Programming in Objective-C 2.0

A complete introduction to the Objective-C language for Mac OS X and iPhone development

Developer's Library
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**

```cpp
// 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.
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).

![Xcode Icon](image)

**Figure 2.1** Xcode Icon

A window appears, as shown in Figure 2.3.

![New Project Menu](image)

**Figure 2.2** Starting a new project
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.

Click Choose. This brings up a new window, shown in Figure 2.5.
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>
<thead>
<tr>
<th>Extension</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.c</td>
<td>C language source file</td>
</tr>
<tr>
<td>.cc, .cpp</td>
<td>C++ language source file</td>
</tr>
<tr>
<td>.h</td>
<td>Header file</td>
</tr>
<tr>
<td>.m</td>
<td>Objective-C source file</td>
</tr>
<tr>
<td>.mm</td>
<td>Objective-C++ source file</td>
</tr>
<tr>
<td>.pl</td>
<td>Perl source file</td>
</tr>
<tr>
<td>.o</td>
<td>Object (compiled) file</td>
</tr>
</tbody>
</table>
Figure 2.6  Xcode progl project window

Figure 2.7  File progl.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:

```c
#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:

```c
// 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](image)

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.

![Terminal program icon](image)

**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
...```
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 proname

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.

proname 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
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:

```c
// 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*:

```c
// 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:

```objc
/*
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:

```c
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

```c
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:

```c
@"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 \texttt{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:

\[
\text{[pool drain];}
\]

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 \texttt{main} looks like this:

\[
\text{return 0;}
\]

It says to terminate execution of \texttt{main} and to send back, or \texttt{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 \texttt{NSLog}:

\[
\text{The Debugger has exited with status 0.}
\]

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 \texttt{NSLog} routine, as shown in Program 2.2. Remember that every Objective-C program statement must be terminated by a semicolon.

\begin{verbatim}
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;
}
\end{verbatim}
Chapter 2  Programming in Objective-C

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...
..1
...2
....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.
Displaying the Values of Variables

Program 2.4

```c
#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**

```c
#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:

```c
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:

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

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

    // COMPUTE RESULT
    int 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?

```cpp
#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;
}
```
Symbols
# directive, 567
# operator, 246
## operator, 247
% (percent), 24
    modulus operator, 104
& operator, pointers and, 283
&& (logical AND operator), 107
(*) indirection operator, 284
(++) increment operator, 292, 296-299
(—) decrement operator, 292, 296-299
= (single equal sign), if statement and, 106
== (double equal sign), if statement and, 106
? (question mark), conditional operator, 128
: (colon), conditional operator, 128
; (semicolon), 21
\ (backslash) in #define statement, 243
|| (logical OR operator), 107

A
absolute pathnames, 378
abstract classes, 183-184
abstract protocols, 234
accessor methods, 139-140
accumulators, 65
addCard: method, 356
addObject: method, 344, 373, 408
address books
creating, 345-356
looking up names in, 356-359
removing names from, 359-362
sorting entries, 362-365
address cards, creating, 345-354
address operator, pointers and, 283
AddressBook class
implementation file, 572-574
interface file, 570
AddressCard class, 411-412
implementation, 571
interface file, 569
addresses, memory
indirection operator, 302
pointers and, 301
adopting protocols, 231
alignment, justification, 84
alloc method, 39-40, 324
allocating memory, 166
allocF class method, as entry, 211-213
AND operator (&), 107
AND operator (bitwise), 68
AppKit (Application Kit) framework (Cocoa), 317, 456-457
application hierarchy diagram, 455
Application Services layer, application hierarchy diagram, 455
applications, iPhone, 461
fraction calculator, creating, 476-490
archiving
basic data types, 445-446
custom archives, NSData object, 447-450
definition of, 435
Foundation framework and, 317
keyed archive, definition of, 437
NSKeyedArchiver class, 437-439
objects, 440
copying, 450-452
plists, 435-437
argv argument, main function and, 308
argument type declaration, functions and, 267-268
arguments
argv argument, main function and, 308
argv argument, main function and, 308
comma separator, 23
command-line arguments, 20, 308-310
fractions, passing as, 144
functions and, 263-265
methods, 36-37
local variables, 147
multiple arguments in, 141-143
no name arguments, 143
parameters, 20
zone arguments, 430
arguments method, 396
argv argument, main function and, 308
arithmetic, integer, 58-60
conversions, 62-63
modulus operator, 60-61
type cast operator, 63
arithmetic conversions, 538-539
arithmetic operators, 529
associative properties, 56
binary, 56
bit operators, 67
bitwise AND operator, 68
bitwise Exclusive-OR operator, 69
bitwise Inclusive-OR operator, 69
left shift operator, 71
ones complement operator, 70-71
right shift operator, 72
precedence, 56-58
unary minus operator, 60
array elements, 256
array objects, 341-344, 362
  address books, creating, 345-365
  address cards, creating, 345-356
arrays, 241, 257
  array elements, 256-260
  character arrays, 259-260
  declaring, 257
  #define statements and, 241
  defining, 256-258
    number of elements, 259
    unions and, 303
  elements, passing functions/methods to, 268-269
  functions, passing to, 268-269
  immutable arrays, 341
  linear arrays, 260
  manipulating, 256-258
  methods, passing to, 268-269
  multidimensional arrays, 260-262, 517-518
  mutable arrays, 341
  one-dimensional arrays, 260
  operators and, 534
  pointers to, 290-293, 536-537
  single-dimension, 516
  sorting, 362, 364-365
  of structures, 278
  two-dimensional arrays, 260-262
  variable-length, 517
arrayWithCapacity: method, 342
arrayWithObjects: method, 342
ASCII format of character storage, 114
assignment operators, 64-67, 531
associative properties, arithmetic operators, 56
attributes dictionary, 380
auto keyword, 213
automatic local variables, 213, 265
autorelease messages, 324, 419
autorelease pools, 20, 323-326, 406, 418-419

