Chapter 1

Computer Programming Exposed!

In This Chapter

- ▶ Understanding the basics of computer programming
- ▶ Getting how computer languages work
- ▶ Knowing how Objective-C works

Looking at it from the outside, computer programming can appear complicated and a bit mysterious. But once I let you in on a few of the secrets, you'll realize that when you write a computer program, whether it is a small program that's just a few lines or one that is tens or even hundreds of thousands of lines, you are generally doing the same thing:

1. **Getting input — from a keyboard or touch screen, or even something stored on your computer.**

   The input might be instructions to the program itself — for example, to display the web page `https://developer.apple.com`, to print a document such as Chapter 1, to process data like "Enter your Apple ID and Password" when you log on to the Mac Dev Center (the browser is just another program), or even to process a list of credit card transactions stored on a computer.

2. **Doing something based on, or with, the input.**

   Your browser may go on the Internet and access the page corresponding to `https://developer.apple.com`, or your word processing program may display a Print dialog box and print the chapter (at least that is what mine does). Based on your input, the program may also go out and use data it has stored or even has access to over the Internet. For example, when you enter your Apple ID and password, eventually a computer accesses a database to see whether your Apple ID and password are both valid and, if so, allows you access to the site and displays the site for you.

3. **Displaying the results of your adroitness on a monitor (or storing it away for future use).**

Computers are no doubt engineering marvels. But what will make you a good programmer is not your understanding of all that wizardry. No, what will make you a good programmer is taking the time to really understand the world of the user, and what you can do with a computer to make things better. For example, when I travel, I often zone out on the fact that even though it looks like Monopoly money, foreign currency actually does amount to something in dollars. I could use a computer to keep track of my budget and convert foreign currency into dollars. Writing a program simply involves detailing the steps the computer needs to follow (in a language the computer understands — but I'll get to that). You know, something like

> Subtract the amount he just spent from the amount he started with.

or

> Multiply the amount in foreign currency times the exchange rate.

Is it hard? No, not really. It can be pedestrian, but even more often, it is fun.

Why a Computer Program Is Like a Recipe

At its heart (yes, it does have one), computer programming is actually not that alien to most people. If you don't believe me, take the following programming test. Now, don't peek ahead for the answer. Okay?

The Never-Fail Programming Test:

> Write down the recipe for making a peanut butter and jelly sandwich.

Answer:

> If what you wrote down looks anything like

```
Recipe: Peanut Butter and Jelly Sandwich
  Ingredients
    Peanut Butter
    Jelly
    2 slices of bread
  Directions
    Place the two slices of bread close to each other.
    Spread peanut butter on one slice of bread.
    Spread jelly on the other slice of bread.
    Put one slice of bread on top of the other.
```

