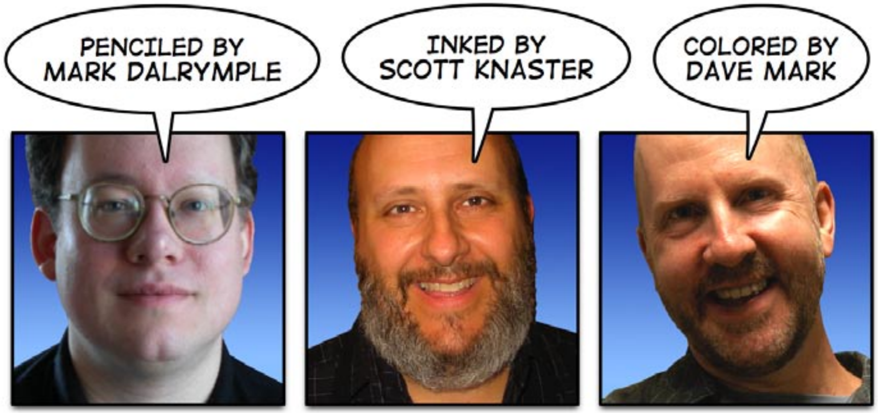
eBook 1.0 – Check for Updates

$14.95 USD

Learn Objective-C on the Macintosh



SpiderWorks

For more great books, visit us online at
http://www.spiderworks.com

Contents

eBook Reading Tips

We recommend using Adobe Acrobat or the free Adobe Reader to view this eBook. Apple Preview and other third-party PDF viewers may also work, but many of them do not support the latest PDF features. For best results, use Adobe Acrobat/Reader.

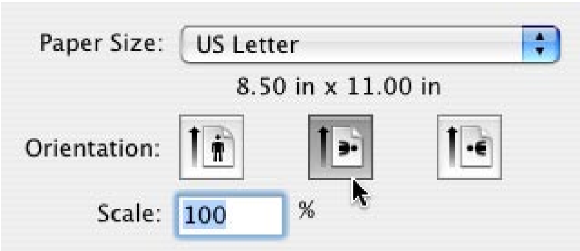
To view this PDF onscreen like a book using Adobe Reader, select "Facing Pages" as your Page Layout preference (View Menu / Page Layout / Facing). To display the eBook full-screen using Adobe Reader, select Full Screen from the View Menu.

To jump directly to a specific page, click on a topic from either the Table of Contents pages or from the PDF Bookmarks. In Adobe Reader, the PDF Bookmarks can be accessed by clicking on the Bookmarks tab on the left side of the screen. In Apple Preview, the PDF Bookmarks are located in a drawer (Command-T to open).

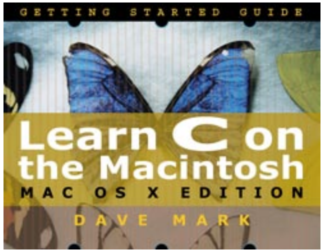
If your mouse cursor turns into a hand icon when hovering over some text, that indicates the text is a hyperlink. Table of Contents links jump to a specific page within the ebook when clicked. Text links that begin with "http" will attempt to access an external web site when clicked (requires an Internet connection).

Printing the eBook

Since SpiderWorks eBooks utilize a unique horizontal page layout for optimal on-screen viewing, you should choose the "Landscape" setting (in Page Setup) to print pages sideways on standard 8.5" x 11" paper. If the Orientation option does not label the choices as "Portrait" and "Landscape", then choose the visual icon of the letter "A" or person's head printed sideways on the page (see example below).



Also Available...



New to programming? Step through the basics with the first book in SpiderWorks' Mac programming series, *Learn C on the Macintosh (Mac OS X Edition)*, by Dave Mark. Perfect for beginners learning to program with Xcode! Download the free preview and order online at http://www.spiderworks.com/

Publisher Credits

Cover Design: **Mark Dame** and **Dave Wooldridge**

Cover Illustration: **Russell Tate** (iStockphoto.com)

Interior Page Design: **Robin Williams**

PDF Production: **Dave Wooldridge**

Written by...

Mark Dalrymple has been a Mac developer since 1985 and a Unix programmer since 1990. Over the years he has worked on projects ranging from cross-platform development toolkits, high-performance web server software, medical applications, and video products for Hollywood. He is the co-author of *Core Mac OS X and Unix Programming* and *Advanced Mac OS X Programming*.

Scott Knaster is a legendary Mac hacker and author of such best-selling books as *Hacking Mac OS X Tiger* and *Macintosh Programming Secrets*. His book *How to Write Macintosh Software* was required reading for Mac programmers for more than a decade.

Edited by...

Dave Mark, a long-time Mac developer and author of numerous books on Macintosh development including *Learn C on the Macintosh*, *The Macintosh Programming Primer* series, and *Ultimate Mac Programming*. Dave has also served as Editor-in-Chief and contributing writer for *MacTech Magazine*.

Downloading the Companion Files

The collection of companion project files and examples from this book is contained in a download called *LearnObjC_Projects.zip*, which can be downloaded from the SpiderWorks Customer Download Center at http://www.spiderworks.com/extras/. To login, you will need your SpiderWorks Username and Password that were listed in your order confirmation e-mail.

Requirements

This book assumes that you are running Mac OS X 10.4 or later. To utilize the companion source code files, you should have Apple's free Developer Tools installed.

Installation

Once you have downloaded and decompressed *LearnObjC_Projects.zip*, you will see a directory called *Learn ObjC Projects*. Nested inside that directory are further sub-directories labeled for the chapter to which the example files apply. Not all chapters have example files in the *Learn ObjC Projects* collection. Move the *Learn ObjC Projects* directory to a convenient location on your hard disk from which you can open the files in Xcode and/or other applicable software applications.

Chapter 3
Introduction to Object-Oriented Programming

If you've been using and programming computers for any length of time, you've probably heard the term *object-oriented programming* more than once. Object-oriented programming, frequently shortened to its initials, *OOP*, is a programming technique originally developed for writing simulation programs. OOP soon caught on with developers of other kinds of software, such as those involving graphical user interfaces. Before long, OOP became a major industry buzzword. It promised to be the magical silver bullet that would make programming simple and joyous.

Of course, nothing can live up to that kind of hype. Like most pursuits, OOP requires study and practice to gain proficiency, but it truly does make some kinds of programming tasks easier, and in some cases, even fun. In this book we'll be talking about OOP a lot, mainly because Cocoa is based on OOP concepts, and Objective-C is a language that is designed to be object-oriented.

So what is OOP? OOP is a way of constructing software composed of objects. Objects are like little machines living inside your computer and talking to each other in order to get work done. In this chapter, we'll look at some basic OOP concepts. After that, we'll examine the style of programming that leads to OOP, describing the motivation behind some OOP features. We'll wrap up with a thorough description of the mechanics of OOP.

> Like many "new" technologies, the roots of OOP stretch way back into the mists of time. OOP evolved from Simula in the 1960s, Smalltalk in the 1970s, Clascal in the 1980s, and other related languages. Modern languages such as C++, Java, Python and, of course, Objective-C draw inspiration from these older languages.

As we dive into OOP, stick a Babel fish in your ear and be prepared to encounter some strange terminology along the way. OOP comes with a lot of fancy-sounding lingo that makes it sound more mysterious and difficult than it actually is. You might even think that computer scientists create long, impressive sounding words to show everyone how smart they are – but of course, they don't all do that. Well, don't worry. We'll explain each term as we encounter it.

Before we get into OOP itself, let's take a look at a key concept of OOP: indirection.

It's all Indirection

An old saying in programming goes something like this: "There is no problem in computer science that can't be solved by adding another level of indirection." Indirection is a fancy word with a simple meaning: instead of using a value directly in your code, use a pointer to the value. Here's a real-word example: you might not know the phone number of your favorite pizza place, but you know that you can look in the phone book to find it. Using the phone book like this is a form of indirection.

Indirection can also mean that you ask another person to do something rather than doing it yourself. Let's say you have a box of books to return to your friend Andrew who lives across town. You know that your next-door neighbor is going to visit Andrew tonight. Rather than driving across town, dropping off the books, and driving back, you ask your friendly neighbor to deliver the box. This is another kind of indirection: you have someone else do the work instead of doing it yourself.

In programming, you can take indirection to multiple levels, writing code that consults other code, which accesses yet another level of code. You've probably had the experience of calling a technical support line. You explain your problem to the support person, who then directs you to the specific department that can handle your problem. The person there then directs you the second-level technician with the skills to help you out. And if you're like us, at this point you find out you called the wrong number and you have to be transferred to some other department for help. This runaround is a form of indirection. Luckily, computers have infinite patience and can handle being sent from place to place to place looking for an answer.