B

basic data types, 514-515
binary arithmetic operators, 56
binding, dynamic, 183, 187
bit fields, 280-282
bit operators, 67, 280, 530-531
  bitwise AND operator, 68
  bitwise Exclusive-OR operator, 69
  bitwise Inclusive-OR operator, 69
  left shift operator, 71
  ones complement operator, 70-71
  right shift operator, 72
blank spaces, operators and, 109
blocks
  enumerated data types, 218
  of statements, 83
Bool data type, 73
BOOL special type, 127
Boolean variables, 123-126, 128
break statements, 95, 123, 557
  switch statements and, 121
Build menu, Xcode, 15
built-in values, 127

C

C language in relation to Objective-C language, 310
C-style strings, 296
Calculator class, 115, 488-489
    assignment operators and, 65-67
calling functions, 544-545
calls, routines, 20
case sensitivity, naming conventions, 34
categories, 165, 230
    defining, 225-226, 551-552
    instance variables and, 230
    master class definition file, 230
methods
    adding, 230
    defining, 227-230
    overriding, subclassing and, 230
    number allowed, 230
    object/category named pairs, 231
    overriding methods, 230
    protocols, adopting, 234
    subclassing and, 230
cells, memory, 301
changeCurrentDirectoryPath: method, 386
char characters, unichar characters and, 326
char data type, 49-53
character arrays, 259-260
    pointers and, 296
    terminating null character and, 260
character constants, 511-512
character string constants, 512-513
character strings, 260
    pointers to, 294-295
character variable operator, reading operators into, 117
character variables, unsigned, 222
characters
    character storage, ASCII format, 114
    newline, 22
    as signed quantities, 222
    terminating null character, 260
charPtr variable, 286
@class directive, 167, 171
class message, 196
class methods, 29, 35-36
class.h files, 133-134
classes
    abstract classes, 183-184
    AddressBook
        implementation file, 572-574
        interface file, 570
    AddressCard
        implementation file, 571
        interface file, 569
    Calculator, 115
        assignment operators and, 65-67
categories, 165
    defining, 551-552
class definition, 549-550
    Complex, 187-190, 195
    copy method, adding, 429-432
declarations, 133-138
definition, 30-32, 546
    extending, 155-156
    instance variables, 546
    interface section, 546
    method declarations, 547-549
    property declarations, 546-547
    extending, inheritance and, 162-174
    Foundation, 250
    Fraction, 30-32, 39, 101, 157, 195-197
    grandchildren, 158
    inheritance, extending classes and, 162-174
    initializing, 205-207
    message expressions, 555-556
    messages, 28
    methods, 28, 211
        adding, 162-174
compiling, Macs and
   Terminal window, 16-18
   Xcode, 10-16
Complex class, 187-190, 195
Complex data type, 73
composite objects, 235-236
compound literals, 305, 538
compound relational expressions, formation of, 107
compound relational test, 106-109
compound statements, 556
concatenation, character string constants, 512
conditional compilation, 250-251
conditional operator, 128-129, 532
conforming to protocols, 231
conformToProtocol: method, protocols and, 233
const keyword, 214, 523-524
constant character string objects, 296, 327
constant expressions, 49, 528
constants, 49
   character string constants, 512-513
   enumeration constants, 513
   floating-point, 51, 510
   integers, 510
containsObject: method, 360, 373
containsAtPath: method, 384
continue statements, 96, 557
conversions
   arithmetic, 538-539
   data types, 220-222, 538-539
   floating-point numbers, 62-63
   integers, 62-63
convertToNum: method, 101-104
convertToString: method, 488
coordinates, XYPoint class and, 166
copy method, 424-425
  classes, adding to, 429-432
  immutable object and, 433
copying. See also copy method
deep copy, 426-428
files, NSProcessInfo class, 393-396
fractions, 429-432
mutable copy, 424-425
objects, 423-425
  via archiver, 450-452
deep copies, 451-452
getter method and, 432-434
setter methods and, 432-434
shallow copy, 426-428
copyPath:toPath: method, 405
copyPath:toPath:handler: method, 383, 393, 397
copyWithZone: method, 430-434
Core Services, application hierarchy diagram, 455
count class method, as entry, 211-213
countForObject: method, 374
Cox, Brad J., 1
createFileAtPath:contents:attributes: method, 384
creating iPhone application project, 461, 463
code, entering, 463, 466
interface, designing, 467-468, 471-472, 476
currentDirectory: method, 391
custom archives
 NSData object, 447-450
  writeToFile:atomically: messages, 449