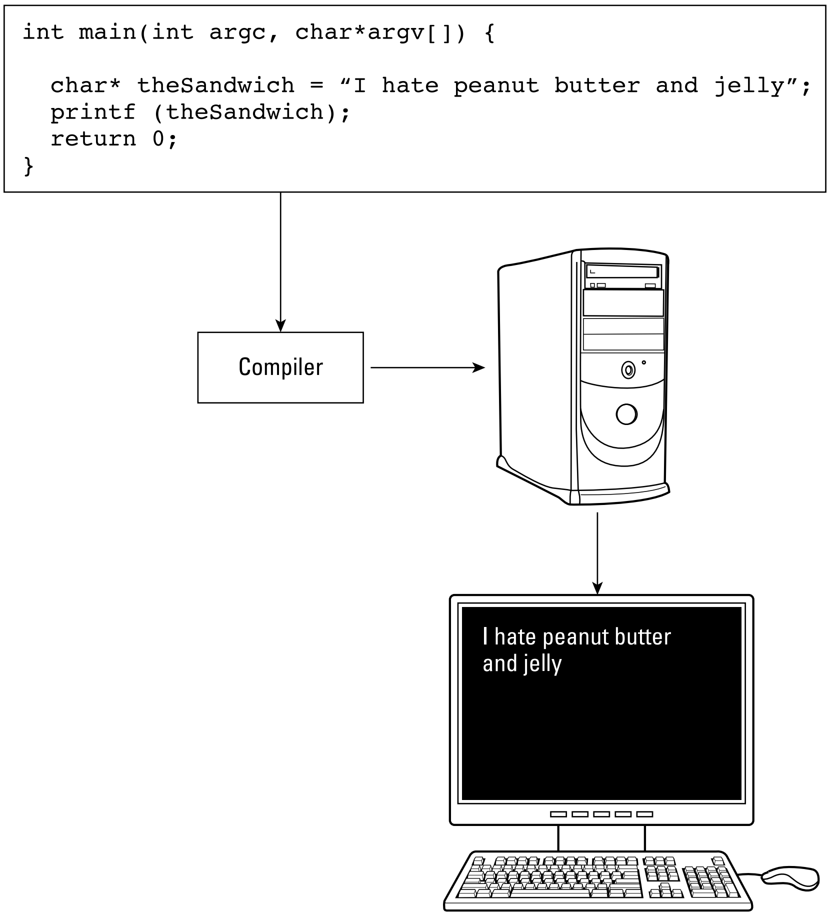
> you're ready to go.

Although this example may seem overly simple, it generally illustrates what programming is all about. When you write a program in Objective-C, all you are doing is providing a set of instructions for the computer to follow. The preceding example is not perfect, but actually it is much closer to illustrating how Objective-C programming works than you might think. So, considering the peanut butter and jelly sandwich example, here is how you get your lunch made (if you are lucky enough to have a chef):

1. **Give your chef the recipe.**
2. **He or she gets the ingredients together and then follows the instructions on what to do with the ingredients.**

   Voilà, you have a peanut butter and jelly sandwich.

Figure 1-1 shows how a computer program works, using the peanut butter and jelly sandwich example.



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

  char* theSandwich = "I hate peanut butter and jelly";
  printf (theSandwich);
  return 0;
}
```

```
I hate peanut butter
and jelly
```

Figure 1-1: The peanut butter and jelly program outputs data.

This is what you do to get that output:

1. **You write instructions for the computer to follow.**

   Unfortunately, the computer can't speak English, or read for that matter, so you use something called a *compiler* to take the instructions you write in the Objective-C language and translate them into something the computer can understand.

2. **You provide data for the computer to use.**

   In this case, you write, "I hate peanut butter and jelly," and then the computer follows the instructions you have given it on what to do with that data.

   Voilà, you see "I hate peanut butter and jelly" displayed on your computer screen.

Fundamentally, programs manipulate numbers and text, and all things considered, a computer program has only two parts: *variables* (and other structures), which "hold" data, and *instructions,* which perform operations on that data.

Examining a simple computer program

Is there really any difference between a chef reading a recipe to create a peanut butter and jelly sandwich and a computer following some instructions to display something on a monitor? Quite frankly, no.

Here is the simple Objective-C program that displays `I hate peanut butter and jelly` on the computer screen:

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

  char* theSandwich = "I hate peanut butter and jelly";

  printf (theSandwich);
  return 0;
}
```

This program shows you how to display a line of text on your computer screen. The best way to understand programming code is to take it apart line by line:

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

Ignore the first line; it's not important now. It just provides your program with some information it can use. I explain exactly what that line means in the next few chapters.

```
char* theSandwich = "I hate peanut butter and jelly";
```

`theSandwich` is what is known as a *variable*. The best way to think of it for now is as a bucket that holds some kind of data (I get more precise in Chapter 4). `char*` tells you what kind of variable it is. In this case, `theSandwich` is a bunch of characters (text) known as a *string* (technically a string is more than that, but for now, that description is good enough for our purposes). `I hate peanut butter and jelly` is the data that the variable contains.