Variables

You might be surprised to find out that you have already used indirection in your programs. The humble variable is a real-world use of indirection. Consider this small program that prints the numbers from 1 to 5. You can find this program in the *Learn ObjC Projects* folder, in *03.01 - Count-1*.

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

int main (int argc, const char *argv[])
{
  NSLog (@"The numbers from 1 to 5:");

  int i;
  for (i = 1; i <= 5; i++) {
    NSLog (@"%d\n", i);
  }

  return (0);

} // main
```

Count-1 has a **for** loop that runs 5 times, using **NSLog()** to display the value of **i** each time around. When you run this program, you see output like this:

```
2006-09-01 13:14:40.513 Count-1[2233] The
 numbers from 1 to 5:
2006-09-01 13:14:40.514 Count-1[2233] 1
2006-09-01 13:14:40.514 Count-1[2233] 2
2006-09-01 13:14:40.514 Count-1[2233] 3
2006-09-01 13:14:40.514 Count-1[2233] 4
2006-09-01 13:14:40.514 Count-1[2233] 5
```

Now suppose you want to upgrade your program to print the numbers from 1 to 10. You have to edit your code in two places, and then rebuild the program. (This version is in the folder *03.02 - Count-2*).

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

int main (int argc, const char * argv[])
{
  NSLog (@"The numbers from 1 to 10:");

  int i;
  for (i = 1; i <= 10; i++) {
    NSLog (@"%d\n", i);
  }

  return (0);

} // main
```

Count-2 produces this output:

```
2006-09-01 13:21:52.435 Count-2[2290] The
 numbers from 1 to 10:
2006-09-01 13:21:52.435 Count-2[2290] 1
2006-09-01 13:21:52.435 Count-2[2290] 2
2006-09-01 13:21:52.435 Count-2[2290] 3
2006-09-01 13:21:52.436 Count-2[2290] 4
2006-09-01 13:21:52.436 Count-2[2290] 5
2006-09-01 13:21:52.436 Count-2[2290] 6
2006-09-01 13:21:52.436 Count-2[2290] 7
2006-09-01 13:21:52.436 Count-2[2290] 8
2006-09-01 13:21:52.436 Count-2[2290] 9
2006-09-01 13:21:52.436 Count-2[2290] 10
```

Modifying the program in this way is obviously not a very tricky change to make: you can do it with a simple search-and-replace, and there are only two places that have to be changed. However, it would be a lot trickier to do a similar search-and-replace in a larger program, consisting of, say, tens of thousands of lines of code. We would have to be careful about simply replacing 5 with 10: no doubt, there would be other instances of the number 5 that aren't related to this and so shouldn't be changed to 10.

This is what variables are for. Rather than sticking the upper loop value (5 or 10) directly in the code, we can solve this problem by putting the number in a variable, thus adding a layer of indirection. When you add the variable, instead of saying, "go through the loop 5 times", you're telling the program "go look in this variable named **`count`** – it will tell you how many times to run the loop". Now the program, Count-3, looks like this:

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

int main (int argc, const char * argv[])
{
  int count = 5;

  NSLog (@"The numbers from 1 to %d:", count);

  int i;
  for (i = 1; i <= count; i++) {
    NSLog (@"%d\n", i);
  }

  return (0);

} // main
```

The program's output should be unsurprising:

```
2006-09-01 13:32:43.667 Count-3[2318] The
 numbers from 1 to 5:
2006-09-01 13:32:43.677 Count-3[2318] 1
2006-09-01 13:32:43.678 Count-3[2318] 2
2006-09-01 13:32:43.678 Count-3[2318] 3
2006-09-01 13:32:43.678 Count-3[2318] 4
2006-09-01 13:32:43.678 Count-3[2318] 5
```

The **NSLog()** time stamp and other information take up a lot of space so, for clarity, we'll leave it out of future listings.

If you want to print the numbers from 1 to 100, you just have to touch the code in one obvious place:

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

int main (int argc, const char * argv[])
{
  int count = 100;

  NSLog (@"The numbers from 1 to %d:", count);

  int i;
  for (i = 1; i <= count; i++) {
    NSLog (@"%d\n", i);
  }

  return (0);

} // main
```

By adding a variable, our code is now much cleaner and easier to extend. This is especially true when other programmers need to change the code. To change the loop values, they won't have to scrutinize every use of the number 5 to see if they need to modify it. Instead, they can just change the **count** variable to get the result they want.

Filenames

Files provide another example of indirection. Consider Word-Length-1, a program that prints a list of words along with their lengths. This vital program is the key technology for your new internet startup, Length-o-words.com. This program is in the *03.04 - Word-Length-1* folder. Here's the listing:

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

int main (int argc, const char * argv[])
{
  const char *words[4] = { "aardvark",
                 "abacus", "allude", "zygote" };
  int wordCount = 4;

  int i;
  for (i = 0; i < wordCount; i++) {
    NSLog (@"%s is %d characters long",
        words[i], strlen(words[i]));
  }

  return (0);

} // main
```

The **for** loop determines which word in the **words** array is being processed at any time. The **NSLog()** function inside the loop prints out the word using the **%s** format specifier. We use **%s** because **words** is an array of C strings rather than **@"NSString"** objects. The **%d** format specifier takes the integer value of the **`strlen()`** function, which calculates the length of the string, and prints it out along with the word itself.

When you run Word-Length-1, you see informative output like this:

```
aardvark is 8 characters long
abacus is 6 characters long
allude is 6 characters long
zygote is 6 characters long
```

Remember that we're leaving out the time stamp and process ID that **`NSLog()`** adds to its output.

Now suppose the venture capitalists investing in Length-o-words.com want you to use a different set of words. They've scrutinized your business plan and have concluded that you can sell to a broader market if you use the names of country music stars.

Because we stored the words directly in the program, we have to edit the source, replacing the original word list with the new names. When we edit, we have to be careful with the punctuation, such as the quotes in Joe-Bob's name and the commas between entries. Here is the updated program, which can be found in the *03.05 - Word-Length-2* folder:

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

int main (int argc, const char * argv[])
{
  const char *words[4]
    = { "Joe-Bob \"Handyman\" Brown",
      "Jacksonville \"Sly\" Murphy",
      "Shinara Bain",
      "George \"Guitar\" Books" };
  int wordCount = 4;

  int i;
  for (i = 0; i < wordCount; i++) {
    NSLog (@"%s is %d characters long",
        words[i], strlen(words[i]));
  }

  return (0);

} // main
```

Because we were careful with the surgery, the program still works as we expect.

```
Joe-Bob "Handyman" Brown is 24 characters long
Jacksonville "Sly" Murphy is 25 characters long
Shinara Bain is 12 characters long
George "Guitar" Books is 21 characters long
```

Making this change required entirely too much work: we had to edit *main.m*, fix any typos, and then rebuild the program. If the program runs on a web site, we then have to re-test and redeploy the program in order to upgrade to Word-Length-2.

Another way to construct this program is to move the names completely out of the code and put them all into a text file, one name on each line. Let's all say it together: this is indirection. Rather than putting the names directly in the source code, the program looks for the names elsewhere. The program reads a list of names from a text file, then proceeds to print them out, along with their lengths. The project files for this new program live in the *03.06 - Word-Length-3* folder, and the code looks like this:

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

int main (int argc, const char * argv[])
{
  FILE *wordFile
           = fopen ("/tmp/words.txt", "r");
  char word[100];

  while (fgets(word, 100, wordFile)) {
    // strip off the trailing \n
    word[strlen(word) - 1] = ‘\0’;

    NSLog (@"%s is %d characters long",
        word, strlen(word));
  }

  fclose (wordFile);

  return (0);

} // main
```

Let's stroll through Word-Length-3 and see what it's doing. First, **`fopen()`** opens the *words.txt* file for reading. Next, **`fgets()`** reads a line of text from the file and places it into **`word`**. The **`fgets()`** call preserves the newline character that separates each line, but we really don't want it – if we leave it, it will be counted as a character in the word. To fix this, we replace the newline character with a zero, which indicates the end of the string. Finally, we use our old friend **`NSLog()`** to print out the word and its length.

> Take a look at the path name we used with **`fopen()`**. It's */tmp/words.txt*. This means that *words.txt* is a file that lives in the */tmp* directory, the unix "temporary" directory, which gets emptied when the computer reboots. You can use */tmp* to store scratch files that you want to mess around with, but you really don't care about keeping. For a real live program, you'd put your file in a more permanent location, such as the home directory.

Before you run the program, use your text editor to create the file *words.txt* in the */tmp* directory. Type the following names into the file:

```
Joe-Bob "Handyman" Brown
Jacksonville "Sly" Murphy
Shinara Bain
George "Guitar" Books
```

If you prefer, instead of typing the names, you can copy *words.txt* from the *03.06 - Word-Length-3* directory into */tmp*. To see */tmp* in the Finder, choose Go > Go to Folder.

When you run Word-Length-3, the program's output looks just as it did before:

```
Joe-Bob "Handyman" Brown is 24 characters long
Jacksonville "Sly" Murphy is 25 characters long
Shinara Bain is 12 characters long
George "Guitar" Books is 21 characters long
```

Word-Length-3 is a shining example of indirection. Rather than coding the words directly into your program, you're instead saying, "Go look in */tmp/words.txt* to get the words". With this scheme, we can change the set of words any time we want, just by editing this text file, without having to change the program. Go ahead and try it out: add a couple of words to your *words.txt* file and re-run the program. We'll wait for you here.

This approach is better, because text files are easier to edit and far less fragile than source code. You can get your non-programmer friends to use TextEdit to do the editing. Your marketing staff can keep the list of words up to date, which frees you to work on more interesting tasks.

As you know, people always come along with new ideas for upgrading or enhancing a program. Maybe your investors have decided that counting the length of cooking terms is the new path to profit. Now that your program looks at a file for its data, you can change the set of words all you want without ever having to touch the code.

Despite great advances in indirection, Word-Length-3 is still rather fragile, because it insists on using a full path name to the words file. And that file itself is in a precarious position: if the computer reboots, */tmp/words.txt* vanishes. Also, if someone else is using the program on your machine with their own */tmp/words.txt* file, they could accidentally stomp on your copy. You could edit the program each time to use a different path, but we already know that that's no fun, so let's add another indirection trick to make our lives easier.

Instead of using the technique "Go look in */tmp/words.txt* to get the words", we'll change it to "Go look at the first launch parameter of the program to figure out the location of the words file." Here is the program (it's Word-Length-4, which can be found in the *03.07 - Word-Length-4* folder). It uses a command-line parameter to specify the file name. The changes we made to Word-

Length-3 are highlighted:

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

int main (int argc, const char * argv[])
{
    if (argc == 1) {
        NSLog (@"you need to provide a file name");
        return (1);
    }

    FILE *wordFile = fopen (argv[1], "r");
    char word[100];

    while (fgets(word, 100, wordFile)) {
        // strip off the trailing \n
        word[strlen(word) - 1] = '\0';

        NSLog (@"%s is %d characters long",
            word, strlen(word));
    }

    fclose (wordFile);

    return (0);

} // main
```

The loop that processes the file is the same as in Word-Length-3, but the code that sets it up is new and improved. The **`if`** statement verifies that the user supplied a path name as a launch parameter. The code consults the **`argc`** parameter to **`main()`**, which holds the number of launch parameters. Because the program name is always passed as a launch parameter, **`argc`** is always 1 or greater. If the user doesn't pass a file path, the value of **`argc`** is 1, and we have no file to read, so we print an error message and stop the program.

If the user was thoughtful and provided a file path, **`argc`** is greater than 1. We then look in the **`argv`** array to see what that file path is. **`argv[1]`** contains the file name the user has given us. (In case you're curious, the **`argv[0]`** parameter holds the name of the program.)

If you're running the program in Terminal, it's easy to specify the name of the file on the command line, like so:

```
$ ./Word-Length /tmp/words.txt
Joe-Bob "Handyman" Brown is 24 characters long
Jacksonville "Sly" Murphy is 25 characters long
Shinara Bain is 12 characters long
George "Guitar" Books is 21 characters long
```

If you're editing the program along with us in Xcode, supplying a file path as you run it is a little more complicated. Launch arguments, also called command-line parameters, are a little trickier to control from Xcode than from Terminal. Here's what you need to do to change the launch arguments:

First, in the Xcode files list, expand Executables and double-click the program (Word-Length), as shown in Figure 3.1.

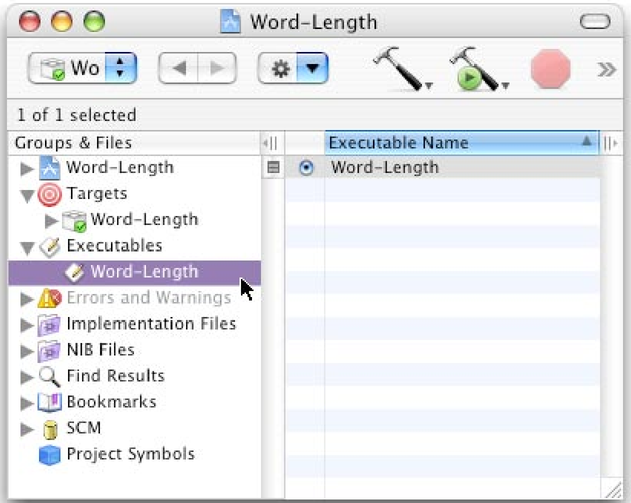


Figure 3.1. Expand Executables and Double-Click the Program

Then, as shown in Figure 3.2, click the Arguments tab, then click the plus sign and type the launch argument – in this case, the path to the *words.txt* file.

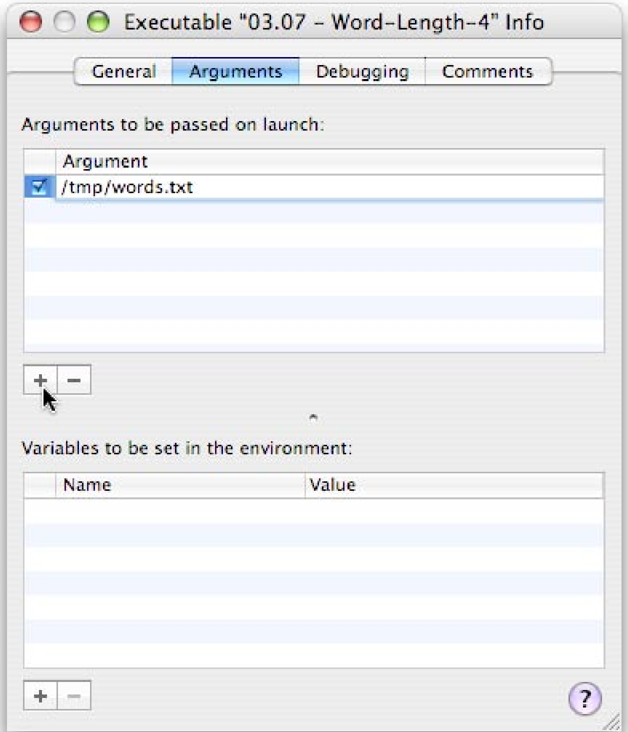


Figure 3.2. Add the Launch Argument

Now, when you run the program, Xcode passes your launch argument into Word-Length-4's **`argv`** array. Figure 3.3 shows what you'll see when you run the program:

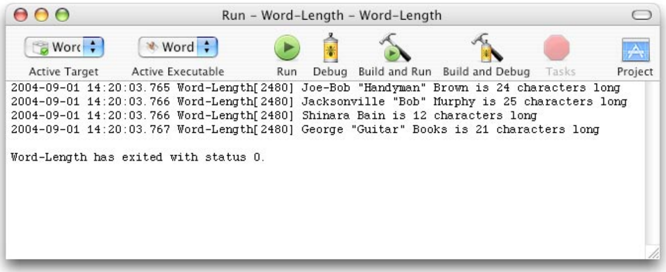


Figure 3.3. Running Word-Length-3

By supplying arguments at runtime, anybody can use your program to get the length of *any* set of words they want to. Users can change the data without changing the code, just as nature intended. This is the essence of indirection, telling us where to get the data we need.

Object Oriented Programming and Indirection

Object-Oriented Programming is all about indirection. OOP uses indirection for accessing data, just as we did in the previous examples by employing variables, files and arguments. The real revolution of OOP is that it uses indirection for calling *code*. Rather than calling a function directly, you end up calling it indirectly.

Now that you know that, you're an expert in OOP. Everything else is a side-effect of this indirection.

Procedural Orientation

To complete your appreciation of the flexibility of OOP, we'll take a quick look at procedural programming, so you can get an idea of the kinds of problems that OOP was created to solve. Procedural programming has been around a long, long time, since just after the invention of dirt. Procedural programming is the kind typically taught in introductory programming books and classes. Most programming in languages like BASIC, C, Tcl, and Perl is procedural.

In procedural programs, data is typically kept in simple structures, such as C **`structs`**. There are also more complex data structures such as linked lists and trees. When you call a function, you pass the data to the function, and it manipulates the data. Functions are the center of the procedural programming experience: you decide which functions you want to use, and then you call the functions, passing the data they need.