data encapsulation, 44-47
data storage, 383
data type conversions, 220-222
data types, 49
  basic data types, 55, 514-515
  Bool, 73
  char, 49, 51-53, 222
  Complex, 73
  conversions, 538-539
  derived data types, 516
    arrays, 516-518
    pointers, 521-522
    structures, 518-520
    unions, 520-521
double, 49, 51
  enumerated data type, 205, 522-523
    keyword enum, 215-218
  float, 49, 51
  id, 55, 191-194
  Imaginary, 73
  int, 49-50
  modifiers, 523-524
  qualifiers, 53-55
  storage sizes, 50
dataValue variable, 192
dealloc method, 417
  overriding, 179
debugging
  comments and, 19
  statements, 251
declarations, 513
  implementation files, 133-138
  interface files, 133-138
  @interface section, 133-134
  of methods, 37
decodeObject:forKey: method, 440

decoding
  basic data types, 445-446
  objects, 440

decoding methods, writing, 440-445

decrement operator (—), 83, 292, 296-299, 531

deep copy, 426-428, 451-452

defaultManager messages, 380

#define statement, 239-241, 562-564
  arguments, 243
  arrays and, 241
  backslash (\) character, 243
  constant values and, 241
  defined names, 241-245
  defined names and, 241
  macros, 244-245
    # operator and, 246
    ## operator and, 247
  placement of, 240
  preprocessor and, 239-241
  syntax, 240

defined names, 241-245

deleterCharactersInRange: method, 336

deleting names from address books, 359-362

denominator integer instance variable, 101

derived data types, 516
  arrays
    multidimensional, 517-518
    single dimension, 516
    variable-length, 517
  pointers, 521-522
  structures, 518-520
  unions, 520-521

description method, 374

designated initializer, 206

designing interface for iPhone application project, 467-468, 471-472, 476

device drivers, application hierarchy diagram, 455

dictionaries
  attributes dictionary, 380
  enumerations, 368-369
  immutable, 367
  mutable, 367
  plists, creating from, 436

dictionary objects, 367-369

dictionaryWithObjectsAndKeys: method, 368-369

digraph characters, 505

directives, 507-508
  instance variables, scope control, 208
  @package directive, 208
  @private directive, 208
  @protected directive, 208
  @protocol directive, 233
  @public directive, 208
  @selector directive, 197

directories
  contents, enumerating, 387-389
  home directories (~), 378
  managing via NSFileManager, 380-382, 405
    basic directory operations, 385-386
    defaultMessage messages, 380
    enumerating directory contents, 387-389
    methods of, 379, 384-385
    NSDAta class, 383-384
  root directories, 378

directoryContentsAtPath: method, 387, 389

dispatch tables, 301

divide: method, 120

do statements, 94-95, 557
do-while loops, 344
doesNotRecognize: method, 198
dot operator, 271
  properties, accessing via, 140
double data type, 49-51
dynamic binding, 183, 187, 191-193
dynamic typing, 187, 192, 195-196

E
#elif preprocessor statement, 252
else clause, 110-111, 119
else if statement, 111-114, 117-120
  conditional operator and, 129
  n-valued logic decision and, 111
#else statement, 250-251
encodeObject:forKey: method, 440, 442
encodeWithCoder: method, 440, 442
encoding
  basic data types, 445-446
  objects, 440
encoding methods, writing, 440-445
#endif statement, 250-251
enumerated data type, 522-523
  blocks and, 218
  definition of, 205
  enum keyword, 215-218
enumeration
  constants, 513
  dictionaries, 368-369
  directory contents, 387-389
  identifier, 215-218
enumerationAtPath: method, 389
enumeratorAtPath: method, 387
#error directive, 564
escape sequences, 511
exception handling, 200-202, 561

Exclusive-OR operator (bitwise), 69
exponents, floating-point values, 51
expressions
  compound relational expressions,
    formation of, 107
  constant expressions, 49, 528
  lvalues, 524
  message expressions, 555-556
  operators, 524-527
extending classes, inheritance and, 162-174, 181-183
extensions, filenames, 12
extern keyword, 209-212
external global variable, 209
external variable, 209-211
  definition of, 209
  extern keyword, 209-212
  static keyword, 211-213