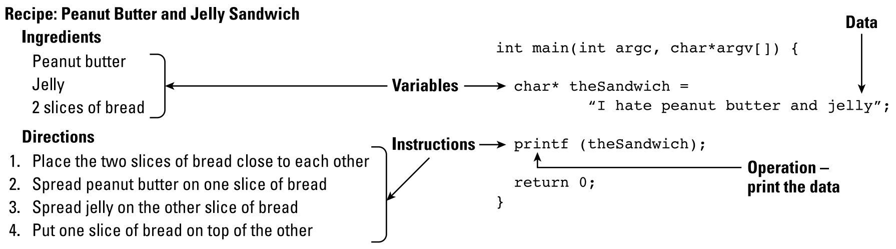
```
printf (theSandwich);
```

`printf` is an instruction (called an *operation*) that tells the computer to display whatever data is in the `theSandwich` bucket.

```
   return 0;
}
```

You can also safely ignore the last two lines for the time being.

Figure 1-2 shows the similarities between the program and the recipe for making a sandwich.


Recipe: Peanut Butter and Jelly Sandwich
Ingredients
Peanut butter
Jelly
2 slices of bread
Directions
1. Place the two slices of bread close to each other
2. Spread peanut butter on one slice of bread
3. Spread jelly on the other slice of bread
4. Put one slice of bread on top of the other
Variables
Instructions
Data
Operation – print the data

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

  char* theSandwich =
          "I hate peanut butter and jelly";

  printf (theSandwich);

  return 0;
}
```

Figure 1-2: A computer program can be compared to a peanut butter and jelly sandwich recipe.

You can think of the following ingredients as variables that represent the data. For example, `peanut butter` is the name you give to pureed peanuts (and whatever else is in peanut butter), `jelly` is the name you give to some fruit that's been processed and put in a jar, and so on.