Consider a program that draws a bunch of geometric shapes on the screen. Thanks to the magic of computers, you can do more than consider it – you'll find the source code to this program in the *03.08 - Shapes-Procedural* folder. For simplicity's sake, the Shapes-Procedural program doesn't actually draw shapes on the screen, it just quaintly prints out some shape-related text.

Shapes-Procedural uses plain C and the procedural programming style. The code starts out by defining some constants and a structure.

First is an enumeration that specifies the different kinds of shapes that can be drawn: circle, square, and something vaguely egg-shaped:

```
typedef enum {
  kCircle,
  kRectangle,
  kOblateSpheroid
} ShapeType;
```

Next is an **`enum`** that defines the colors that can be used to draw the shape:

```
typedef enum {
  kRedColor,
  kGreenColor,
  kBlueColor
} ShapeColor;
```

Then there's a structure that describes a rectangle, which specifies the area on the screen where the shape will be drawn:

```
typedef struct {
  int x, y, width, height;
} ShapeRect;
```

Finally, we have a structure that pulls all these things together to describe a shape:

```
typedef struct {
  ShapeType type;
  ShapeColor fillColor;
  ShapeRect bounds;
} Shape;
```

Next up in our example, **`main()`** declares an array of shapes we're going to draw. After declaring the array, each shape structure in the array is initialized by assigning its fields. The following code gives us a red circle, a green rectangle, and a blue spheroid.

```
int main (int argc, const char * argv[])
{
  Shape shapes[3];

  ShapeRect rect0 = { 0, 0, 10, 30 };
  shapes[0].type = kCircle;
  shapes[0].fillColor = kRedColor;
  shapes[0].bounds = rect0;

  ShapeRect rect1 = { 30, 40, 50, 60 };
  shapes[1].type = kRectangle;
  shapes[1].fillColor = kGreenColor;
  shapes[1].bounds = rect1;

  ShapeRect rect2 = { 15, 18, 37, 29 };
  shapes[2].type = kOblateSpheroid;
  shapes[2].fillColor = kBlueColor;
  shapes[2].bounds = rect2;

  drawShapes (shapes, 3);
```

```
    return (0);

  } // main
```

The rectangles in **`main()`** are declared using a handy little C trick: when you declare a variable that's a structure, you can initialize all the elements of that structure at once:

```
    ShapeRect rect0 = { 0, 0, 10, 30 };
```

The structure elements get values in the order they're declared. Recall that **`ShapeRect`** is declared like this:

```
  typedef struct {
    int x, y, width, height;
  } ShapeRect;
```

The assignment to **`rect0`** above means that **`rect0.x`** and **`rect0.y`** will both have the value zero, **`rect0.width`** will be 10, and **`rect0.height`** will be 30.

This technique lets you reduce the amount of typing in your program without sacrificing readability.

After initializing the **`shapes`** array, **`main()`** calls the **`drawShapes()`** function to "draw" the shapes.

`drawShapes()` has a loop that inspects each **`Shape`** structure in the array. A **`switch`** statement looks at the **`type`** field of the structure and chooses a function that draws the shape. The program calls the appropriate drawing function, passing parameters for the screen area and color to use for drawing. Check it out:

```
void drawShapes (Shape shapes[], int count)
{
  int i;

  for (i = 0; i < count; i++) {

    switch (shapes[i].type) {

    case kCircle:
      drawCircle (shapes[i].bounds,
            shapes[i].fillColor);
      break;

    case kRectangle:
      drawRectangle (shapes[i].bounds,
              shapes[i].fillColor);
      break;

    case kOblateSpheroid:
      drawEgg (shapes[i].bounds,
           shapes[i].fillColor);
      break;
    }
  }
} // drawShapes;
```

Here is the code for **drawCircle()**, which just prints out the bounding rectangle and the color passed to it:

```
void drawCircle (ShapeRect bounds,
         ShapeColor fillColor)
{
  NSLog (@"drawing a circle at (%d %d %d %d)
     in %@",
     bounds.x, bounds.y,
     bounds.width, bounds.height,
     colorName(fillColor));

} // drawCircle
```

The **colorName()** function called inside **NSLog()** simply does a switch on the passed-in color value and returns a literal **NSString** such as **@"red"** or **@"blue"**.

The other draw functions are almost identical to **drawCircle**, except that they "draw" a rectangle or an egg.

Here is the output of Shapes-Procedural (minus the time stamp and other information added by **NSLog()**):

```
drawing a circle at (0 0 10 30) in red
drawing a rectangle at (30 40 50 60) in green
drawing an egg at (15 18 37 29) in blue
```

This all seems pretty simple and straightforward, right? When you use procedural programming, you spend your time connecting data with the functions designed to deal with that type of data. You have to be careful to use the right function for each data type: for example, you must call **drawRectangle()** for a shape of type **kRectangle**. It's disappointingly easy to pass a rectangle to a function meant to work with circles.

Another problem with coding like this is that it can make extending and maintaining the program difficult. To illustrate, let's enhance Shapes-Procedural to add a new kind of shape: a triangle. You can find the modified program in the *03.09 - Shapes-Procedural-2* project. We have to modify the program in at least four different places to accomplish this task.

First, we'll add a **kTriangle** constant to the **ShapeType enum**:

```
typedef enum {
  kCircle,
  kRectangle,
  kOblateSpheroid,
  kTriangle
} ShapeType;
```

Then, we'll implement a **drawTriangle()** function that looks just like its siblings:

```
void drawTriangle (ShapeRect bounds,
          ShapeColor fillColor)
{
  NSLog (@"drawing triangle at (%d %d %d %d)
    in %@",
    bounds.x, bounds.y,
    bounds.width, bounds.height,
    colorName(fillColor));

} // drawTriangle
```

Next, we'll add a new **`case`** to the **`switch`** statement in **`drawShapes().`** This will test for **`kTriangle`** and will call **`drawTriangle()`** if appropriate:

```
void drawShapes (Shape shapes[], int count)
{
  int i;

  for (i = 0; i < count; i++) {

    switch (shapes[i].type) {

    case kCircle:
      drawCircle (shapes[i].bounds,
            shapes[i].fillColor);
      break;

    case kRectangle:
      drawRectangle (shapes[i].bounds,
              shapes[i].fillColor);
      break;

    case kOblateSpheroid:
      drawEgg (shapes[i].bounds,
           shapes[i].fillColor);
      break;

    case kTriangle:
      drawTriangle (shapes[i].bounds,
              shapes[i].fillColor);
      break;
    }
  }
} // drawShapes
```

Finally, we'll add a triangle to the **`shapes`** array:

```
int main (int argc, const char * argv[])
{
  Shape shapes[4];

  ShapeRect rect0 = { 0, 0, 10, 30 };
  shapes[0].type = kCircle;
  shapes[0].fillColor = kRedColor;
  shapes[0].bounds = rect0;

  ShapeRect rect1 = { 30, 40, 50, 60 };
  shapes[1].type = kRectangle;
  shapes[1].fillColor = kGreenColor;
  shapes[1].bounds = rect1;

  ShapeRect rect2 = { 15, 18, 37, 29 };
  shapes[2].type = kOblateSpheroid;
  shapes[2].fillColor = kBlueColor;
  shapes[2].bounds = rect2;
```

```
    ShapeRect rect3 = { 47, 32, 80, 50 };
    shapes[3].type = kTriangle;
    shapes[3].fillColor = kRedColor;
    shapes[3].bounds = rect3;

    drawShapes (shapes, 4);

    return (0);

} // main
```

OK, let's take a look at Shapes-Procedural-2 in action:

```
drawing a circle at (0 0 10 30) in red
drawing a rectangle at (30 40 50 60) in green
drawing an egg at (15 18 37 29) in blue
drawing a triangle at (47 32 80 50) in red
```

Adding support for triangles wasn't too bad. But our little program only does one kind of action: drawing shapes. The more complex the program, the trickier it is to extend. For example, let's say the program does more messing around with shapes, such as computing their area and determining if the mouse pointer lies within them. In that case, you'll have to modify every function that performs an action on shapes, touching code that has been working perfectly and possibly introducing errors.

Here's another scenario that's fraught with peril: adding a new shape that needs more information to describe it. For example, a rounded rectangle needs to know its bounding rectangle as well as the radius of the rounded corners. To support rounded rectangles, you could add a radius field to the **`Shape`** structure, which is a waste of space, because the field won't be used by other shapes, or you could use a C union to overlay different data layouts in the same structure, which complicates things by making all shapes dig into the union to get to their interesting data.

OOP addresses these problems elegantly. As we teach our program to use OOP, we'll see how OOP handles the first problem, modifying already-working code to add new kinds of shapes.

Object Orientation

Procedural programs are based on functions. The data orbits around the functions. Object orientation reverses this point of view, placing a program's data at the center, with the functions orbiting around the data. Instead of focusing on functions in your programs, you concentrate on the data.