F
factory methods, 29
fast enumeration, 354-356
Fibonacci numbers, 257-258
fields, bit fields, 282
fileAttributesAtPath:traverseLink: method, 380, 382
fileExistsAtPath: method, 389
fileHandleForUpdatingAtPath: method, 398
fileHandleForWritingAtPath: method, 398
filenames extensions, 12
files
  basic operations of, 397-402
class.h, 133-134
  copying, NSProcessInfo class, 393-396
  include files, 250
  linking, 378
  managing via NSFileManager, 380-382, 405
basic directory operations, 385-386
defaultMessage messages, 380
enumerating directory contents, 387-389
methods of, 379, 384-385
NSData class, 383-384
master class definition file, categories in, 230
naming, 12
NSFileHandle class, 399-402
methods of, 378, 397-398
opening, 378
pathnames, 378
NSMutableArray.h, 389-396
float data type, 49, 51
floating constants, hexadecimals, 51
floating-point constants, 51, 510
floating-point numbers, 23, 62-63
for loops, execution order, 81
for statement, 78-89, 557
keyboard input, 84-86
loop variants, 88-89
nested loops, 86-88
terminal input, 85-86
format strings, 25
forwarding, 162
Foundation Framework, 317
archiving and, 317
array objects, 341-344
address book creation, 345-365
address card creation, 345-356
Cocoa, 456-457
dictionary objects, 367-369
initialization methods and, 206
Mac documentation, 317-319
methods
copy method, 424-425
mutableCopy: method, 424-425
mutable strings, 333-337
NSCountedSet class, 373
number objects, 322-326
protocols and, 231
set objects, 370-374
string objects, 326-340
typedef statement and, 219
Foundation library, NSLog function, 327
Foundation string class, 250
fraction calculator application, creating for iPhone, 476-480
Calculator class, 488-489
Fraction class, 485-488
user interface, designing, 490
view controller, defining, 480-485
Fraction class,
30-32, 39, 101, 157, 195-197, 485-488
class method, 211-213
convertToNum: method, 101-104
copy method, adding, 429-432
fractions
adding, 144-146
arguments, passing as, 144
copying, 429-432
referencing, 145
frameworks, defining, 3
function calls, 311
functions, 543
argument types, declaring, 267-268
arguments, 263-265
arrays, passing, 268-269
calling, 544-545
definitions, 543-544
local variables, 263-265
main, 262-263, 308
multidimensional array element, passing, 270
pointer to array, passing to, 293-294
pointers, 300-301, 545
as argument, 301
dispatch tables, 301
returning as result, 288-289
printMessage, 262-263
prototypes, declaring, 267
qsort function, 301
results, returning, 265-266
return types, declaring, 267-268
sign, 111
static functions, 268
storage class, 540
values, returning, 265-266

