



# Java Programming Primer

Ideally, you should have a fairly solid grasp on the Java programming language before diving into this book. But maybe it's been a while since you used Java, or maybe you're just adventurous enough to tackle mobile game programming with a passing knowledge of Java. If you're in either of these camps, this primer is for you. It isn't intended to teach you everything there is to know about Java—there are plenty of other good books that do that. Instead, this primer attempts to provide you with enough core Java knowledge to dive into mobile game programming.

## Hello, World!

The best way to learn a programming language is to jump right in and see how a real program works. In keeping with a traditional introductory programming example, following is a Java version of the classic “Hello, World!” program:

```
class HelloWorld {  
    public static void main(String args[]) {  
        System.out.println("Hello, World!");  
    }  
}
```

It's worth pointing out that this program isn't designed to be run on a mobile phone. Remember, this primer is a general Java programming primer, so none of it is specific to mobile phone development. In fact, most of the example code in this primer is designed for use in command-line Java applications, which are the simplest of all Java programs.

After compiling the HelloWorld program with the Java compiler (javac), you are ready to run it in the Java interpreter. The Java compiler places the executable output in a file called HelloWorld.class. This naming convention might seem strange considering the fact that most programming languages use the .EXE file extension for executables. Not so in Java!

Following the object-oriented nature of Java, all Java programs are stored as Java classes that are created and executed as objects in the Java runtime environment. To run the HelloWorld program, type **java HelloWorld** at the command prompt. If you've properly installed the Java Software Development Kit (SDK), the program will respond by displaying "Hello, World!" on your computer screen.

To fully understand what is happening in HelloWorld, let's examine the program line by line. First, you must understand that Java relies heavily on classes. In fact, the first statement of HelloWorld reveals that HelloWorld is a class, not just a program. Furthermore, by looking at the class statement in its entirety, you can see that the name of the class is defined as HelloWorld. This name is used by the Java compiler as the name of the executable output class.

The HelloWorld class contains one member function, or method. For now, you can think of this function as a normal procedural function that happens to be linked to the class. The single method in the HelloWorld class is called main(), and should be familiar if you have used C or C++. The main() method is where execution begins when the class is executed in the Java interpreter. The main() method is defined as being public static with a void return type. public means that the method can be called from anywhere inside or outside the class. static means that the method is the same for all instances of the class. The void return type means that main() does not return a value.

The main() method is defined as taking a single parameter, String args[]. args is an array of String objects that represent command-line arguments passed to the class at execution. Because HelloWorld doesn't use any command-line arguments, you can ignore the args parameter. You learn a little more about strings later in this primer.

The main() method is called when the HelloWorld class is executed. main() consists of a single statement that prints the message "Hello, World!" to the standard output stream, as follows:

```
System.out.println("Hello, World!");
```

This statement may look a little confusing at first because of the nested objects. To help make things clearer, examine the statement from right to left. First, notice that the statement ends in a semicolon, which is standard Java syntax borrowed from C/C++. Moving to the left, you see that the "Hello, World!" string is in parentheses, which means it is a parameter to a method call. The method being called is actually the `println()` method of the `out` object. The `println()` method is similar to the `printf()` method in C, except that it automatically appends a newline character (`\n`) at the end of the string. The `out` object is a member variable of the `System` object that represents the standard output stream. Finally, the `System` object is a global object in the Java environment that encapsulates system functionality.

That pretty well covers the `HelloWorld` class, which is your first Java program. If you got lost a little in the explanation of the `HelloWorld` class, don't be too concerned. `HelloWorld` was presented with no previous explanation of the Java language and was meant only to get your feet wet with Java code. The rest of this primer focuses on a more structured discussion of the fundamentals of the Java language.

## Tokens

When you submit a Java program to the Java compiler, the compiler parses the text and extracts individual tokens. A *token* is the smallest element of a program that is meaningful to the compiler; tokens define the structure of the Java language. All the tokens that comprise Java are known as the *Java token set*. Java tokens can be broken into five categories: identifiers, keywords, literals, operators, and separators. The Java compiler also recognizes and subsequently removes comments and whitespace.

The Java compiler removes all comments and whitespace while tokenizing a source file. The resulting tokens are then compiled into machine-independent Java bytecode capable of being run from within an interpreted Java environment. The bytecode conforms to the hypothetical Java virtual machine, which generalizes processor differences into a single virtual processor. Keep in mind that an interpreted Java environment can be the Java command-line interpreter, a Java-capable browser, or in the case of most of the examples in this book, a Java-powered mobile phone.

Identifiers

Identifiers are tokens that represent names. These names can be assigned to variables, methods, and classes to uniquely identify them to the compiler and give them meaningful names for the programmer. HelloWorld is an identifier that assigns the name HelloWorld to the class residing in the HelloWorld.java source file.

Although you can be creative in naming identifiers in Java, there are some limitations. All Java identifiers are case sensitive and must begin with a letter, an underscore ( \_ ), or a dollar sign ( \$ ). Letters include both uppercase and lowercase letters. Subsequent identifier characters can include the numbers 0 to 9. The only other limitation to identifier names is that the Java keywords, which are listed in the next section, cannot be used. Table A.1 contains a list of valid and invalid identifier names.