That sounds interesting, but how does it work? In OOP, data contains references to the code that operates on it, using indirection. Rather than telling the **`drawRectangle()`** function "Go draw a rectangle using this shape structure", you instead ask a rectangle to "Go draw yourself". (Gosh, that sounds rude, but it's really not.) Through the magic of indirection, the rectangle's data knows how to find the function that will perform the drawing.

So what exactly is an object? It's nothing more than a fancy C struct that has the ability to find code it's associated with, usually via a function pointer. Figure 3.4 shows four Shape objects: two squares, a circle, and a spheroid. Each object is able to find a function to do its drawing.

Each object has its own **`draw()`** function that knows how to draw its specific shape. For example, a circle object's **`draw()`** knows to draw a circle. A rectangle's **`draw()`** knows to draw four straight lines that form a rectangle.

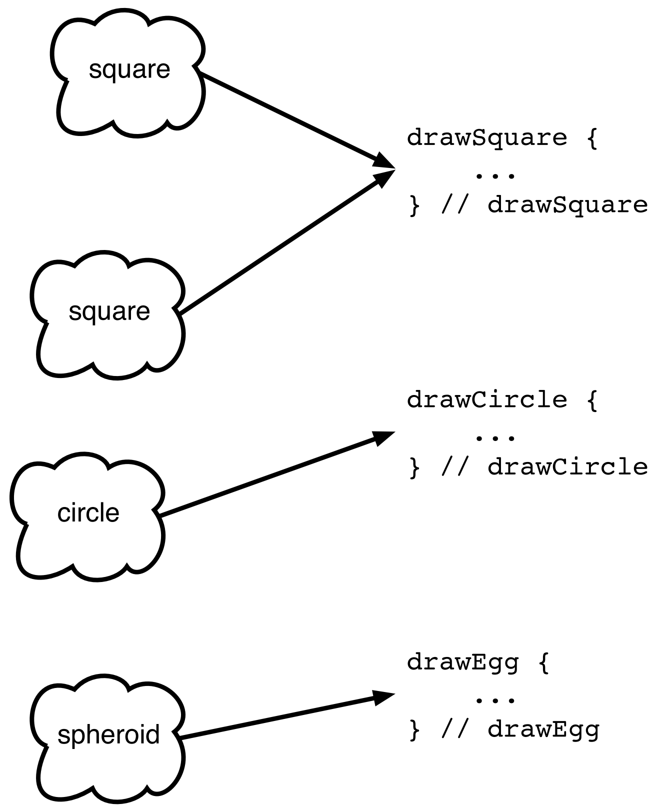


Figure 3.4. Basic Objects

The program Shapes-Object (available at *03.10 - Shapes-Object*) does the same stuff as Shapes-Procedural, but uses Objective-C's object-oriented features to do it. Here's **`drawShapes()`** from Shapes-Object:

```
void drawShapes (id shapes[], int count)
{
  int i;

  for (i = 0; i < count; i++) {
    id shape = shapes[i];
    [shape draw];
  }

} // drawShapes;
```

This function contains a loop that looks at each shape in the array. In the loop, the program tells the shape to draw itself.

Notice the differences between this version of **`drawShapes()`** and the original. For one thing, this one is a lot shorter! The code doesn't have to ask each individual shape what kind it is.

Another change is **`shapes[]`**, the first argument to the function: it's now an array of **`id`**s. What is an **`id`**? Is it a psychological term referring to the part of the mind in which innate instinctive impulses and primary processes are manifest? Not in this case: it stands for **identifier**, and it's pronounced "eye-dee". An **`id`** is a generic type that's used to refer to any kind of object. Remember that an object is just a C struct with some code attached, so **`id`**s are actually pointers to these structures; in this case, they're structures that make various kinds of shapes.

The third change to **`drawShapes()`** is the body of the loop:

```
id shape = shapes[i];
[shape draw];
```

The first line looks like ordinary C. The code gets an **`id`** – that is, a pointer to an object – from the **`shapes`** array and sticks it into the variable named **`shape`**, which has the type **`id`**. This is just a pointer assignment: it doesn't actually copy the entire contents of the shape. Take a look at Figure 3.5 to see the various shapes available in Shapes-Object. **`shapes[0]`** is a pointer to the red circle, **`shapes[1]`** is a pointer to a green rectangle, and **`shapes[2]`** is a pointer to a blue egg.

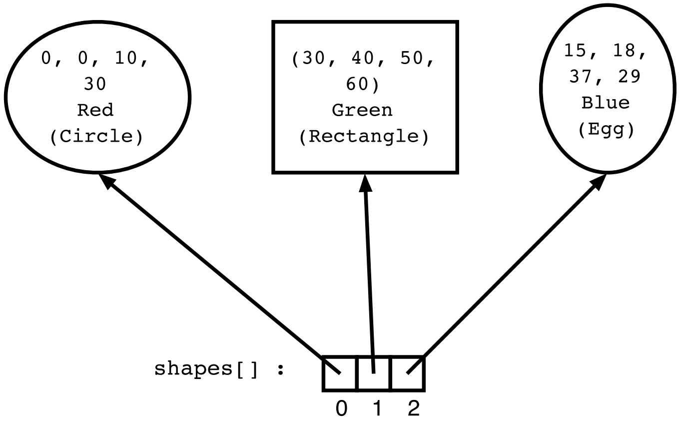


Figure 3.5. The Shapes Array

Now we've come to the last line of code in the function:

```
[shape draw];
```

This is seriously weird. What's going on? We know that C uses square brackets to refer to array elements, but it doesn't look like we're doing anything with arrays here. In Objective-C, square brackets have an additional meaning: they're used to tell an object what to do. Inside the square brackets, the first item is an object, and the rest is an action that you want the object to perform. In this case, we're telling an object named **shape** to perform the action **draw**. If **shape** is a circle, a circle is drawn. If **shape** is a rectangle, we'll get a rectangle.

In Objective-C, telling an object to do an action is called **sending a message**. The code **[shape draw]** sends the message **draw** to the object **shape**. One way to pronounce **[shape draw]** is "send **draw** to **shape**." How the shape actually does the drawing is up to the **shape**'s implementation.

When you send a message to an object, how does the necessary code get called? This happens with the assistance of behind-the-scenes helpers called **classes**.

Take a look at Figure 3.6 on the next page, please. The left side of the figure shows that this is the circle object at index zero of the **shapes** array, last seen in Figure 3.5. The object has a pointer to its class. The class is a structure that tells how to be an object of its kind. In Figure 3.6, the Circle class has a pointer to code for drawing circles, for calculating the area of circles, and other stuff required in order to be a good Circle citizen.

What's the point of having class objects? Wouldn't it be simpler just to have each object point directly to its code? Indeed it would be simpler, and there are some OOP systems that do just that. But having class objects is a great advantage: if you change the class at runtime, all objects of that class automatically pick up the changes. We'll discuss this more in later chapters.

Figure 3.7 (next page) shows how the **draw** message ends up calling the right function for the circle object.

