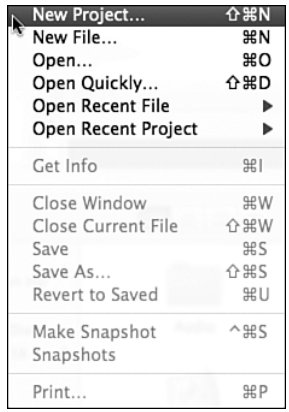

Figure 2.2 Starting a new project

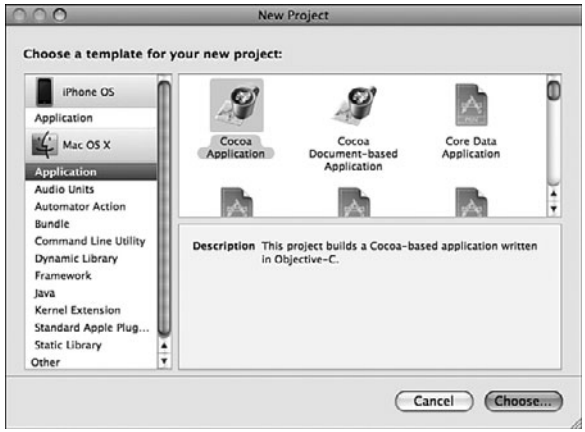

Figure 2.3 Starting a new project: selecting the application type

Scroll down the left pane until you get to Command Line Utility. In the upper-right pane, highlight Foundation Tool. Your window should now appear as shown in Figure 2.4.

Figure 2.4 Starting a new project: creating a Foundation tool

Click Choose. This brings up a new window, shown in Figure 2.5.

Figure 2.5 Xcode file list window

We'll call the first program `prog1`, so type that into the Save As field. You may want to create a separate folder to store all your projects in. On my system, I keep the projects for this book in a folder called `ObjC Progs`.

Click the Save button to create your new project. This gives you a project window such as the one shown in Figure 2.6. Note that your window might display differently if you've used Xcode before or have changed any of its options.

Now it's time to type in your first program. Select the file `prog1.m` in the upper-right pane. Your Xcode window should now appear as shown in Figure 2.7.

Objective-C source files use `.m` as the last two characters of the filename (known as its *extension*). Table 2.1 lists other commonly used filename extensions.

Table 2.1 **Common Filename Extensions**

Extension	Meaning
`.c`	C language source file
`.cc, .cpp`	C++ language source file
`.h`	Header file
`.m`	Objective-C source file
`.mm`	Objective-C++ source file
`.pl`	Perl source file
`.o`	Object (compiled) file