TABLE A.1 Valid and Invalid Java Identifiers

Valid	Invalid
HelloWorld	Hello World (uses a space)
Hi_Mom	Hi_Mom! (uses a space and punctuation mark)
heyDude3	3heyDude (begins with a numeral)
tall	short (this is a Java keyword)
poundage	#age (does not begin with letter)

In addition to the mentioned restrictions in naming Java identifiers, you should follow a few stylistic rules to make your Java programming easier and more consistent. It is standard Java practice to name multiple-word identifiers in lowercase except for the beginning letter of words in the middle of the name. For example, the variable toughGuy is in correct Java style; the variables toughguy, ToughGuy, and TOUGHGUY are all in violation of this style rule. The rule isn't etched in stone—it's just a good rule to follow because most other Java code you run into follows this style.

Keywords

Keywords are predefined identifiers reserved by Java for a specific purpose and are used only in a limited, specified manner. The following keywords are reserved for Java:

abstract	double	int	strictfp
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	package	throw
char	for	private	throws
class	goto	protected	transient
const	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while

In addition to these keywords, Java also reserves the words `null`, `true`, and `false`, so they are off limits as well.

## Literals

Program elements used in an invariant manner are called *literals* or *constants*. Literals can be numbers, characters, or strings. Numeric literals include integers, floating-point numbers, and Booleans. Character literals always refer to a single Unicode character. Strings, which contain multiple characters, are still considered literals even though they are implemented in Java as objects.

If you aren't familiar with the Unicode character set, it is a 16-bit character set that replaces the ASCII character set. Because it is a 16-bit character set, there are enough entries to represent many symbols and characters from other languages.

**Construction Cue**



## Integer Literals

Integer literals are the primary literals used in Java programming. They come in a few different formats: decimal, hexadecimal, and octal. These formats correspond to the base of the number system used by the literal. Decimal (base 10) literals appear as ordinary numbers with no special notation. Hexadecimal numbers (base 16) appear with a leading `0x` or `0X`, similar to the way they do in C/C++. Octal (base 8) numbers appear with a leading `0` in front of the digits. For example, an integer literal for the decimal number 12 is represented in Java as `12` in decimal, `0xC` in hexadecimal, and `014` in octal.

Integer literals default to being stored in the `int` type, which is a signed 32-bit value. If you are working with very large numbers, you can force an integer literal to be stored in the `long` type by appending an `L` or `l` to the end of the number, as in `79L`. The `long` type is a signed 64-bit value.

### Floating-Point Literals

Floating-point literals represent decimal numbers with fractional parts, such as `3.142`. They can be expressed in either standard or scientific notation, meaning that the number `563.84` also can be expressed as `5.6384e2`.

Unlike integer literals, floating-point literals default to the `double` type, which is a 64-bit value. You have the option of using the smaller 32-bit `float` type if you know the full 64 bits are not required. You do this by appending an `f` or `F` to the end of the number, as in `5.6384e2f`. If you are a stickler for details, you also can explicitly state that you want a `double` type as the storage unit for your literal, as in `3.142d`. But because the default storage for floating-point numbers is `double` already, this addition isn't necessary.

#### **Construction Cue**



I mention floating-point literals because this primer is a general Java programming primer. However, it's important to note that floating-point data types and literals are not supported in the MIDP (Mobile Information Device Profile), which is the Java device profile used for mobile phones. In other words, you can't rely on the `double` or `float` types in Java mobile game code.

### Boolean Literals

Boolean literals are certainly welcome if you are coming from the world of C/C++. In C, there is no boolean type, and therefore no Boolean literals. The Boolean values `true` and `false` are represented by the integer values `1` and `0`. Java fixes this problem by providing a boolean type with two possible states: `true` and `false`. Not surprisingly, these states are represented in the Java language by the keywords `true` and `false`.

Boolean literals are used in Java programming about as often as integer literals because they are present in almost every type of control structure. Any time you have to represent a condition or state with two possible values, a Boolean is what you want. You learn a little more about the boolean type later in this primer. For now, just remember the two Boolean literal values: `true` and `false`.

## Character Literals

Character literals represent a single Unicode character and appear within a pair of single quotation marks. Special characters (control characters and characters that cannot be printed) are represented by a backslash (\) followed by the character code. A good example of a special character is `\n`, which forces the output to a new line when printed. Table A.2 shows the special characters supported by Java.

**TABLE A.2** Special Characters Supported by Java

Description	Representation
Backslash	<code>\\</code>
Continuation	<code>\</code>
Backspace	<code>\b</code>
Carriage return	<code>\r</code>
Form feed	<code>\f</code>
Horizontal tab	<code>\t</code>
Newline	<code>\n</code>
Single quote	<code>\'</code>
Double quote	<code>\"</code>
Unicode character	<code>\u0000</code>
Octal character	<code>\000</code>

An example of a Unicode character literal is `\u0048`, which is a hexadecimal representation of the character H. This same character is represented in octal as `\110`.

To find out more information about the Unicode character set, check out the Unicode website at <http://www.unicode.org/>.

**Construction  
Cue**



## String Literals

String literals represent multiple characters and appear within a pair of double quotation marks. Unlike all the other literals discussed in this primer, string literals are implemented in Java by the `String` class.

When Java encounters a string literal, it creates an instance of the `String` class and sets its state to the characters appearing within the double quotation marks. From a usage perspective, the fact that Java implements strings as objects is relatively unimportant. However, it is worth mentioning at this point because it is a reminder that Java is very object oriented in nature.