garbage collection (memory management), 420
gcc command, 17
generic pointer type, id type, 311
getter methods, 46, 432-434
global structure definition, 276
global variable, 209-211
goto statements, 306, 558
grandchildren (classes), 158
GUI (graphical user interfaces), 3
headers, precompiled, 321
hexadecimal, floating constants, 51
home directories (~), 378
id data type, 55, 191-194, 311
id object declaration, 554-555
identical objects, 360
identifiers, 506
directives, 507-508
enumeration identifier, 215-218
keywords, 506-507
predefined, 509
universal character names, 506
if statement, 99-104, 558
= (equal sign) and, 106
nested, 109-111
satisfied conditions, 125
#if statement, 252, 564-565
if-else statement, 104-106
#ifdef statement, 250-251, 565
#ifndef statement, 250-251, 565
Imaginary data type, 73
immutable arrays, 341
immutable dictionaries, 367
immutable objects, 328-332
immutable strings, 426-427
implementation dependency, 50
implementation files
AddressBook class, 572-574
AddressCard class, 571
Fraction.m, 136
@implementation section (object-oriented programming), 32, 37-38
implementation section, class definition, 549
#import statement, 247-250, 565-566
include files, 250
#include statement, 247-250, 566
Inclusive-OR operator (bitwise), 69
increment operator (++), 83, 292, 296-299, 531
index number, 256
indexOfObject: colon, 360
indirection, pointers and, 283
indirection operator (*), 284
pointers, memory address and, 302
informal protocols, 234-235
inheritance
extending classes, 162-174
instance variables, 181-183, 208
methods, 161
adding to classes, 162-174
determining which method is selected, 177-179
overriding, 175-176, 179
root classes, 157-158, 161
inherited methods, 161
init method, 40
initialization
array elements, 258-259
character array, pointers and, 296
classes, 205-207
instance variables, 206
methods, 206
structures, 277
two-dimensional arrays, 261-262
initialize method, 207
initializers, designated, 206
initVar: method, 158
initWithCoder: method, 440, 442
initWithName: method, 352, 354
insertString:atIndex: method, 336
installing iPhone SDK, 459
instance methods, 35-36
instance variables, 33, 35, 44, 541-542
accessing, 44-47
categories and, 230
declarations, 546
extending classes through inheritance, 181-183
getters, 46
inherited, 208
initialization, 206
reference counting, 411-417
scope, 208
setters, 46
in structures, 310-311
instances
defining, 27
methods, 28-29
int data type, 49-50
integer arithmetic, 58-60
conversions, 62-63
modulus operator, 60-61
type cast operator, 63
integer instance variables, 101
integers, 510
integral promotion, 539
interface, designing for iPhone application project, 467-468, 471-472, 476
Interface Builder, 460
interface files, 134
AddressBook class, 570
AddressCard class, 569
compilers and, 136
extending, 155-156
interface section, class definition, 546
@interface section (object-oriented programming), 32
class declarations, 133-134
instance variables, 33-35
methods, 33-37
names, choosing, 33-34
parent classes, 33
intersect: method, 373
intPtr variable, 283, 285
iPhone
application, creating, 461-463
code, entering, 463, 466
interface, designing, 467-468, 471-472, 476
application templates, 461
Cocoa Touch, 456-457
fraction calculator application, creating, 476-480
  Calculator class, 488-489
  Fraction class, 485-488
user interface, designing, 490
  view controller, defining, 480-485
Objective-C development, 2
web resources, 578
iPhone SDK, installing, 459
iPhone simulator, 460
isa member, 310
isEqualToNumber: method, 325
isEqualToString: method, 330
J - K
justification, right justification, 84
keyboard input, for loops, 84-86
keyed archives, definition of, 437
keywords, 506-507
  const, 214, 523-524
  enum, 215-218
  restrict, 523-524
  volatile, 523-524
linear arrays, 260
linking files, 378
LinuxSTEP development environment, Objective-C development, 1
literals, compound, 305, 538
local files, quotes, 137
local structure definition, 276
local variables, 146-147
  auto keyword, 213
  functions and, 263, 265
  method arguments, 147
  static keyword, 147-149, 265
logical AND operator (&&), 107
logical negation operator, 126
logical operators, 529
logical OR operator (||), 107
logical right shift operators, 72
long long qualifier, 53-54
long qualifier, 53-54
lookup: method, 356-362
loop conditions, 79
loop variable, 125
loops
  break statement, 95
  continue statement, 96
  do loops, 94-95
  do-while loops, 344
  for loops, 78-89
    execution order, 81
    keyboard input, 84-86
    nested, 86-88
    terminal input, 85-86
    variants, 88-89
  while loops, 89-93
lookup: method, 356-362
loop conditions, 79
loop variable, 125
loops
  break statement, 95
  continue statement, 96
  do loops, 94-95
  do-while loops, 344
  for loops, 78-89
    execution order, 81
    keyboard input, 84-86
    nested, 86-88
    terminal input, 85-86
    variants, 88-89
  while loops, 89-93
lowercaseString: method, 330
lvalues, expressions, 524
machine dependency, 50
Macintosh Foundation framework, documentation, 317, 319
macros, 244-245
  # operator and, 246
  ## operator and, 247
  #define statements and, 244-245
Macs
  Cocoa development environment, Object-C development, 1
  compiling and
    Terminal window, 16-18
    Xcode, 10-16
  iPhone, Objective-C development, 2
main function, 308
mantissa, floating-point values, 51
matrixes, 260
memory, 405
  addresses, 301-302
  allocating, 166
  autorelease pools, 20, 323-326, 406
    example of, 418-419
  cells, 301
  garbage collection, 420
  leakage, 153
  management rules, summary of, 419-420
  reference counting
    instance variables, 411-417
    objects, 407-409
    strings, 409-411, 423
  releasing objects, 338-340
  uses, 301
message expressions, 311, 555-556
messages, 28, 196
methods, 33
  adding to classes, 162-174
  arguments, 36-37
    local variables, 147
    multiple, 141-143
    no name arguments, 143
  arrays, passing, 268-269
  categories, defining, 227-230
  class methods, 29, 35-36, 211-213
  conformityToProtocol, protocols and, 233
  convertToNum, 101-104
  copy method, immutable objects and, 433
  declarations, 37, 547-549
  decoding methods, writing, 440-445
  defining, 37
    class definition, 549-550
    protocols and, 232
  determining which method is selected, 177-179
  doesNotRecognize: method, 198
  encoding methods, writing, 440-445
  factory methods, 29
  as functions, 311
  getter methods, 46, 432-434
  inherited methods, 161
  inheritance, 161
  initialization methods, 206
  instance methods, 28-29, 35-36
  multidimensional array element, passing, 270
  objects, allocating/returning, 150-154
  overriding, 175-176
    dealloc method, 179
    release methods, 179-180
    subclassing categories and, 230
perform method, 197
pointers, returning as result, 288-289
return values, 36
setOrigin: method, 172, 179
setter methods, 46, 432-434
synthesized accessor methods, 550
module, definition of, 207
modulus operator, 60-61, 104
movePath:toPath: method, 382, 405
movePath:toPath:handler: method, 386
multibyte characters, character string constants, 512
multidimensional arrays, 260-262, 270, 517-518
multiply method, 67
mutable arrays, 341
mutable dictionaries, 367
mutable objects, 328-332
mutable strings, 333-337, 426-427
mutableCopy: method, 424-425, 428

N

name definitions, 241-245
named pairs, object/category, 231
names
    address books
        looking up in, 356-359
        removing from, 359-362
        sorting in, 362-365
    files, 12
naming conventions, 33-34
navigation-based iPhone application templates, 462
negation operator, 126
nested if statement, 109-111
nested loops, for loops, 86-88
newline character, 22