```
Peanut butter
Jelly
2 slices of bread
```

Similarly

```
Place the two slices of bread close to each other.
Spread peanut butter on one slice of bread.
Spread jelly on the other slice of bread.
Put one slice of bread on top of the other.
```

are simply instructions on how to take the ingredients and make a sandwich. `Spread peanut butter on one slice of bread` is the instruction. Actually, spreading the peanut butter is the operation you are performing on the pureed peanuts being referenced by the `peanut butter` variable.

Understanding How Computer Languages Work

While conceptually it is pretty easy to understand computer programming — all you are doing is giving the computer a set of instructions and some data on which to perform those instructions — one of the challenges, as I mention previously, is that it's not that easy to tell a computer what to do.

Computers don't speak English, although computer scientists have been working on that for years (think of trying to do that as the Computer Scientist Full Employment Act). A computer actually has its own language made up of 1s and 0s. For that matter, Objective-C is not something a computer can understand either, but it is a language that can be turned into those 1s and 0s by using a *compiler.* A compiler is nothing more than a program that translates Objective-C instructions into computer code.

Creating a computer program

To create a computer program by using a computer language, follow these steps (see Figure 1-3):

1. **Decide what you want the computer to do.**

   You can have the computer write a line of text on the monitor or create an online multiplayer game that will take two years to complete. It really doesn't matter.

2. **Break the task you want the computer to complete into a series of modules that contain the instructions the computer follows to do what you want, and then provide the data it needs to do that.**

   The series of modules is often referred to as your *application architecture.* The data you provide to the computer can be some text, graphics, where the hidden treasure is, or the euro-U.S. dollar exchange rate.

3. **Run the instructions through the compiler.**

   A compiler is actually just another program, albeit one that uses your instructions as data for its instructions on how to turn Objective-C into computer code.

4. **Link the result to other precompiled modules.**

   As you will see, the code you write is a relatively small part of what makes up your program. The rest is made up of all the plumbing you need to run the program, open and close windows, and do all that user interface stuff. Fortunately, that code is provided for you in a form that is easy to attach (link) to your program. A linker program takes your code, identifies all the things it needs, collects all pieces (from the disk), and combines them into the executable program you see in your applications or utilities folder.

5. **Store that output somewhere.**

   You usually store the output on a hard drive, but it can be anything the computer can access, like punch cards.

6. **Run the program.**

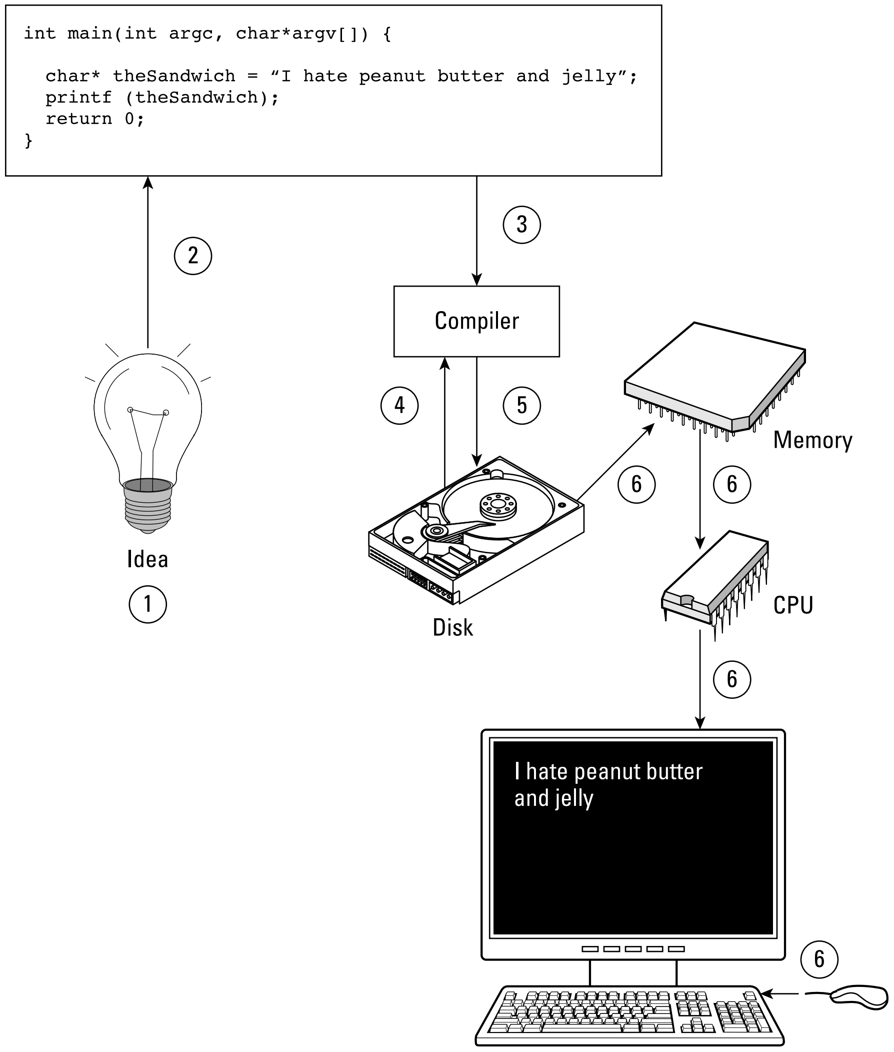
   When you want to run the program (say, the user double-clicks the program icon), the operating system (Mac OS X, for example, which is also just another program) gets the program from where it's stored and loads it into memory, and then the CPU (central processing unit) executes the instructions.

Running a computer program

Just as you don't need to be a weatherman to know which way the wind blows, you don't need to be an engineer who understands the intimate details of a computer to write a world-class application.

Most people don't find it that difficult to learn to drive a car. Although you don't have to know everything about internal combustion engines, fuel injection, drivetrains and transmissions, you do need to know a little bit about how a car works. That means you need to know how to start it, make it go forward, make it go backward, make it stop (generally a very valuable piece of information), make it turn left or right, and so on.

In the same way, you do need to know *a little bit* about how computers work to have what you do to write a computer program make sense.



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

  char* theSandwich = “I hate peanut butter and jelly”;
  printf (theSandwich);
  return 0;
}
```

Figure 1-3: How programming works.