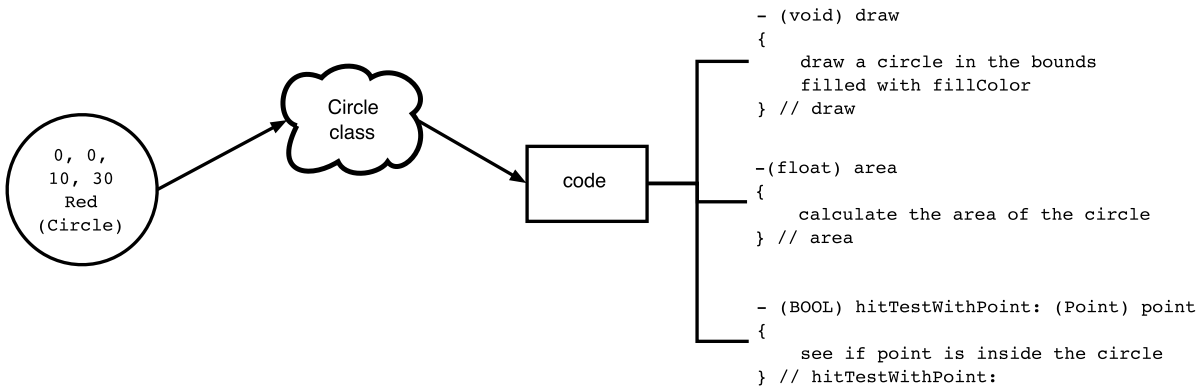


Figure 3.6. A Circle and its Class

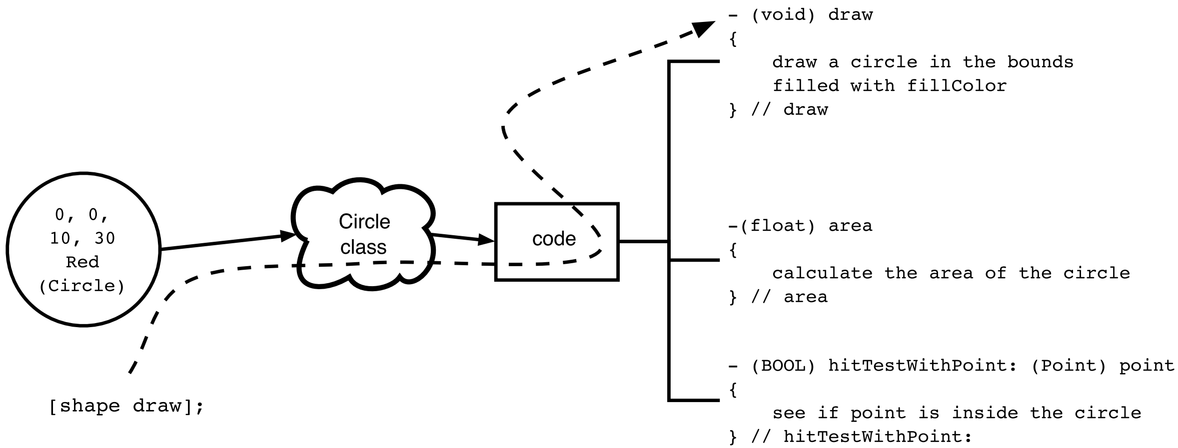


Figure 3.7. Circle finds its draw code

Here are the steps:

1. The object that is the target of the message (the red circle in this case) is consulted to see what its class is.
2. The class looks through its code and finds out where the **`draw`** function is.
3. Once it's found, the function that draws circles is executed.

Figure 3.8 shows what happens when you call **`[shape draw]`** on the second shape in the array, which is the green rectangle. The steps used are nearly identical:

1. The target object of the message (the green rectangle) is consulted to see what its class is.
2. The rectangle class checks its pile of code and gets the address of the **`draw`** function.
3. Objective-C runs the code that draws a rectangle.

This is some very cool indirection in action! In the procedural version of this program, we had to write code that determined which function to call. Now, that decision is made behind the scenes by Objective-C, as it asks the objects which class they belong to. This reduces

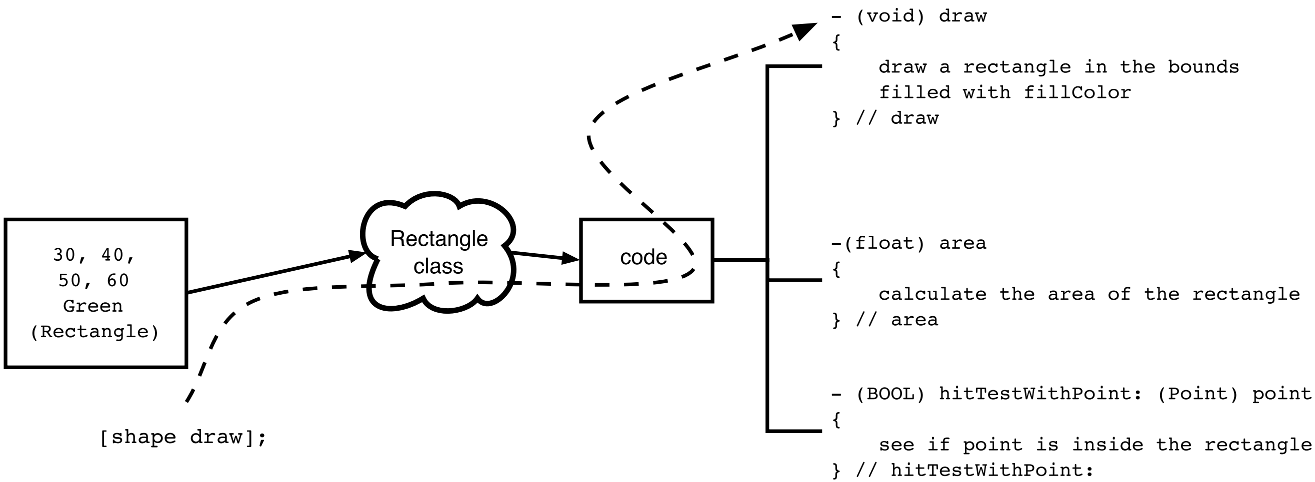


Figure 3.8. A Rectangle find its draw code.

the chance of calling the wrong function and makes our code easier to maintain.

Time Out for Terminology

Before we dig into the rest of the Shapes-Object program, let's take a moment to go over some object-oriented terminology. We've already talked about some of these terms – others are brand new.

Class: A structure that represents an object's type. An object refers to its class to get various information about itself, particularly what code to run to handle each action. Simple programs might have a handful of classes; moderately complex ones will have a couple of dozen. Objective-C style encourages us to capitalize class names.

Object: A structure containing values and a hidden pointer to its class. Running programs typically have hundreds or thousands of objects. Objective-C variables that refer to objects are typically not capitalized.

Instance: Another word for object. For example, a circle object can also be called an instance of class Circle.

Message: An action that an object can perform. This is what you send to an object to tell it to do something. In the code `[shape draw]` above, the `draw` message is sent to the `shape` object to tell it to draw itself. When an object receives a message, its class is consulted to find the proper code to run.

Method: The code that runs in response to a message. A message, such as draw, can invoke different methods depending on the class of the object.

Method Dispatcher: The mechanism used by Objective-C to divine which method will be executed in response to a particular message. We'll get out our shovels and dig a lot more into the Objective-C method dispatch mechanism in the next chapter.

Those are the key OOP terms you'll need for the rest of the book. In addition, there are a couple of generic programming terms that will soon become very important:

Interface: The description of the features provided by a class of objects. For example, the interface for class Circle declares that circles can accept the draw message.

> The concept of interfaces is not limited to OOP. For example, header files in C provide interfaces for libraries such as the standard I/O library (which you get when you `#include <stdio.h>`), and the math library (`#include <math.h>`). Interfaces do not provide implementation details, and the general idea is that you shouldn't care about them.

Implementation: This is the code that makes the interface work. In our examples, the implementation for the circle object holds the code for drawing a circle on the screen. When you send the draw message to a circle object, you don't know or care how the function works, just that it draws a circle on the screen.

OOP in Objective-C

If your brain is starting to hurt now, that's OK. We've been filling it up with a lot of new stuff, and it takes awhile to assimilate all the terms and technology. While your subconscious is chewing on the previous couple of sections, let's take a look at the rest of the code for Shapes-Object, including some new syntax for declaring classes.

The @interface Section

Before you can create objects of a particular class, the Objective-C compiler needs some information about that class. Specifically, it has to know about the data members of the object (that is, what the C struct for the object looks like), and which features it provides. You use the **`@interface`** directive to give this information to the compiler.

In Shapes-Object, we put everything into its *main.m*. In larger programs you'll use multiple files, giving each class its own set of files. We'll explore ways of organizing classes and files in a later chapter.

Here is the interface for the Circle class.

```
@interface Circle : NSObject
{
  ShapeColor      fillColor;
  ShapeRect bounds;
}

- (void) setFillColor: (ShapeColor) fillColor;

- (void) setBounds: (ShapeRect) bounds;

- (void) draw;

@end // Circle
```

This code includes some syntax we haven't talked about yet, so let's examine it now. There's a lot of information packed into these few lines. Let's pull it apart.

The first line looks like this:

```
@interface Circle : NSObject
```

As we said in Chapter 2, whenever you see an at-sign in Objective-C, you're looking at an extension to the C language. **`@interface Circle`** says to the compiler "Here comes the interface for a new class named Circle."

> The **`NSObject`** in the **`@interface`** line tells the compiler that the **`Circle`** class is based on the **`NSObject`** class. This statement says that every **`Circle`** is also an **`NSObject`**, and every **`Circle`** will inherit all the behaviors that are defined by class **`NSObject`**. We'll explore inheritance in much greater detail in the next chapter.

After starting to declare a new class, we tell the compiler about the various pieces of data that circle objects need:

```
{
  ShapeColor      fillColor;
  ShapeRectbounds;
}
```

The stuff between the curly braces is a template used to churn out new **`Circle`** objects. It says that when a new **`Circle`** object is created, it will be made up of two elements. The first, **`fillColor`**, of type **`ShapeColor`**, is the color used to draw the circle. The second, **`bounds`**, is the circle's bounding rectangle. Its type is **`ShapeRect`**. This rectangle tells where the circle will be drawn on the screen.