NEXTSTEP development environment, Objective-C development, 1
NSArray class, methods of, 366
NSCoding protocol, 440
NSCopying protocol, 452
NSCountedSet class, 373
NSData class, 383-384
custom archiving, 447-450
NSDictionary class, methods of, 369
NSFileHandle class, 399-402
methods of, 378, 397-398
NSFileManager, 380-382
defaultMessage messages, 380
directories
    basic operations of, 385-386
    enumerating contents of, 387-389
    methods of, 379, 384-385
NSData class, 383-384
NSHomeDirectory function, 391
NSKeyedArchiver class, archiving with, 437-439
NSLog function, 20, 327
% (percent) character, 24
NSNSMutableArray class, methods of, 366
NSNSMutableArray method, 362
NSNSMutableArray class, methods of, 370
NSMutableSet class, methods of, 374
NSMutableString class, 328-338, 340, 411
NSNumber class, 322-324, 326
NSNumber objects, 324-325, 408
NSObject class, 157, 159
NSPathUtilities.h, 389-391
    functions of, 393
    methods of, 392-393
    NSProcessInfo class, 393-396
NSProcessInfo class, 393-396
NSSet class, methods of, 374
NSString class, 338-340, 410
NSTemporaryDirection function, 391
null character, 260
null statements, 306, 559
number objects, 322-326
numberOfDays function, 277-278
numbers
  floating point, 23
  index number, as reference, 256
  real numbers, 23
  subscript, as reference, 256
  triangular, 77
  truncated numbers, 62-63
numerator integer instance variable, 101

O

Object class, 197
object variables, as pointer variables, 311
object-oriented programming, 28, 282
  classes
    defining, 30-32
    fractions and, 30-32
    messages, 28
    receivers, 28
  @implementation section, 32, 37-38
  instances, 27-29
  @interface section, 32
    instance variables, 33-35
    methods, 33-37
    name selection, 33-34
    parent classes, 33
    methods, 28-29
    objects, defining, 27
    program section, 32, 38-44
object/category named pairs, 231
objectAtIndex: method, 342, 428
objectForKey: method, 368
Objective-C
  development of, 1
  as procedural language, 2
  web resources, 576
objects
  array objects, 341-344
    address book creation, 345-365
    address card creation, 345-356
  assigning, 423, 428
  classes, ownership, 171-174
  composite objects, 235-236
  constant character string objects, 513
  copying, 423-425
    deep copies, 426-428, 451-452
    getter methods and, 432-434
    setter methods and, 432-434
    shallow copying, 426-428
    via archiver, 450-452
  defining, 27, 554-555
  dictionary objects, 367-369
  id object, declaration, 554-555
  immutable objects, 328-332
    copy method and, 433
  memory release, 338-340
  methods, allocating/returning, 150-154
  mutable copying, 424-425
  mutable objects, 328-332
  number objects, 322-326
  reference counting, 407-409
  set objects, 370-374
  Square object, 196
  state, 28
  string objects, 326-340
octal notation, int data type, 50
one-dimensional arrays, 260
ones complement operator, 70-71
ONESTEP development environment, Objective-C development, 1
OpenGL ES iPhone application templates, 462
opening files, 378
operations
increment operation, 297
on pointers, 300-301
operators, 524-527
& operator, pointers and, 283
address operator, pointers and, 283
arithmetic, 529
associative properties, 56
binary, 56
bit operators, 67-72
precedence, 56-58
type cast operator, 63
arrays and, 534
assignment operators, 64-67, 531
bit operator, 280
bitwise operators, 530-531
blank spaces and, 109
comma operator, 306-307, 533
conditional operator, 128-129, 532
decrement operator (—), 83, 292, 296-299, 531
dot operator, 271
increment operator (++), 83, 292, 296-299, 531
indirection operator (*), 284
logical AND operator (&&), 107
logical negation operator, 126
logical operators, 529
logical OR operator (||), 107
modulus, 60-61, 104
pointers and, 535-537
post-increment operator, 297-298
pre-increment operator, 297-298
reading into character variable operator, 117
relational operators, 80, 530
sizeof operator, 307-308, 533
structures and, 534-535
ternary operator, 128
type cast operators, 532
unary minus operator, 126
unary operator, pointers and, 283
OR operator (||), 107
origin values, storing as separate, 166
outlets, 464
overriding methods, 175-176
categories, subclassing and, 230
dealloc method, 179
release method, 179-180
super keyword, 179-180