When you run a computer program, the computer does its primary work in a part of the machine you cannot see, the central processing unit (CPU), which executes the program instructions that are loaded into the computer’s memory. (This is a fast, temporary form of storage that is in one of those chips you see when you look inside a computer, as opposed to persistent storage such as a disk drive, which is slower and offers permanent storage.) It requests the data it needs from memory, processes it, and writes new data back to memory millions of times every second.

But if the data is all in memory, the CPU needs to be able to find a particular instruction or piece of data. How does it do that?

The location in memory for each instruction and each piece of data is identified by an *address,* like the mailboxes in the post office or an apartment house you see in Figure 1-4 (and notice that the first address for a mailbox in your computer is always 0). But these are very small mailboxes that can hold only one piece of information at a time, referred to as a *byte.* So for all practical purposes, you can think of the smallest division of memory as a byte, with each byte being able to be addressed on its own. The good news is that if you need more mailboxes, they are yours for the taking. So if you get more than one letter a day, the number of mailboxes assigned to you will increase to hold all the letters you need them to.

A byte represents 8 *bi*nary digi*ts*, also known as *bits*. Most of the time, you'll use all 8 bits as a group, which could be the number 119 or the letter Z. You can also write code to access the individual bits independently, so those 8 bits could represent 8 flags that are either true (equal to 1) or false (equal to 0). I cover bytes in detail in Chapter 4.

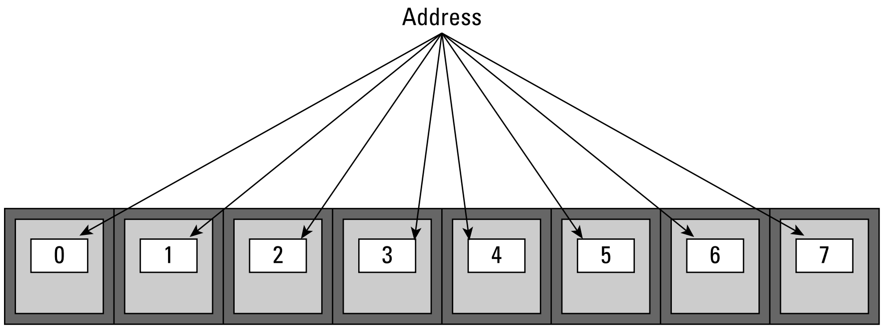


Figure 1-4: Finding things in memory.

What Is Objective-C, Anyway?

Objective-C is an object-oriented programming language, which means that it was created to support a certain style of programming. Yes, I know it is hard to believe, but even things like programming have different styles, in fact a lot of them, although the two heavyweights are object-oriented and procedural. Unless you're a dyed-in-the-wool member of a particular camp, it is really unnecessary to get into that discussion here (or probably ever). But you will, I promise, intimately understand what object-oriented programming is by the time you're done with this book, and you'll probably wonder why anyone would ever want to program in any other way.

But it takes more than a language to write a program; it takes a village. So who lives in the Objective-C village? Most object-oriented development environments consist of several parts:

- An object-oriented programming language
- A runtime environment
- A framework or library of objects and functions
- A suite of development tools

This is where, for many people, things start to cloud up. You may be saying, “You mean I have to master more than the language, and what is all this stuff about runtime environment and frameworks and libraries?” The answer is yes; but not to worry. I take you slowly through each part. The following sections cover each part of the Objective-C development environment.

Understanding programming languages

When you write a program, you write it as series of *statements.* Some of these statements are about data. You may allocate areas of memory to use for data in your program, as well as describe how data is structured. Other statements are really instructions for the computer to do something.