## Operators

Operators, also known as *operands*, specify an evaluation or computation to be performed on a data object or objects. These operands can be literals, variables, or function return types. The operators supported by Java follow:

```
+ - * / % & |
^ ~ && || ! < >
<= >= << >> >>> = ?
++ -- == += -= *= /=
%= &= |= ^= != <=< >=>
>>>= . [ ] ( )
```

Just seeing these operators probably doesn't help you a lot in determining how to use them. Don't worry—you'll learn a lot more about operators and how they are used a bit later in this primer.

## Separators

Separators are used to inform the Java compiler of how things are grouped in the code. For example, items in a list are separated by commas, much as lists of items appear in a sentence. The separators supported by Java follow:

```
{ } ; , :
```

## Comments and Whitespace

Earlier in this primer, you learned that comments and whitespace are removed by the Java compiler during the tokenization of the source code. You may be wondering, "What qualifies as whitespace and how are comments supported?" First, whitespace consists of spaces, tabs, and linefeeds. All occurrences of spaces, tabs, and linefeeds are removed by the Java compiler, as are comments. Comments can be defined in three different ways, as shown in Table A.3.

**TABLE A.3** Types of Comments Supported by Java

Type	Usage
<code>/* comment */</code>	All characters between <code>/*</code> and <code>*/</code> are ignored.
<code>// comment</code>	All characters after the <code>//</code> to the end of the line are ignored.
<code>/** comment */</code>	Same as <code>/* */</code> , except that the comment can be used with the javadoc tool to create automatic documentation.



The first type of comment (`/* comment */`) should be familiar if you have programmed in C. The compiler ignores all characters inside the `/*` and `*/` comment delimiters. The second type of comment (`// comment`) should also be familiar if you have used C++. The compiler ignores all characters appearing after the `//` comment delimiter up to the end of the line. These two comment types are borrowed from C and C++. The final comment type (`/** comment */`) works in the same fashion as the C-style comment type, with the additional benefit that it can be used with the Java documentation generator tool, `javadoc`, to create automatic documentation from the source code. Following are a few examples of using the various types of comments:

```
/* This is a C style comment. */  
// This is a C++ style comment.  
/** This is a javadoc style comment. */
```

## Data Types

One of the fundamental concepts of any programming language is *data types*. Data types define the storage methods available for representing information, along with how the information is interpreted. Data types are linked tightly to the storage of variables in memory because a variable's data type determines how the compiler interprets the contents of the memory.

To create a variable in memory, you must declare it by providing the type of the variable, as well as an identifier that uniquely identifies the variable. The syntax of the Java declaration statement for variables follows:

```
Type Identifier [, Identifier];
```

The declaration statement tells the compiler to set aside memory for a variable of type *Type* with the name *Identifier*. The optional bracketed *Identifier* indicates that you can make multiple declarations of the same type by separating them with commas. Finally, as in all Java statements, the declaration statement ends with a semicolon.

Java data types can be divided into two categories: simple and composite. Simple data types are core types not derived from any other types. Integer, floating-point, Boolean, and character types are all simple types. Composite types, on the other hand, are based on simple types and include strings, arrays, and both classes and interfaces in general. You learn about arrays later in this primer.

## Integer Data Types

Integer data types are used to represent signed integer numbers. There are four integer types: `byte`, `short`, `int`, and `long`. Each of these types takes up a different amount of space in memory, as shown in Table A.4.

**TABLE A.4** Java Integer Types

Type	Size
<code>byte</code>	8 bits
<code>short</code>	16 bits
<code>int</code>	32 bits
<code>long</code>	64 bits

To use the integer types to declare variables, use the declaration syntax mentioned previously with the desired type. Following are some examples of declaring integer variables:

```
int i;  
short rocketFuel;  
long angle, magnitude;  
byte red, green, blue;
```

## Floating-Point Data Types

Floating-point data types are used to represent numbers with fractional parts. There are two floating-point types: `float` and `double`. The `float` type reserves storage for a 32-bit single-precision number; the `double` type reserves storage for a 64-bit double-precision number.

Declaring floating-point variables is very similar to declaring integer variables. Following are some examples of floating-point variable declarations:

```
float temperature;  
double windSpeed, barometricPressure;
```

### Construction Cue



The `double` and `float` data types aren't supported in the MIDP, and therefore can't be used when developing mobile phone games in Java. This isn't too terribly bad of a restriction because floating-point calculations are typically too inefficient to use in game calculations.

## Boolean Data Type

The Boolean data type (`boolean`) is used to store values with one of two states: `true` or `false`. You can think of the `boolean` type as a 1-bit integer value (because 1 bit can have only two possible values: 1 or 0). However, rather than use 1 and 0, you use the Java keywords `true` and `false`. To declare a Boolean value, just use the `boolean` type declaration:

```
boolean gameOver;
```

## Character Data Type

The character data type is used to store single Unicode characters. Because the Unicode character set is composed of 16-bit values, the `char` data type is stored as a 16-bit unsigned integer. You create variables of type `char` as follows:

```
char firstInitial, lastInitial;
```

Remember that the `char` type is useful only for storing single characters. If you come from a C/C++ background, you may be tempted to fashion a string by creating an array of `chars`. In Java, this isn't necessary because the `String` class takes care of handling strings. This doesn't mean that you should never create arrays of characters, it just means that you shouldn't use a character array when you really want a string. C and C++ do not distinguish between character arrays and strings, but Java does.