@package directive, 208
parent classes, 33
parentheses around condition operators, 128
pathComponents: method, 392
pathExtension: method, 391
pathnames
absolute pathnames, 378
NSPathUtilities.h, 389-391
functions of, 393
methods of, 392-393
NSProcessInfo class, 393-396
relative pathnames, 378
percent (%) character, 24
perform method, 197
plists
archiving with, 435-437
dictionaries, creating from, 436

pointers, 39, 283-284, 521-522
& operator and, 283
address operator and, 283
arrays and, 290-293, 536-537
character arrays and, 296
character strings and, 294-295
charPtr variable, 286
constant character strings and, 296
decrement operator (—), 292, 296-299
definition of, 283
function pointers, 545
functions and, 300-301
dispatch tables, 301
passing to, 288-289
increment operator (++), 292, 296-299
indirection and, 283
intPtr variable, 283, 285
memory addresses and, 301-302
methods, passing to, 288-289
operations, 300-301
operators and, 535-537
passing as arguments, 288-289
post-increment operator, 297-298
pre-increment operator, 297-298
structures and, 287-288, 537
types, id type, 311
unary operator and, 283

polymorphism, 187-192

positive integers, prime numbers and, 123
post-increment operator, 297-298
pound sign (#), preprocessor and, 239
#pragma directive, 566

pre-increment operator, 297-298
precompiled headers, 321
predefined identifiers, 509, 567
preprocessors
conditional compilation, 250-251
defining, 239, 242
directives, 562
#, 567
#define, 239-241, 562-564
#error, 564
#if, 564-565
#endif, 565
#include, 247-250, 565-566
#include, 247-250, 566
#line, 566
#pragma, 566
#undef, 567
predefined identifiers, 567
pound sign (#) and, 239
trigraph sequences, 561

primes array, allocation and, 344
print method, 38, 41, 371, 373
printVar method, 159
@private directive, 208

procedural language, Objective-C as, 2
program section (object-oriented programming), 32, 38-44

program statements, if statement and, 99

programming. See object-oriented programming
programming errors, 118-119

properties, accessing via dot operator, 140
property lists, 435
property variables, declarations, 546-547
@protected directive, 208
@protocol directive, 233

protocols
  abstract protocols, 234
  adopting, 231-232
  category adoption of, 234
  conforming to, 231
  defining, 231, 552-553
  Foundation framework and, 231
  informal, 234-235
  methods, 232-233
  @protocol directive, 233

prototype declaration, 267
@public directive, 208

Q

qsort function, 301

qualifiers, data types, 53-54

quotes, local files, 137

R

rangeOfString: method, 333, 337
ranges, string, 331-332
=readDataOfLength: method, 400
=readDataToEndOfFile: method, 400
real numbers, 23
receivers, 28
Rectangle class, 198-199
reference counting
  instance variables, 411-417
  objects, 407-409
  strings, 409-411, 423
references, numbers, 256
relational operators, 80, 530
relative pathnames, 378
release messages, memory management, 419
release method, overriding, 179-180
removeCard: method, 359-362
removeFileAtPath: method, 380
removeFileAtPath:handler: method, 382, 397
removeObject: method, 360-362, 373
removeObjectAtIndex: method, 409
removeObjectIdenticalTo: method, 360
replaceObject:atIndex:withObject: method, 429
reserved names/words (naming conventions), 34
restrict keyword, 523-524
results, returning, 265-266
retain counts, 426
retainName:andEmail: method, 434
return statements, 559
return type declaration, functions and, 267-268
return values, methods, 36
right justification, 84
right shift operator, 72
Ritchie, Dennis, 1
root classes, 157-158, 161
root directories, 378
routines. See also functions; methods
calls, 20
  getter routines, 432-434
  setter routines, 432-434
runtime checking, 193
S
satisfied conditions, if statements, 125
scientific notation, 51
scope, 539-540
  instance variables, directives, 208
  variables, 207
seekToEndOfFile: method, 402
@selector directive, 197
selectors, 197
self keyword, 149-150
semicolon (;), 21
set objects, 370-374
setEmail: method, 346
setName: method, 346, 411-412
setName:andEmail: method, 434
setObject:forKey: method, 368
setOrigin: method, 172, 179
setStr: method, 413-416
setString: method, 337
setter methods, 46, 432-434
setWithObject: method, 373
shallow copy, 426-428
short qualifier, 53-54
sign extension, 221-222
sign function, 111
signed char variable, 222
signed qualifier, 53-54
signed quantities, characters as, 222
single-dimensional arrays, 516
sizeof operators, 307-308, 533
skipDescendants messages, 387
slashes (/ /), comments, 19
sort method, 364
sorting arrays, 362-365
sortUsingSelector: method, 362
special type BOOL, 127
Square class, 198-199
    as subclass, 164
Square object, 196
state (objects), 28
statements, 20, 556
    blocks, 83
    break statement, 95, 121, 123
compound, 556
continue statement, 96
debugging statements, 251
#define statement, 239-241
    arguments, 243
    backslash (\) character, 243
    constant values and, 241
    defined names, 241-245
    macros, 244-245
    placement of, 240
    syntax, 240
do loops, 94-95
    #elif preprocessor statement, 252
    #else, 250-251
    else if statement, 111-114, 117-120
    #endif statement, 250-251
    for loops, 78-89
goto statement, 306
    #if preprocessor statement, 252
    if statement, 99-104
        nested, 109-111
        program statements and, 99
    if-else statement, 104-106
    #ifdef statement, 250-251
    #ifndef statement, 250-251
    #import statement, 247-250
    #include statement, 247-250
null statement, 306
program statement, if statement and, 99
switch statement, 120, 123
    if statement, translation into, 121
typedef statement, 205, 218-219
    #undef statement, 253
while loops, 89-93
static functions, 268
### static local variables, 147-149, 265
- subclasses, 157
  - alternatives to, 235
  - creating, 164
  - designated initializer and, 206