Here is an example of an Objective-C statement that adds together *b* and *c* and assigns the result to *a* (and you thought you’d never use all that algebra you picked up in school):

```
a = b + c;
```

Statements like these use operators (like + or –) or tell *modules* (functions or objects) to do something to, or with, the data. For now, think of functions or objects as simply a packaged series of statements that perform a task. It might help to think of operators and modules as words you use to create sentences (the statements) that tell the computer what to do. Chapters 4, 5, 6, and 7 cover operators, functions, objects, and modules in detail.

When most people want to learn how to program, they usually focus on the language. I want to program in C++, for example. Or C++ is a real dog; give me Java any day. People really do become passionate about languages, and believe me, it is best to keep out of the way when an unstoppable force meets an immovable object.

What you really should keep in mind, unless computer science is your life, is that what you want to master is how to create applications. What makes that easy or difficult is not just the language, but the application development tools available to you as well.

If you want to develop iOS and Mac OS X applications, Objective-C is the language you need to take full advantage of everything both platforms have to offer. Objective-C is the dominant programming language used by developers on both platforms, and Apple stands behind it.

Running your program in a runtime environment

One of the features of Objective-C is its runtime system. This is one of those things that gets linked into your program in Step 4 in the section "Creating a computer program," earlier in this chapter. It acts as a kind of operating system (like the Mac OS or iOS) for an individual Objective-C program. It is this runtime system that is responsible for making some of the very powerful features of Objective-C work.

Objective-C's runtime environment also makes it possible to use tools like Interface Builder (I explain Interface Builder in Chapters 17 and 18) to create user interfaces with a minimum of work (I'm all for that, and after you find out about Interface Builder, you will be, too).

Using frameworks and libraries

The framework you will use is called *Cocoa.* It came along with Objective-C when Apple acquired NeXT in 1996 (when it was called NeXTSTEP). I have worked in many development environments in my life, and Objective-C and Cocoa are hands down my favorites.

Cocoa enables you to write applications for Mac OS X, and a version of it enables you to write applications for iOS devices, such as the iPhone, iPad, and iPod. If the operating system does the heavy lifting vis-à-vis the hardware, the framework provides all the stuff you need to make your application an application. It provides support for windows and other user-interface items as well as many of the other things that are needed in most applications. When you use Cocoa to develop your application, all you need to do is add the application's specific functionality — the content as well as the controls and views that enable the user to access and use that content — to the Cocoa framework.

Framework or library?

What is the difference between a library and a framework? A library is a set of reusable functions or data structures that are yours to use. A framework, on the other hand, has an architecture or programming model that requires an application to be designed (divided into modules) in a certain way (application architecture) to use it. I like to think that whereas *you use a library, a framework uses you.*

Now, two excellent books explain the use of frameworks on the Mac and iPhone. One is *Mac Application Development For Dummies,* by Karl Kowalski. The other is *iPhone Application Development For Dummies,* 4th Edition, by Neal Goldstein (both published by John Wiley & Sons, Inc.).

Your suite of development tools

The main development tool you will use is Xcode. I explain Xcode in Chapter 2, and you use it throughout this book. In addition, when it's time to add a user-interface front end to your application, you'll make use of a tool within Xcode called Interface Builder. I talk a little about Interface Builder in Chapters 17 and 18, but again, pick up copies of *iPhone Application Development For Dummies* and *Mac Application Development For Dummies* to really understand the frameworks.

Using Xcode 4.4

You will use the Xcode 4.4 developer tools package that is available at the Mac App Store. This version of Xcode requires Mac OS X 10.7.3 (Lion) or later, and it is a major improvement over previous versions of Xcode.

Using Objective-C Version 2.0

You will find out about Version 2.0 of the Objective-C language, which was released with Mac OS X 10.5, and yes, you should care. I provide you examples of some of the very useful features of Objective-C 2.0, such as declared properties, fast enumeration, and Automatic Reference Counting, which greatly simplify memory management. All these features are available in the latest versions of Mac OS X and iOS. If possible, I'll also indicate some work-arounds if you need to write applications that run under earlier versions of the OS, but in general, writing applications that run under earlier versions of the OS will be up to you.