## Blocks and Scope

In Java, source code is divided into parts separated by opening and closing curly braces: `{` and `}`. Everything between curly braces is considered a block and exists more or less independently of everything outside the braces. Blocks aren't important just from a logical sense—they are required as part of the syntax of the Java language. If you don't use braces, the compiler has trouble determining where one section of code ends and the next section begins. And from a purely aesthetic viewpoint, it is very difficult for you or someone else reading your code to understand what is going on if you don't use the braces.

Braces are used to group related statements together. You can think of everything between matching braces as being executed as one statement. In fact, from an outer block, that's exactly what an inner block appears like: a single statement. But what's a block? A *block* is simply a section of code. Blocks are organized in a hierarchical fashion, meaning that code can be divided into individual blocks nested under other blocks. One block can contain one or more nested subblocks.

It is standard Java programming style to identify different blocks with indentation. Every time you enter a new block, you should indent your source code by a number of spaces—preferably two. When you leave a block, you should move back, or unindent, two spaces. This is a fairly established convention in many programming languages. However, indentation is just a style issue and is not technically part of the language. The compiler produces identical output even if you don't indent anything. Indentation is used for the programmer, not the compiler; it simply makes the code easier to follow and understand. Following is an example of the proper indentation of blocks in Java:

```
for (int i = 0; i < 5; i++) {  
    if (i < 3) {  
        System.out.println(i);  
    }  
}
```

Following is the same code without any block indentations:

```
for (int i = 0; i < 5; i++) {  
if (i < 3) {  
System.out.println(i);  
}  
}
```

The first bit of code clearly shows the breakdown of program flow through the use of indentation; it is obvious that the `if` statement is nested within the `for` loop. The second bit of code, on the other hand, provides no visual clues about the relationship between the blocks of code. Don't worry if you don't know anything about `if` statements and `for` loops; you'll learn plenty about them later in this primer.

The concept of scope is tightly linked to blocks and is very important when you are working with variables in Java. *Scope* refers to how sections of a program (blocks) affect the lifetime of variables. Every variable declared in a program has an associated scope, meaning that the variable is used only in that particular part of the program.

Scope is determined by blocks. To better understand blocks, take a look again at the `HelloWorld` class earlier in this primer. The `HelloWorld` class is composed of two blocks. The outer block of the program is the block defining the `HelloWorld` class:

```
class HelloWorld {  
    ...  
}
```

Class blocks are very important in Java. Almost everything of interest is either a class or belongs to a class. For example, methods are defined inside the classes to which they belong. Both syntactically and logically, everything in Java takes place inside a class. Getting back to `HelloWorld`, the inner block defines the code within the `main()` method, as follows:

```
public static void main (String args[]) {  
    ...  
}
```

The inner block is considered to be nested within the outer block of the program. Any variables defined in the inner block are local to that block and are not visible to the outer block; the scope of the variables is defined as the inner block.

To get an even better idea behind the usage of scope and blocks, take a look at the `HowdyWorld` class:

```
class HowdyWorld {  
    public static void main (String args[]) {  
        int i;  
        printMessage();  
    }  
  
    public static void printMessage () {  
        int j;  
        System.out.println("Howdy, World!");  
    }  
}
```

The `HowdyWorld` class contains two methods: `main()` and `printMessage()`. `main()` should be familiar to you from the `HelloWorld` class, except that in this case, it declares an integer variable `i` and calls the `printMessage()` method. `printMessage()` is a new method that declares an integer variable `j` and prints the message `Howdy, World!` to the standard output stream, much as the `main()` method does in `HelloWorld`.

You've probably figured out already that `HowdyWorld` results in the same output as `HelloWorld` because the call to `printMessage()` results in a single text message being displayed. What you may not see right off is the scope of the integers defined in each method. The integer `i` defined in `main()` has a scope limited to the body of the `main()` method. The body of `main()` is defined by the curly braces around the method (the method block). Similarly, the integer `j` has a scope limited to the body of the `printMessage()` method. The importance of the scope of these two variables is that the variables aren't visible beyond their respective scopes; the `HowdyWorld` class block knows nothing about the two integers. Furthermore, `main()` doesn't know anything about `j`, and `printMessage()` knows nothing about `i`.

Scope becomes more important when you start nesting blocks of code within other blocks. The following `GoodbyeWorld` class is a good example of variables nested within different scopes:

```
class GoodbyeWorld {
    public static void main (String args[]) {
        int i, j;
        System.out.println("Goodbye, World!");
        for (i = 0; i < 5; i++) {
            int k;
            System.out.println("Bye!");
        }
    }
}
```

The integers `i` and `j` have scopes within the `main()` method body. The integer `k`, however, has a scope limited to the `for` loop block. Because `k`'s scope is limited to the `for` loop block, it cannot be seen outside that block. On the other hand, `i` and `j` still can be seen within the `for` loop block. What this means is that scoping has a top-down hierarchical effect—variables defined in outer scopes can still be seen and used within nested scopes; however, variables defined in nested scopes are limited to those scopes.

For more reasons than visibility, it is important to pay attention to the scope of variables when you declare them. Along with determining the visibility of variables, the scope also determines the lifetime of variables. This means that variables are actually destroyed when program execution leaves their scope. Look at the `GoodbyeWorld` example again: Storage for the integers `i` and `j` is allocated when program execution enters the `main()` method. When the `for` loop block is entered, storage for the integer `k` is allocated. When program execution leaves the `for` loop block, the memory for `k` is freed and the variable is destroyed. Similarly, when program execution leaves `main()`, all the variables in its scope (`i` and `j`) are freed and destroyed. The concepts of variable lifetime and scope become even more important when you start dealing with classes.