### static typing, 194

### static variable, 211-213

### storage (data), 383

### storage class, 539-540

### string objects, 326-340

### string ranges, 331-332

### stringByAppendingPathComponent: method, 391, 397

### stringByAppendingString: method, 330

### stringByExpandingTildeInPath: method, 392

### stringByStandardizingPath: method, 392

### strings
- character string constants, 512-513
- character strings, 260
- format strings, 25
- immutable strings, 426-427
- mutable strings, 333-337, 426-427
- reference counting, 409-411, 423

### stringWithFormat: method, 488

### stringWithString: method, 411, 413

### structure variables, 271, 333

### structures, 271, 273-275, 277, 280, 518-520
- arrays of, 278
- bit fields, 281-282
- classes and, 276
- defining, 278-280
- global structure definition, 276
- initializing, 277
- instance variables, storage in, 310-311
- local structure definition, 276
- members, 310
- operators and, 534-535
- pointers and, 287-288, 537
- structures within, 278-279
- syntax of, 271
- variables, 271-273

### subclasses, 157
- alternatives to, 235
- creating, 164
- designated initializer and, 206

### subclassing categories, 230

### subscript, 256

### substringFromIndex: method, 332

### substringToIndex: method, 332

### substringWithRange: method, 333

### super keyword, overriding, 179-180

### switch statement, 120-123, 560

### syntax, structures, 271

### @synthesize directive, 139

### synthesized accessor methods, 139-140, 550

---

### T

- Tab Bar iPhone application templates, 462

### tables
- dispatch tables, 301
- truth tables, 68

### templates (iPhone application), 461

### terminal input for loops, 85-86

### Terminal window, compiling on Macs, 16-18

### terminating null character, 260

### ternary operator, 128

### tests, compound relational test, 106-109

### tilde (~), home directory, 378

### triangular numbers, 77

### trigraphs, 561

### truncated numbers, 62-63

### truth tables, 68

### @try blocks, 200-202

### two-dimensional arrays, 260
- elements, 260
- initializing, 261-262
- matrixes and, 260
type cast operators, 63, 532

type modifiers, protocol definitions, 553

typedef statement, 205, 218-219, 523

types
   BOOL (special type), 127
dynamic typing, 187, 192, 195-196
static typing, 194

U
unarchiving objects, 440
unary minus operator, 60, 126
unary operator, pointers and, 283
#undef statement, 253, 567
unichar characters, 326
union: method, 373
unions, 302-304, 520-521
units, 282
universal character names, 506
unsigned character variable, 222
unsigned qualifier, 53-54
uppercaseString: method, 330
user interface, designing for iPhone fraction calculator project, 490
usual arithmetic conversions, 538-539
utility iPhone application templates, 462

V
values
   0, FALSE or off state, 125-126
   1, TRUE or off state, 126
   1, TRUE or on state, 125
   built-in, 127
   return values, methods, 36
   returning, 265-266
   variables, displaying, 22-24
variable-length arrays, 517

variables
   Boolean variables, 123-128
   character variables, unsigned, 222
   charPtr variable, 286
   dataValue variable, 192
   external global variable, 209
   external variable, 209-213
   global variable, 209-211
   instance variables, 33-35, 44, 541-542
      accessing, 44-47
      categories and, 230
      declarations, 546
      extending classes and, 181-183
      getters, 46
      inherited, 208
      initialization, 206
      reference counting, 411-417
      scope control, 208
      setters, 46
   integer instance, 101
   intptr, 283, 285
   keywords
      auto, 213
      const, 214
      extern, 209-212
      static, 211-213
      volatile, 214-215
   local, 263-265
   loop variable, 125
   object variable, as pointer variable, 311
   property variables, declarations, 546-547
   scope, 207
   signed char variable, 222
   storage class, 540
   structure definition and, 280
   structure variables, 271, 333
   values, displaying, 22-24
variants for loops, 88-89
view controller, defining for iPhone fraction calculator project, 480-485
view-based iPhone application templates, 462
volatile variable, 214-215, 523-524

W

web resources
  Cocoa, 577
  iPhone, 578
  Objective-C, 576
while statement, 89-93, 560
wide character constants, 512
wide character string constants, 513
Windows-based iPhone application templates, 462
writeToFile:atomically: messages
  custom archives, 449
  plist archives, 436

X – Y - Z

Xcode, 460
  Build menu, 15
  compiling on Macs, 10-16
  iPhone application, creating, 461
    code, entering, 463
    interface, designing, 467
XYPoint class, 166, 199

zone arguments, 430