You specify **`fillColor`** and **`bounds`** in the class declaration. Then, every time a **`Circle`** object is created, it includes these two elements. So, every object of class **`Circle`** has its own **`fillColor`** and its own **`bounds`**. The **`fillColor`** and **`bounds`** values are called **instance variables** for objects of class **`Circle`**.

The closing brace tells the compiler we're done specifying the instance variables for **`Circle`**.

What follows are some lines that look kind of like C function prototypes:

```
- (void) draw;

- (void) setFillColor: (ShapeColor) fillColor;

- (void) setBounds: (ShapeRect) bounds;
```

In Objective-C, these are called **method declarations**. They're a lot like good old-fashioned C function prototypes, which are a way of saying "Here are the features I support". The method declarations give the name of each method, the method's return type, and any arguments.

Let's start out with the simplest one, **`draw`**:

```
- (void) draw;
```

The leading dash signals that this is the declaration for an Objective-C method. That's one way you can distinguish a method declaration from a function prototype, which has no leading dash. Following the dash is the return type for the method, enclosed in parentheses. In our case, **`draw`** just draws, and won't be returning anything. Objective-C uses **`void`** to indicate that there's no return value.

Objective-C methods can return the same types as C functions: standard types (int, float, char), pointers, object references, and structures.

The next method declarations are more interesting:

```
- (void) setFillColor: (ShapeColor) fillColor;

- (void) setBounds: (ShapeRect) bounds;
```

Each of these methods takes a single argument. **`setFillColor:`** takes a color for its argument. Circles uses this color when they draw themselves. **`setBounds:`** takes a rectangle. Circles use this rectangle to define their bounds.

Objective-C uses a syntax technique called **infix notation**. The name of the method and its arguments are all intertwingled. For instance, you call a single-argument method like this:

```
[circle setFillColor: kRedColor];
```

A method that takes two arguments is called like this:

```
[textThing setStringValue:
           @"hello there"
           color: kBlueColor];
```

The **`setStringValue:`** and **`color:`** thingies are the names of the arguments (and are actually part of the method name – more on that later), and **`@"hello there"`** and **`kBlueColor`** are the arguments being passed.

This syntax differs from C, in which you call a function with its name followed by all its arguments, like so:

```
setTextThingValueColor (textThing,
                        @"hello there",
                        kBlueColor);
```

We really like the infix syntax, although it does look a little weird at first. It makes the code very readable, and it's easy to match arguments with what they do. With C and C++ code, you'll sometimes have four or five arguments to a function and it can be difficult to know exactly which argument does what without consulting the documentation.

The `setFillColor:` declaration starts out with the usual leading dash and the return type in parentheses:

```
- (void)
```

As with the draw method, the leading dash says, "This is the declaration for a new method." The `(void)` says that this method will not return anything. Continuing:

```
setFillColor:
```

The name of the method is `setFillColor:`. The trailing colon is part of the name. It's a clue to compilers and humans that a parameter is coming next.

```
 (ShapeColor) fillColor;
```

The type of the argument is specified in parentheses, and in this case it's one of our `ShapeColors` (such as `kRedColor`, `kBlueColor`, and so on). The name that follows, `fillColor`, is the parameter name. You use this name to refer to the parameter in the body of the method. You can make your code easier to read by choosing meaningful parameter names, rather than naming them after your pets or favorite superheroes.

> It's important to remember that the colon is a very significant part of the method's name. The method
>
> ```
> - (void) scratchTheCat;
> ```
>
> is distinct from
>
> ```
> - (void) scratchTheCat:
>               (CatType) critter;
> ```
>
> A common mistake made by many freshly minted Objective-C programmers is to indiscriminately add a colon to the end of a method name that has no arguments. In the face of a compiler error, you might be tempted to toss in an extra colon and hope it fixes things. The rule to follow is: "If a method takes an argument, it has a colon. If it takes no arguments, it has no colons."

The declaration of `setBounds:` is exactly the same as the one for `setFillColor:`, except that the type of the argument is `ShapeRect` rather than `ShapeColor`.

The last line tells the compiler we're finished with the declaration of the `Circle` class:

```
@end // Circle
```

We advocate putting comments on all `@end` statements noting the class name. This makes it easy to know what you're looking at if you've scrolled to the end of a file, or you're on the last page of a long printout.

That's the complete interface for the **`Circle`** class. Now anyone reading the code knows that that this class has a couple of instance variables and three methods. One method sets the bounds, one sets the color, and the third draws the shape.

Now that we have the interface done, it's time to write the code to make this class actually do stuff. (You didn't think we were done, did you?)

The @implementation section

The **`@interface`** section, which we just discussed, defines a class's public interface. The interface is often called the API, which is a TLA for "Application Programming Interface". (TLA is a TLA for "Three Letter Acronym".) The actual code to make objects work is found in the **`@implementation`** section.

Here is the implementation for class **`Circle`** in its entirety:

```
@implementation Circle

- (void) setFillColor: (ShapeColor) c
{
  fillColor = c;
} // setFillColor

- (void) setBounds: (ShapeRect) b
{
  bounds = b;
} // setBounds


- (void) draw
{
  NSLog (@"drawing a circle at (%d %d %d %d)
 in %@",
       bounds.x, bounds.y,
       bounds.width, bounds.height,
       colorName(fillColor));
} // draw

@end // Circle
```

Now we'll examine the code in detail, in our customary fashion. The implementation for **`Circle`** starts out with the line

```
@implementation Circle
```

`@implementation` is a compiler directive that says you're about to present the code for the guts of a class. The name of the class appears after **`@implementation`**. There is no trailing semicolon on this line, because you don't need semicolons after Objective-C compiler directives.

The definitions of the individual methods are next. They don't have to appear in the same order as they do in the `@interface`. You can even define methods in an `@implementation` that don't have a corresponding declaration in the `@interface`. You can think of these as private methods, used just in the implementation of the class.

> You might think that defining a method solely in the `@implementation` makes it inaccessible from outside the implementation, but that's not the case. Objective-C doesn't really have private methods. There is no way to mark a method as being private and preventing other code from calling it. This is a side effect of Objective-C's dynamic nature.

`setFillColor:` is the first method defined:

```
- (void) setFillColor: (ShapeColor) c
{
  fillColor = c;
} // setFillColor
```

The first line of the definition of `setFillColor:` looks a lot like the declaration in the `@interface` section. The main difference is that this one doesn't have a semicolon at the end. You may notice that we renamed the parameter to simply `c`. It's OK for the parameter names to differ between the `@interface` and the `@implementation`. In this case, if we had left the parameter name as `fillColor`, it would have hidden the `fillColor` instance variable and generated a warning from the compiler).

> Why exactly do we have to rename `fillColor`? We already have an instance variable named `fillColor` defined by the class. We can refer to that variable in this method – it's "in scope". So, if we define another variable with the same name, the compiler will cut off our access from the instance variable. Using the same variable name "hides" the original variable. We avoid this problem by using a new name for the parameter.

In the `@interface` section, we used the name `fillColor` in the method declaration because it tells the reader exactly what the argument is for. In the implementation, we have to distinguish between the parameter name and the instance variable name, and it's easiest to simply rename the parameter.

The body of the method is one line:

```
fillColor = c;
```

If you're extra-curious, you might wonder where the instance variables are stored. When you call a method in Objective-C, a secret hidden parameter called **self** is passed to the receiving object that refers to the receiving object. For example, in the code **[circle setFillColor: kRedColor]**, the method passes **circle** as its **self** parameter. Because **self** is passed secretly and automatically, you don't have to do it yourself. Code inside a method that refers to instance variables works like this:

```
self->fillColor = c;
```

By the way, passing hidden arguments is yet another example of indirection in action (bet you thought we were all done talking about indirection, huh?). Because the Objective-C runtime can pass different objects as the hidden **self** parameter, it can change which objects get their instance variables changed.

> The Objective-C runtime is the chunk of code that supports applications, including ours, when users are running them. The runtime performs important tasks like sending messages to objects and passing parameters. We'll have more about the runtime in future chapters.

The second method, **setBounds:** is just like **setFillColor:**

```
- (void) setBounds: (ShapeRect) b
{
  bounds = b;
} // setBounds
```

This code sets a circle object's bounding rectangle to be the rectangle that's passed in.

The last method is our **draw** method. Note that there's not a colon at the end of the method's name, which tells us that it doesn't take any arguments.

```
- (void) draw
{
  NSLog (@"drawing a circle at (%d %d %d %d)
 in %@",
       bounds.x, bounds.y,
       bounds.width, bounds.height,
       colorName(fillColor));
} // draw
```

The **draw** method uses the hidden **self** parameter to find the values of its instance variables, just as **setFillColor:** and **setBounds:** did. This method then uses **NSLog()** to print out the text for all the world to see.

The **@interface** and **@implementation** for the other classes (**Rectangle** and **OblateSphereoid**) are nearly identical to those for **Circle**.