## Arrays

An array is a construct that provides for the storage of a list of items of the same type. Array items can be either a simple or composite data type. Arrays also can be multidimensional. Java arrays are declared with square brackets: `[]`. Following are a few examples of array declarations in Java:

```
int numbers[];
char[] letters;
long grid[][];
```

If you are familiar with arrays in another language, you may be puzzled by the absence of a number between the square brackets specifying the number of items in the array. Java doesn't allow you to specify the size of an empty array when declaring the array. You always must explicitly set the size of the array with the new operator or by assigning a list of items to the array at the time of creation.

Another strange thing you may notice about Java arrays is the optional placement of the square brackets in the array declaration. You can place the square brackets either after the variable type or after the identifier.

Following are a couple examples of arrays that have been declared and set to a specific size; the new operator was used and a list of items in the array declaration was assigned:

```
char alphabet[] = new char[26];  
int primes = {7, 11, 13};
```

More complex structures for storing lists of items, such as stacks and hash tables, are also supported by Java. Unlike arrays, these structures are implemented in Java as classes.

## Strings

In Java, strings are handled by a special class called `String`. Even literal strings are managed internally by an instantiation of a `String` class. An instantiation of a class is simply an object that has been created based on the class description. Following are a few strings declared with the Java `String` class:

```
String message;  
String name = "Mr. Blonde";
```

## Expressions and Operators

After you create variables, you typically want to do something with them. Operators enable you to perform an evaluation or computation on a data object or objects. Operators applied to variables and literals form expressions. An expression can be thought of as a programmatic equation. More formally, an expression is a sequence of one or more data objects (operands) and zero or more operators that produce a result. An example of an expression follows:

```
x = y / 3;
```

In this expression, *x* and *y* are variables, 3 is a literal, and = and / are operators. This expression states that the *y* variable is divided by 3 using the division operator (/), and using the assignment operator (=) is used to store the result in *x*. Notice that the expression was described from right to left. Although this approach of analyzing the expression from right to left is useful in terms of showing the assignment operation, most Java expressions are, in fact, evaluated from left to right. You get a better feel for this in the next section.

## Operator Precedence

Even though Java expressions are typically evaluated from left to right, there still are many times when the result of an expression would be indeterminate without other rules. The following expression illustrates the problem:

```
x = 2 * 6 + 16 / 4;
```

Strictly using the left-to-right evaluation of the expression, the multiplication operation  $2 * 6$  is carried out first, which leaves a result of 12. The addition operation  $12 + 16$  is then performed, which gives a result of 28. The division operation  $28 / 4$  is then performed, which gives a result of 7. Finally, the assignment operation  $x = 7$  is handled, in which the number 7 is assigned to the variable *x*.

If you have some experience with operator precedence from another language, you might already be questioning the evaluation of this expression, and for good reason—it's wrong! The problem is that using a simple left-to-right evaluation of expressions can yield inconsistent results, depending on the order of the operators. The solution to this problem lies in *operator precedence*, which determines the order in which operators are evaluated. Every Java operator has an associated precedence. Following is a list of all the Java operators from highest to lowest precedence. In this list of operators, all the operators in a particular row have equal precedence. The precedence level of each row decreases from top to bottom. This means that the [] operator has a higher precedence than the \* operator, but the same precedence as the () operator.

```
. [] ()
++ -- ! ~
* / %
+ -
<< >> >>>
< > <= >=
== !=
&
^
&&
||
?:
=
```



Evaluation of expressions still moves from left to right, but only when dealing with operators that have the same precedence. Otherwise, operators with a higher precedence are evaluated before operators with a lower precedence. Knowing this, take another look at the sample equation:

```
x = 2 * 6 + 16 / 4;
```

Before using the left-to-right evaluation of the expression, first look to see whether any of the operators have differing precedence. Indeed they do! The multiplication (\*) and division (/) operators both have the highest precedence, followed by the addition operator (+), and then the assignment operator (=). Because the multiplication and division operators share the same precedence, evaluate them from left to right. Doing this, you first perform the multiplication operation  $2 * 6$  with the result of 12. You then perform the division operation  $16 / 4$ , which results in 4. After performing these two operations, the expression looks like this:

```
x = 12 + 4;
```

Because the addition operator has a higher precedence than the assignment operator, you perform the addition operation  $12 + 4$  next, resulting in 16. Finally, the assignment operation  $x = 16$  is processed, resulting in the number 16 being assigned to the variable  $x$ . As you can see, evaluating the expression using operator precedence yields a completely different result.

Just to get the point across, take a look at another expression that uses parentheses for grouping purposes:

```
x = 2 * (11 - 7);
```

Without the grouping parentheses, you would perform the multiplication operation first and then the subtraction operation. However, referring back to the precedence list, the () operator comes before all other operators. So the subtraction operation  $11 - 7$  is performed first, yielding 4 and the following expression:

```
x = 2 * 4;
```

The rest of the expression is easily resolved with a multiplication and an assignment to yield a result of 8 in the variable  $x$ .

## Integer Operators

Three types of operations can be performed on integers: unary, binary, and relational. Unary operators act only on single integer numbers, and binary operators act on pairs of integer numbers. Both unary and binary integer operators typically return integer results. Relational operators, on the other hand, act on two integer numbers but return a Boolean result rather than an integer.