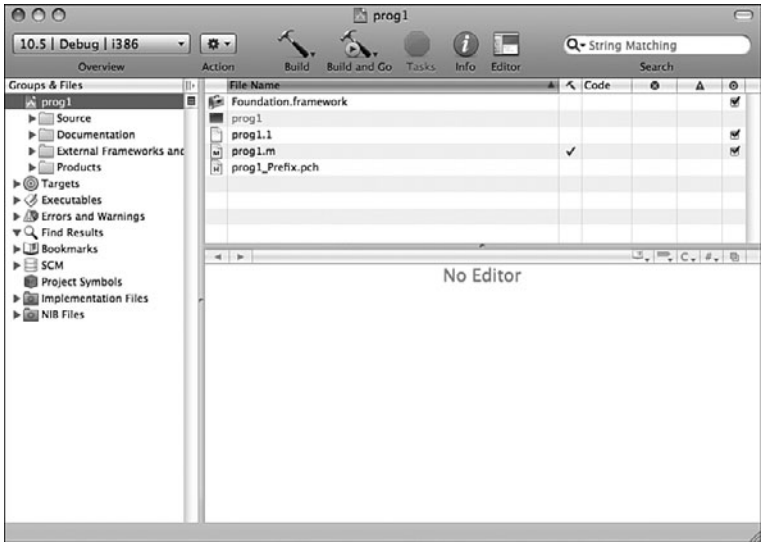

Figure 2.6 Xcode **`prog1`** project window

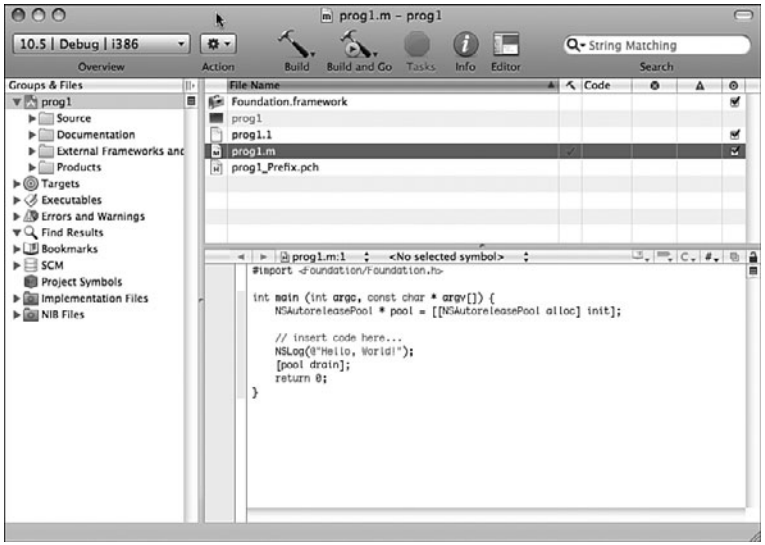

Figure 2.7 File **`prog1.m`** and **`edit window`**

Returning to your Xcode project window, the bottom-right side of the window shows the file called `prog1.m` and contains the following lines:

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

int main (int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   // insert code here...
   NSLog (@"Hello World!");
   [pool drain];
   return 0;
}
```

> **Note**
>
> If you can't see the file's contents displayed, you might have to click and drag up the bottom-right pane to get the edit window to appear. Again, this might be the case if you've previously used Xcode.

You can edit your file inside this window. Xcode has created a template file for you to use.

Make changes to the program shown in the Edit window to match Program 2.1. The line you add at the beginning of `prog1.m` that starts with two slash characters (`//`) is called a *comment*; we talk more about comments shortly.

Your program in the edit window should now look like this:

```
// First program example

int main (int argc, const char * argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSLog (@"Programming is fun!");

   [pool drain];
   return 0;
}
```

Don't worry about all the colors shown for your text onscreen. Xcode indicates values, reserved words, and so on with different colors.

Now it's time to compile and run your first program—in Xcode terminology, it's called *build and run*. You need to save your program first, however, by selecting Save from the File menu. If you try to compile and run your program without first saving your file, Xcode asks whether you want to save it.

Under the Build menu, you can select either Build or Build and Run. Select the latter because that automatically runs the program if it builds without any errors. You can also click the Build and Go icon that appears in the toolbar.

> **Note**
>
> Build and Go means "Build and then do the last thing I asked you to do," which might be Run, Debug, Run with Shark or Instruments, and so on. The first time you use this for a project, Build and Go means to build and run the program, so you should be fine using this option. However, just be aware of the distinction between "Build and Go" and "Build and Run."

If you made mistakes in your program, you'll see error messages listed during this step. In this case, go back, fix the errors, and repeat the process. After all the errors have been removed from the program, a new window appears, labeled `prog1 - Debugger Console`. This window contains the output from your program and should look similar to Figure 2.8. If this window doesn't automatically appear, go to the main menu bar and select Console from the Run menu. We discuss the actual contents of the Console window shortly.

Figure 2.8 Xcode Debugger Console window

You're now done with the procedural part of compiling and running your first program with Xcode (whew!). The following summarizes the steps involved in creating a new program with Xcode:

1. Start the Xcode application.
2. If this is a new project, select File, New Project.
3. For the type of application, select Command Line Utility, Foundation Tool, and click Choose.

4. Select a name for your project, and optionally a directory to store your project files in. Click Save.
5. In the top-right pane, you will see the file `prog1.m` (or whatever name you assigned to your project, followed by `.m`. Highlight that file. Type your program into the edit window that appears directly below that pane.
6. Save the changes you've entered by selecting File, Save.
7. Build and run your application by selecting Build, Build and Run, or by clicking the Build and Go Button.
8. If you get any compiler errors or the output is not what you expected, make your changes to the program and repeat steps 6 and 7.

Using Terminal

Some people might want to avoid having to learn Xcode to get started programming with Objective-C. If you're used to using the UNIX shell and command-line tools, you might want to edit, compile, and run your programs using the Terminal application. Here we examine how to go about doing that.

The first step is to start the Terminal application on your Mac. The Terminal application is located in the `Applications` folder, stored under Utilities. Figure 2.9 shows its icon.

Start the Terminal application. You'll see a window that looks like Figure 2.10.

Figure 2.9 Terminal program icon

You type commands after the `$` (or `%`, depending on how your Terminal application is configured) on each line. If you're familiar with using UNIX, you'll find this straightforward.

First, you need to enter the lines from Program 2.1 into a file. You can begin by creating a directory in which to store your program examples. Then you must run a text editor, such as vi or emacs, to enter your program:

```
sh-2.05a$ mkdir Progs    Create a directory to store programs in
sh-2.05a$ cd Progs       Change to the new directory
sh-2.05a$ vi prog1.m     Start up a text editor to enter program
 ..
```

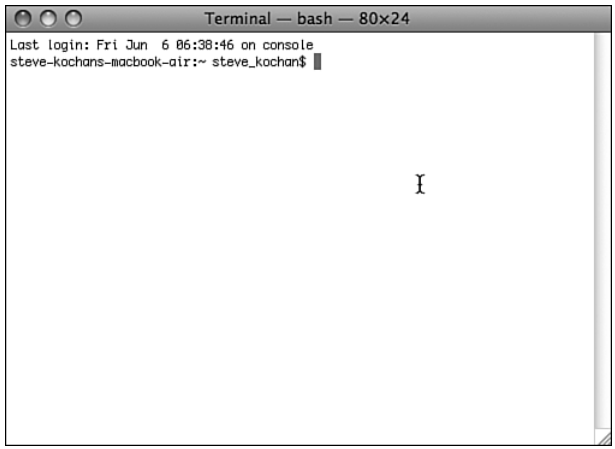

Figure 2.10 Terminal window

> **Note**
>
> In the previous example and throughout the remainder of this text, commands that you, the user, enter are indicated in boldface.

For Objective-C files, you can choose any name you want; just make sure the last two characters are `.m`. This indicates to the compiler that you have an Objective-C program.

After you've entered your program into a file, you can use the GNU Objective-C compiler, which is called gcc, to compile and link your program. This is the general format of the `gcc` command:

```
gcc -framework Foundation files -o progname
```

This option says to use information about the Foundation framework:

```
-framework Foundation
```

Just remember to use this option on your command line. *`files`* is the list of files to be compiled. In our example, we have only one such file, and we're calling it `prog1.m`. *`progname`* is the name of the file that will contain the executable if the program compiles without any errors.

We'll call the program `prog1`; here, then, is the command line to compile your first Objective-C program:

```
$ gcc -framework Foundation prog1.m -o prog1 Compile prog1.m & call it prog1
$
```

The return of the command prompt without any messages means that no errors were found in the program. Now you can subsequently execute the program by typing the name `prog1` at the command prompt:

```
$ prog1          Execute prog1
```

```
sh: prog1: command not found
$
```

This is the result you'll probably get unless you've used Terminal before. The UNIX shell (which is the application running your program) doesn't know where `prog1` is located (we don't go into all the details of this here), so you have two options: One is to precede the name of the program with the characters `./` so that the shell knows to look in the current directory for the program to execute. The other is to add the directory in which your programs are stored (or just simply the current directory) to the shell's `PATH` variable. Let's take the first approach here:

```
$ ./prog1       Execute prog1
2008-06-08 18:48:44.210 prog1[7985:10b] Programming is fun!
$
```

You should note that writing and debugging Objective-C programs from the terminal is a valid approach. However, it's not a good long-term strategy. If you want to build Mac OS X or iPhone applications, there's more to just the executable file that needs to be "packaged" into an application bundle. It's not easy to do that from the Terminal application, and it's one of Xcode's specialties. Therefore, I suggest you start learning to use Xcode to develop your programs. There is a learning curve to do this, but the effort will be well worth it in the end.

Explanation of Your First Program

Now that you are familiar with the steps involved in compiling and running Objective-C programs, let's take a closer look at this first program. Here it is again:

```
// First program example


int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSLog (@"Programming is fun!");
    [pool drain];
    return 0;
}
```

In Objective-C, lowercase and uppercase letters are distinct. Also, Objective-C doesn't care where on the line you begin typing—you can begin typing your statement at any position on the line. You can use this to your advantage in developing programs that are easier to read.

The first line of the program introduces the concept of the *comment*:

```
// First program example
```

A comment statement is used in a program to document a program and enhance its readability. Comments tell the reader of the program—whether it's the programmer or someone else whose responsibility it is to maintain the program—just what the programmer had in mind when writing a particular program or a particular sequence of statements.

You can insert comments into an Objective-C program in two ways. One is by using two consecutive slash characters (//). The compiler ignores any characters that follow these slashes, up to the end of the line.

You can also initiate a comment with the two characters / and *. This marks the beginning of the comment. These types of comments have to be terminated. To end the comment, you use the characters * and /, again without any embedded spaces. All characters included between the opening /* and the closing */ are treated as part of the comment statement and are ignored by the Objective-C compiler. This form of comment is often used when comments span many lines of code, as in the following:

```
/*
 This file implements a class called Fraction, which
 represents fractional numbers. Methods allow manipulation of
 fractions, such as addition, subtraction, etc.

 For more information, consult the document:
    /usr/docs/classes/fractions.pdf
*/
```

Which style of comment you use is entirely up to you. Just note that you can't nest the /* style comments.

Get into the habit of inserting comment statements in the program as you write it or type it into the computer, for three good reasons. First, documenting the program while the particular program logic is still fresh in your mind is far easier than going back and rethinking the logic after the program has been completed. Second, by inserting comments into the program at such an early stage of the game, you can reap the benefits of the comments during the debug phase, when program logic errors are isolated and debugged. Not only can a comment help you (and others) read through the program, but it also can help point the way to the source of the logic mistake. Finally, I haven't yet discovered a programmer who actually enjoys documenting a program. In fact, after you've finished debugging your program, you will probably not relish the idea of going back to the program to insert comments. Inserting comments while developing the program makes this sometimes tedious task a bit easier to handle.

This next line of Program 2.1 tells the compiler to locate and process a file named Foundation.h:

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

This is a system file—that is, not a file that you created. #import says to import or include the information from that file into the program, exactly as if the contents of the file were typed into the program at that point. You imported the file Foundation.h because it has information about other classes and functions that are used later in the program.

In Program 2.1, this line specifies that the name of the program is `main`:

```
int main (int argc, const char *argv[])
```

`main` is a special name that indicates precisely where the program is to begin execution. The reserved word `int` that precedes `main` specifies the type of value `main` returns, which is an integer (more about that soon). We ignore what appears between the open and closed parentheses for now; these have to do with *command-line arguments*, a topic we address in Chapter 13, "Underlying C Language Features."

Now that you have identified `main` to the system, you are ready to specify precisely what this routine is to perform. This is done by enclosing all the program *statements* of the routine within a pair of curly braces. In the simplest case, a statement is just an expression that is terminated with a semicolon. The system treats all the program statements included between the braces as part of the `main` routine. Program 2.1 has four statements.

The first statement in Program 2.1 reads

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

It reserves space in memory for an *autorelease pool*. We discuss this thoroughly in Chapter 17, "Memory Management." Xcode puts this line into your program automatically as part of the template, so just leave it there for now.

The next statement specifies that a routine named `NSLog` is to be invoked, or *called*. The parameter, or *argument*, to be passed or handed to the `NSLog` routine is the following string of characters:

```
@"Programming is fun!"
```

Here, the `@` sign immediately precedes a string of characters enclosed in a pair of double quotes. Collectively, this is known as a constant `NSString` object.

> **Note**
>
> If you have C programming experience, you might be puzzled by the leading `@` character. Without that leading `@` character, you are writing a constant C-style string; with it, you are writing an `NSString` string object.

The `NSLog` routine is a function in the Objective-C library that simply displays or logs its argument (or arguments, as you will see shortly). Before doing so, however, it displays the date and time the routine is executed, the program name, and some other numbers we don't describe here. Throughout the rest of this book, we don't bother to show this text that `NSLog` inserts before your output.

You must terminate all program statements in Objective-C with a semicolon (;). This is why a semicolon appears immediately after the closed parenthesis of the NSLog call.

Before you exit your program, you should release the allocated memory pool (and objects that are associated with it) with a line such as the following:

```
[pool drain];
```

Again, Xcode automatically inserts this line into your program for you. Again, we defer detailed explanation of what this does until later.

The final program statement in main looks like this:

```
return 0;
```

It says to terminate execution of main and to send back, or *return*, a status value of 0. By convention, 0 means that the program ended normally. Any nonzero value typically means some problem occurred—for example, perhaps the program couldn't locate a file that it needed.

If you're using Xcode and you glance back to your Debug Console window (refer to Figure 2.8), you'll recall that the following displayed after the line of output from NSLog:

```
The Debugger has exited with status 0.
```

You should understand what that message means now.

Now that we have finished discussing your first program, let's modify it to also display the phrase "And programming in Objective-C is even more fun!" You can do this by simply adding another call to the NSLog routine, as shown in Program 2.2. Remember that every Objective-C program statement must be terminated by a semicolon.

Program 2.2

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

int main (int argc, const char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSLog (@"Programming is fun!");
    NSLog (@"Programming in Objective-C is even more fun!");

    [pool drain];
    return 0;
}
```

If you type in Program 2.2 and then compile and execute it, you can expect the following output (again, without showing the text that NSLog normally prepends to the output):

Program 2.2 Output

```
Programming is fun!
Programming in Objective-C is even more fun!
```

As you will see from the next program example, you don't need to make a separate call to the NSLog routine for each line of output.

First, let's talk about a special two-character sequence. The backslash (\) and the letter n are known collectively as the *newline* character. A newline character tells the system to do precisely what its name implies: go to a new line. Any characters to be printed after the newline character then appear on the next line of the display. In fact, the newline character is very similar in concept to the carriage return key on a typewriter (remember those?).

Study the program listed in Program 2.3 and try to predict the results before you examine the output (no cheating, now!).

Program 2.3

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


int main (int argc, const char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSLog (@"Testing...\n..1\n...2\n....3");
   [pool drain];
   return 0;
}
```

Program 2.3 Output

```
Testing...
..1
...2
....3
```

Displaying the Values of Variables

Not only can simple phrases be displayed with NSLog, but the values of *variables* and the results of computations can be displayed as well. Program 2.4 uses the NSLog routine to display the results of adding two numbers, 50 and 25.

Program 2.4

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

int main (int argc, const char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int sum;

   sum = 50 + 25;
   NSLog (@"The sum of 50 and 25 is %i", sum);
   [pool drain];

   return 0;
}
```

Program 2.4 Output

```
The sum of 50 and 25 is 75
```

The first program statement inside `main` after the autorelease pool is set up defines the variable `sum` to be of type `integer`. You must define all program variables before you can use them in a program. The definition of a variable specifies to the Objective-C compiler how the program should use it. The compiler needs this information to generate the correct instructions to store and retrieve values into and out of the variable. A variable defined as type `int` can be used to hold only integral values—that is, values without decimal places. Examples of integral values are `3`, `5`, `-20`, and `0`. Numbers with decimal places, such as `2.14`, `2.455`, and `27.0`, are known as *floating-point* numbers and are real numbers.