Instantiating Objects

Now we're ready for the final, meaty part of Shapes-Object, in which we create lovely shape objects, such as red circles and green rectangles. The big-money word for this process is **instantiation**. When you instantiate an object, memory is allocated, and then that memory is initialized to some useful default values – that is, something other than the random values you get with freshly allocated memory. When the allocation and initialization steps are done, we say that a new object **instance** has been created.

> Because an object's local variables are specific to that instance of the object, we call them **instance variables**, often shortened to "ivars".

To create a new object, we send the **`new`** message to the class we're interested in. Once the class receives and handles the **`new`** message, we'll have a new object instance to play with.

One of the nifty features of Objective-C is that you can treat a class just like an object and send it messages. This is handy for behavior that isn't tied to one particular object, but is global to the class. The best example of this kind of message is allocating a new object. When you want a new circle, it's appropriate to ask the **`Circle`** class for that new object, rather than asking an existing circle.

Here is Shapes-Object's **`main()`** function, which creates the circle, rectangle, and egg:

```
int main (int argc, const char * argv[])
{
  id shapes[3];

  ShapeRect rect0 = { 0, 0, 10, 30 };
  shapes[0] = [Circle new];
  [shapes[0] setBounds: rect0];
  [shapes[0] setFillColor: kRedColor];

  ShapeRect rect1 = { 30, 40, 50, 60 };
  shapes[1] = [Rectangle new];
  [shapes[1] setBounds: rect1];
  [shapes[1] setFillColor: kGreenColor];

  ShapeRect rect2 = { 15, 19, 37, 29 };
  shapes[2] = [OblateSphereoid new];
  [shapes[2] setBounds: rect2];
  [shapes[2] setFillColor: kBlueColor];

  drawShapes (shapes, 3);

  return (0);

} // main
```

You can see that Shapes-Object's **`main()`** is very similar to Shapes-Procedural's. There are a couple of differences, though. Instead of an array of Shapes, Shapes-Object has an array of **`id`**'s (which you probably remember are pointers to any kind of object). You create individual objects by sending the **new** message to the class of object you want to create:

```
...
shapes[0] = [Circle new];
...
shapes[1] = [Rectangle new];
...
shapes[2] = [OblateSphereoid new];
...
```

Another difference is that Shapes-Procedural initializes objects by assigning **struct** members directly. Shapes-Object, on the other hand, doesn't muck with the object directly. Instead, Shapes-Object uses messages to ask each object to set its bounding rectangle and fill color:

```
...
[shapes[0] setBounds: rect0];
[shapes[0] setFillColor: kRedColor];
...
[shapes[1] setBounds: rect1];
[shapes[1] setFillColor: kGreenColor];
...
[shapes[2] setBounds: rect2];
[shapes[2] setFillColor: kBlueColor];
...
```

After this initialization frenzy, the shapes are drawn using the **drawShapes()** function we looked at earlier, like so:

```
drawShapes (shapes, 3);
```

Extending Shapes-Object

Remember back awhile ago when we added triangles to the Shapes-Procedural program? Let's do the same for Shapes-Object. It should be a lot neater this time. You can find the project for this in the *03.11 - Shapes-Object-2* folder of *Learn ObjC Projects*.

We had to do a lot of stuff to teach Shapes-Procedural-2 about triangles: edit the **ShapeType enum**, add a **drawTriangle()** function, add a triangle to the list of shapes, and modify the **drawShapes()** function. Some of the work was pretty invasive, especially the surgery done to **drawShapes()**, in which we had to edit the loop that controls the drawing of all shapes, potentially introducing errors.

It's better with Shapes-Object-2. We only have to do two things: create a new **Triangle** class, then add a **Triangle** object to the list of objects to draw.

Here is the **Triangle** class, which happens to be exactly the same as the **Circle** class with all occurrences of "Circle" changed to "Triangle":

```
@interface Triangle : NSObject
{
  ShapeColor      fillColor;
  ShapeRectbounds;
}
- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;
- (void) draw;

@end // Triangle

@implementation Triangle

- (void) setFillColor: (ShapeColor) c
{
  fillColor = c;
} // setFillColor

- (void) setBounds: (ShapeRect) b
{
  bounds = b;
} // setBounds

- (void) draw
{
  NSLog (@"drawing triangle at (%d %d %d %d)
 in %@",
      bounds.x, bounds.y,
      bounds.width, bounds.height,
      colorName(fillColor));
} // draw

@end // Triangle
```

One drawback to "cut and paste programming" like this is that it tends to create a lot of duplicated code, like the **`setBounds:`** and **`setFillColor:`** methods. We'll introduce you to inheritance in the next chapter, which is a fine way to avoid redundant code like this.

Next, we need to edit **`main()`** so it will create the new triangle. First, change the size of the **`shapes`** array from 3 to 4 so it will have enough room to store the new object:

```
id shapes[4];
```

Then add a block of code that creates a new Triangle, just like we create a new Rectangle or Circle:

```
ShapeRect rect3 = { 47, 32, 80, 50 };
shapes[3] = [Triangle new];
[shapes[3] setBounds: rect3];
[shapes[3] setFillColor: kRedColor];
```

And finally, we update the call to **`drawShapes()`** with the new length of the **`shapes`** array:

```
drawShapes (shapes, 4);
```

And that's it. Our program now groks triangles:

```
drawing a circle at (0 0 10 30) in red
drawing a rectangle at (30 40 50 60) in green
drawing an egg at (15 19 37 29) in blue
drawing triangle at (47 32 80 50) in red
```

Note that we were able to add this new functionality without touching the **`drawShapes()`** function or any other functions that deal with shapes. That's the power of object-oriented programming at work.

> This code provides an example of object-oriented programming guru Bertrand Meyer's "Open-Closed Principle". The **`drawShapes()`** function is open to extension: just add a new kind of shape object to the array to draw. **`drawShapes()`** is also closed to modification: we can extend it without modifying it. Software that adheres to the Open-Closed principle tends to be more robust in the face of change, because you don't have to edit code that's already working correctly.

Summary

This is a big "head space" chapter: lots of concepts and ideas – and it's a long chapter, too. We talked about the powerful concept of indirection and showed that you've already been using indirection in your programs, such as when you deal with variables and files. Then we discussed procedural programming and saw some of the limitations caused by its "functions first, data second" view of the world.

We introduced object-oriented programming, which uses indirection to tightly associate data with code that operates on it. This permits a "data first, functions second" style of programming. We talked about messages, which are sent to objects. The objects handle these messages by executing methods, the chunks of code that make the object sing and dance. Every method call includes a hidden parameter named **`self`**, which is the object itself. By using this **`self`** parameter, methods find and manipulate the object's data. The implementation for the methods and a template for the object's data are defined by the object's class. You create a new object by sending the **`new`** message to the class.

Coming up in our next chapter: inheritance, a feature that lets you leverage the behavior of existing objects so you can write less code to do your work. Hey, that sounds great! We'll see you there.

License Agreement

This is a legal agreement between you and SpiderWorks, LLC, a Virginia Limited Liability Corporation, covering your use of this electronic book and related materials (the “Book”). Be sure to read the following agreement before using the Book. BY USING THE BOOK, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT USE THE BOOK AND DESTROY ALL COPIES IN YOUR POSSESSION.

Unauthorized distribution, duplication, or resale of all or any portion of this Book is strictly prohibited. No part of this Book may be reproduced, stored in a retrieval system, shared or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

By using the Book, you acknowledge that the Book and all related products constitute valuable property of SpiderWorks and that all title and ownership rights to the Book and related materials remain exclusively with SpiderWorks. SpiderWorks reserves all rights with respect to the Book and all related products under all applicable laws for the protection of proprietary information, including, but not limited to, intellectual properties, trade secrets, copyrights, trademarks and patents.

The Book is owned by SpiderWorks and is protected by United States copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. Therefore, you must treat the Book like any other copyrighted material. The Book is licensed, not sold. Paying the license fee allows you the right to use the Book on your own personal computer. You may not store the Book on a network or on any server that makes the Book available to anyone other than yourself. You may not rent, lease or lend the Book, nor may you modify, adapt, translate, copy, or scan the Book. If you violate any part of this agreement, your right to use this Book terminates automatically and you must then destroy all copies of the Book in your possession.

The Book and any related materials are provided “AS IS” and without warranty of any kind and SpiderWorks expressly disclaims all other warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Under no circumstances shall SpiderWorks be liable for any incidental, special, or consequential damages that result from the use or inablility to use the Book or related materials, even if SpiderWorks has been advised of the possibility of such damages. In no event shall SpiderWorks’s liability exceed the license fee paid, if any.

Index

E

F

G

H

I

J

M

N

O