Unary and binary integer operators typically return an `int` type. For all operations involving the types `byte`, `short`, and `int`, the result is always an `int`. The only exception to this rule is when one of the operands is a `long`, in which case the result of the operation is also of type `long`.

Unary Integer Operators

Unary integer operators act on a single integer. Table A.5 lists the unary integer operators.

TABLE A.5 The Unary Integer Operators

Description	Operator
Increment	++
Decrement	--
Negation	-
Bitwise complement	~

The increment and decrement operators (`++` and `--`) increase and decrease integer variables by 1. These operators can be used in either prefix or postfix form. A prefix operator takes effect before the evaluation of the expression it is in; a postfix operator takes effect after the expression has been evaluated. Prefix unary operators are placed immediately before the variable; postfix unary operators are placed immediately following the variable. Following are examples of each type of operator:

```
y = ++x;
z = x--;
```

In the first example, `x` is prefix incremented, which means that it is incremented before being assigned to `y`. In the second example, `x` is postfix decremented, which means that it is decremented after being assigned to `z`. In the latter case, `z` is assigned the value of `x` before `x` is decremented.

The negation unary integer operator (`-`) is used to change the sign of an integer value. This operator is as simple as it sounds, as indicated by the following example:

```
x = 8;
y = -x;
```

In this example, `x` is assigned the literal value 8 and then is negated and assigned to `y`. The resulting value of `y` is -8.

The last Java unary integer operator is the bitwise complement operator (~), which performs a bitwise negation of an integer value. Bitwise negation means that each bit in the number is toggled. In other words, all the binary 0s become 1s and all the binary 1s become 0s. Take a look at an example very similar to the one for the negation operator:

```
x = 8;
y = ~x;
```

In this example, *x* is assigned the literal value 8 again, but it is bitwise complemented before being assigned to *y*. What does this mean? Well, without getting into the details of how integers are stored in memory, it means that all the bits of the variable *x* are flipped, yielding a decimal result of -9. This result has to do with the fact that negative numbers are stored in memory by means of a method known as *two's complement* (see the following note).

Integer numbers are stored in memory as a series of binary bits that can each have a value of 0 or 1. A number is considered negative if the highest-order bit in the number is set to 1. Because a bitwise complement flips all the bits in a number—including the high-order bit—the sign of a number is reversed.

**Construction Cue**



## Binary Integer Operators

Binary integer operators act on pairs of integers. Table A.6 lists the binary integer operators.

**TABLE A.6** The Binary Integer Operators

Description	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Bitwise AND	&
Bitwise OR	
Bitwise XOR	^
Left-shift	<<
Right-shift	>>
Zero-fill-right-shift	>>>

The addition, subtraction, multiplication, and division operators (+, -, \*, and /) all do what you expect them to. An important thing to note is how the division operator works; because you are dealing with integer operands, the division operator returns an integer divisor. In cases where the division results in a remainder, the modulus operator (%) can be used to get the remainder value.

Mathematically, a division by zero results in an infinite result. Because representing infinite numbers is a big problem for computers, division or modulus operations by zero result in an error. To be more specific, a runtime exception is thrown.

The bitwise AND, OR, and XOR operators (&, |, and ^) all act on the individual bits of an integer. These operators are sometimes useful when an integer is being used as a bit field. An example of this is when an integer is used to represent a group of binary flags. An `int` is capable of representing up to 32 different flags because it is stored in 32 bits.

The left-shift, right-shift, and zero-fill-right-shift operators (<<, >>, and >>>) shift the individual bits of an integer by a specified integer amount. Following are some examples of how these operators are used:

```
x << 3;  
y >> 7;  
z >>> 2;
```

In the first example, the individual bits of the integer variable `x` are shifted to the left three places. In the second example, the bits of `y` are shifted to the right seven places. Finally, the third example shows `z` being shifted to the right two places, with zeros shifted into the two leftmost places.

Based on these examples, you may be wondering what the difference is between the right-shift (>>) and zero-fill-right-shift (>>>) operators. The right-shift operator appears to shift zeros into the leftmost bits, just like the zero-fill-right-shift operator, right? Well, when dealing with positive numbers, there is no difference between the two operators; they both shift zeros into the upper bits of a number. The difference arises when you start shifting negative numbers. Remember that negative numbers have the high-order bit set to 1. The right-shift operator preserves the high-order bit and effectively shifts the lower 31 bits to the right. This behavior yields results for negative numbers similar to those for positive numbers. That is, -8 shifted right by one results in -4. The zero-fill-right-shift operator, on the other hand, shifts zeros into all the upper bits, including the high-order bit. When this shifting is applied to negative numbers, the high-order bit becomes 0 and the number becomes positive.

## Relational Integer Operators

The last group of integer operators is the relational operators, which all operate on integers but return a type `boolean`. Table A.7 lists the relational integer operators.

**TABLE A.7** The Relational Integer Operators

Description	Operator
Less-than	<
Greater-than	>
Less-than-or-equal-to	<=
Greater-than-or-equal-to	>=
Equal-to	==
Not-equal-to	!=

These operators all perform comparisons between integers.

## Floating-Point Operators

Similar to integer operators, three types of operations can be performed on floating-point numbers: unary, binary, and relational. Unary operators act only on single floating-point numbers, and binary operators act on pairs of floating-point numbers. Both unary and binary floating-point operators return floating-point results. Relational operators, however, act on two floating-point numbers but return a `Boolean` result.