The integer variable `sum` stores the result of the addition of the two integers `50` and `25`. We have intentionally left a blank line following the definition of this variable to visually separate the variable declarations of the routine from the program statements; this is strictly a matter of style. Sometimes adding a single blank line in a program can make the program more readable.

The program statement reads as it would in most other programming languages:

```
sum = 50 + 25;
```

The number `50` is added (as indicated by the plus sign) to the number `25`, and the result is stored (as indicated by the assignment operator, the equals sign) in the variable `sum`.

The `NSLog` routine call in Program 2.4 now has two arguments enclosed within the parentheses. These arguments are separated by a comma. The first argument to the `NSLog` routine is always the character string to be displayed. However, along with the display of the character string, you often want to have the value of certain program variables displayed as well. In this case, you want to have the value of the variable `sum` displayed after these characters are displayed:

```
The sum of 50 and 25 is
```

The percent character inside the first argument is a special character recognized by the NSLog function. The character that immediately follows the percent sign specifies what type of value is to be displayed at that point. In the previous program, the NSLog routine recognizes the letter i as signifying that an integer value is to be displayed.

Whenever the NSLog routine finds the %i characters inside a character string, it automatically displays the value of the next argument to the routine. Because sum is the next argument to NSLog, its value is automatically displayed after "The sum of 50 and 25 is".

Now try to predict the output from Program 2.5.

Program 2.5

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

int main (int argc, const char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int value1, value2, sum;

   value1 = 50;
   value2 = 25;
   sum = value1 + value2;

   NSLog (@"The sum of %i and %i is %i", value1, value2, sum);

   [pool drain];
   return 0;
}
```

Program 2.5 Output

```
The sum of 50 and 25 is 75
```

The second program statement inside main defines three variables called value1, value2, and sum, all of type int. This statement could have equivalently been expressed using three separate statements, as follows:

```
int value1;
int value2;
int sum;
```

After the three variables have been defined, the program assigns the value 50 to the variable value1 and then the value 25 to value2. The sum of these two variables is then computed and the result assigned to the variable sum.

The call to the `NSLog` routine now contains four arguments. Once again, the first argument, commonly called the *format string*, describes to the system how the remaining arguments are to be displayed. The value of `value1` is to be displayed immediately following the phrase "The sum of." Similarly, the values of `value2` and `sum` are to be printed at the points indicated by the next two occurrences of the `%i` characters in the format string.

Summary

After reading this introductory chapter on developing programs in Objective-C, you should have a good feel of what is involved in writing a program in Objective-C—and you should be able to develop a small program on your own. In the next chapter, you begin to examine some of the intricacies of this powerful and flexible programming language. But first, try your hand at the exercises that follow, to make sure you understand the concepts presented in this chapter.

Exercises

1. Type in and run the five programs presented in this chapter. Compare the output produced by each program with the output presented after each program.
2. Write a program that displays the following text:

```
In Objective-C, lowercase letters are significant.
main is where program execution begins.
Open and closed braces enclose program statements in a routine.
All program statements must be terminated by a semicolon.
```

3. What output would you expect from the following program?

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

int main (int argc, const char *argv[])
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init;
   int i;

   i = 1;
   NSLog (@"Testing...");
   NSLog (@"....%i", i);
   NSLog (@"...%i",  i + 1);
   NSLog (@"..%i", i + 2);

   [pool drain];
   return 0;
}
```

4. Write a program that subtracts the value `15` from `87` and displays the result, together with an appropriate message.
5. Identify the syntactic errors in the following program. Then type in and run the corrected program to make sure you have identified all the mistakes:

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

int main (int argc, const char *argv[]);
(
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   INT sum;
   /* COMPUTE RESULT //
   sum = 25 + 37 - 19
   / DISPLAY RESULTS /
   NSLog (@'The answer is %i' sum);

   [pool drain];
   return 0;
}
```

6. What output would you expect from the following program?

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

int main (int argc, const char *argv[]))
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   int answer, result;

   answer = 100;
   result = answer - 10;

   NSLog (@"The result is %i\n", result + 5);

   [pool drain];
   return 0;
}
```