Unary and binary floating-point operators return a `float` type if both operands are of type `float`. If one or both of the operands is of type `double`, however, the result of the operation is of type `double`. Because the MIDP doesn't support floating-point data types, I'll save you the trouble of learning all the details about them. It suffices to say that they are very similar to the integer operators you just saw.

## Boolean Operators

Boolean operators act on `boolean` types and return a `Boolean` result. The `Boolean` operators are listed in Table A.8.

**TABLE A.8** The Boolean Operators

Description	Operator
Evaluation AND	&
Evaluation OR	
Evaluation XOR	^
Logical AND	&&
Logical OR	
Negation	!
Equal-to	==
Not-equal-to	!=
Conditional	?:

The evaluation operators (&, |, and ^) evaluate both sides of an expression before determining the result. The logical operators (&& and ||) avoid the right-side evaluation of the expression if it is not needed. To better understand the difference between these operators, take a look at the following two expressions:

```
boolean result = isValid & (Count > 10);
boolean result = isValid && (Count > 10);
```

The first expression uses the evaluation AND operator (&) to make an assignment. In this case, both sides of the expression are always evaluated, regardless of the values of the variables involved. In the second example, the logical AND operator (&&) is used. This time, the isValid Boolean value is first checked. If it is false, the right side of the expression is ignored and the assignment is made. This operator is more efficient because a false value on the left side of the expression provides enough information to determine the false outcome.

Although the logical operators are more efficient than the evaluation operators, there still may be times when you want to use the evaluation operators to ensure that the entire expression is evaluated. The following code shows how the evaluation AND operator is necessary for the complete evaluation of an expression:

```
while ((++x < 10) && (++y < 15)) {
    System.out.println(x);
    System.out.println(y);
}
```

In this example, the second expression (++y < 15) is evaluated after the last pass through the loop because of the evaluation AND operator. If the logical AND operator had been used, the second expression would not have been evaluated and y would not have been incremented after the last time around.

The three Boolean operators—negation, equal-to, and not-equal-to (!, ==, and !=)—perform exactly as you might expect. The negation operator toggles the value of a Boolean from false to true or from true to false, depending on the original value. The equal-to operator simply determines whether two Boolean values are equal (both true or both false). Similarly, the not-equal-to operator determines whether two Boolean operands are unequal.

The conditional Boolean operator (?:) is the most unique of the Boolean operators and is worth a closer look. This operator also is known as the *ternary operator* because it takes three items: a condition and two expressions. The syntax for the conditional operator follows:

*Condition ? Expression1 : Expression2*

The *Condition*, which is a Boolean, is first evaluated to determine whether it is true or false. If *Condition* evaluates to a true result, *Expression1* is evaluated. If *Condition* ends up being false, *Expression2* is evaluated.

## String Operators

Similar to other data types, strings can be manipulated with operators. Actually, there is only one string operator: the concatenation operator (+). The concatenation operator for strings works very much like the addition operator for numbers—it adds strings together.

## Assignment Operators

One final group of operators you haven't seen yet is the assignment operators. Assignment operators actually work with all the fundamental data types. Table A.9 lists the assignment operators.

**TABLE A.9** The Assignment Operators

Description	Operator
Simple	=
Addition	+=
Subtraction	-=
Multiplication	*=
Division	/=
Modulus	%=
AND	&=
OR	=
XOR	^=

With the exception of the simple assignment operator (`=`), the assignment operators function exactly like their nonassignment counterparts, except that the resulting value is stored in the operand on the left side of the expression. Take a look at the following examples:

```
x += 6;  
x *= (y - 3);
```

In the first example, `x` and 6 are added and the result stored in `x`. In the second example, 3 is subtracted from `y` and the result multiplied by `x`. The final result is then stored in `x`.

## Control Structures

Although performing operations on data is very useful, it's time to move on to the issue of program flow control. The flow of your programs is dictated by two different types of constructs: branches and loops. Branches enable you to selectively execute one part of a program instead of another. Loops, on the other hand, provide a means to repeat certain parts of a program. Together, branches and loops provide you with a powerful means of controlling the logic and execution of your code.

### Branches

Without branches or loops, Java code executes in a sequential fashion, which means that each statement is executed one after the next. But what if you don't always want every single statement executed? Then you use a branch, which gives the flow of your code more options. The concept of branches might seem trivial, but it would be difficult if not impossible to write useful programs without them. Java supports two types of branches: `if-else` branches and `switch` branches.

#### `if-else` Branch

The `if-else` branch is the most commonly used branch in Java programming. It is used to select conditionally one of two possible outcomes. The syntax for the `if-else` statement follows:

```
if (Condition)  
    Statement1  
else  
    Statement2
```



If the Boolean *Condition* evaluates to true, *Statement1* is executed. Likewise, if *Condition* evaluates to false, *Statement2* is executed. The following example shows how to use an if-else statement:

```
if (isTired)
    timeToEat = true;
else
    timeToEat = false;
```

If the Boolean variable `isTired` is true, the first statement is executed and `timeToEat` is set to true. Otherwise, the second statement is executed and `timeToEat` is set to false. You may have noticed that the if-else branch works in a manner very similar to the conditional operator (`?:`) described earlier in this primer. In fact, you can think of the if-else branch as an expanded version of the conditional operator. One significant difference between the two is that you can include compound statements in an if-else branch, which you cannot do with the conditional operator.

Compound statements are blocks of code surrounded by curly braces `{}` that appear as a single, or simple, statement to an outer block of code.

**Construction**  
**Cue**



If you have only a single statement that you want to execute conditionally, you can leave off the else part of the branch, as shown in the following example:

```
if (isThirsty)
    pourADrink = true;
```

On the other hand, if you need more than two conditional outcomes, you can string together a series of if-else branches to get the desired effect. The following example shows multiple if-else branches used to choose between different outcomes:

```
if (x == 0)
    y = 5;
else if (x == 2)
    y = 25;
else if (x >= 3)
    y = 125;
```

In this example, three different comparisons are made, each with its own statement executed on a true conditional result. Notice, however, that subsequent if-else branches are in effect nested within the previous branch. This arrangement ensures that at most one statement is executed.

The last important topic to cover in regard to `if-else` branches is compound statements. As mentioned in the preceding note, a *compound statement* is a block of code surrounded by curly braces that appears to an outer block as a single statement. Following is an example of a compound statement used with an `if` branch:

```
if (performCalc) {  
    x += y * 5;  
    y -= 10;  
    z = (x - 3) / y;  
}
```

Sometimes, when nesting `if-else` branches, it is necessary to use curly braces to distinguish which statements go with which branch. The following example illustrates the problem:

```
if (x != 0)  
    if (y < 10)  
        z = 5;  
else  
    z = 7;
```

In this example, the style of indentation indicates that the `else` branch belongs to the first (outer) `if`. However, because there was no grouping specified, the Java compiler assumes that the `else` goes with the inner `if`. To get the desired results, you must modify the code as follows:

```
if (x != 0) {  
    if (y < 10)  
        z = 5;  
}  
else  
    z = 7;
```

The addition of the curly braces tells the compiler that the inner `if` is part of a compound statement; more importantly, it completely hides the `else` branch from the inner `if`. You can see that code within the inner `if` has no way of accessing code outside its scope, including the `else` branch.

### switch **Branch**

Similar to the `if-else` branch, the `switch` branch specifically is designed to conditionally choose between multiple outcomes. The syntax for the `switch` statement follows:

```
switch (Expression) {  
    case Constant1:  
        StatementList1  
    case Constant2:  
        StatementList2  
    —
```

```

default:
    DefaultStatementList
}

```

The switch branch evaluates and compares *Expression* to all the case constants and branches the program's execution to the matching case statement list. If no case constants match *Expression*, the program branches to the *DefaultStatementList*, if one has been supplied (the *DefaultStatementList* is optional). You might be wondering what a statement list is. A *statement list* is simply a series, or list, of statements. Unlike the if-else branch, which directs program flow to a simple or compound statement, the switch branch directs the flow to a list of statements. When the program execution moves into a case statement list, it continues from there in a sequential manner.

## Loops

When it comes to program flow, branches really tell only half of the story; loops tell the other half. Put simply, loops enable you to execute code repeatedly. Just as branches alter the sequential flow of programs, so do loops. There are three types of loops in Java: for loops, while loops, and do-while loops.

### for Loop

The for loop provides a means to repeat a section of code a designated number of times. The for loop is structured so that a section of code is repeated until some limit has been reached. The syntax for the for statement follows:

```

for (InitializationExpression; LoopCondition; StepExpression)
    Statement

```

The for loop repeats the *Statement* the number of times determined by the *InitializationExpression*, the *LoopCondition*, and the *StepExpression*. The *InitializationExpression* is used to initialize a loop control variable. The *LoopCondition* compares the loop control variable to some limit value. Finally, the *StepExpression* specifies how the loop control variable should be modified before the next iteration of the loop. The following example shows how a for loop can be used to print the numbers from 1 to 10:

```

for (int i = 1; i < 11; i++)
    System.out.println(i);

```

First, *i* is declared as an integer. The fact that *i* is declared within the body of the for loop might look strange to you at this point. Don't despair—this is completely legal. *i* is initialized to 1 in the *InitializationExpression* part of the for loop.

Next, the conditional expression `i < 11` is evaluated to see whether the loop should continue. At this point, `i` is still equal to 1, so *LoopCondition* evaluates to true and the *Statement* is executed (the value of `i` is printed to standard output). `i` is then incremented in the *StepExpression* part of the for loop, and the process repeats with the evaluation of *LoopCondition* again. This continues until *LoopCondition* evaluates to false, which is when `x` equals 11 (10 iterations later).

### while **Loop**

Like the for loop, the while loop has a loop condition that controls the execution of the loop statement. Unlike the for loop, however, the while loop has no initialization or step expressions. The syntax for the while statement follows:

```
while (LoopCondition)
    Statement
```

If the Boolean *LoopCondition* evaluates to true, the *Statement* is executed and the process starts over. It is important to understand that the while loop has no step expression as the for loop does. This means that the *LoopCondition* must somehow be changed by code in the *Statement* or the loop will infinitely repeat, which is a bad thing. An infinite loop causes a program to never exit, which hogs processor time and can ultimately hang the system.

Another important thing to notice about the while loop is that its *LoopCondition* occurs before the body of the loop *Statement*. This means that if the *LoopCondition* initially evaluates to false, the *Statement* is never executed. Although this may seem trivial, it is in fact the only thing that differentiates the while loop from the do-while loop, which is discussed next.

### do-while **Loop**

The do-while loop is very similar to the while loop, as you can see in the following syntax:

```
do
    Statement
while (LoopCondition);
```

The major difference between the do-while loop and the while loop is that, in a do-while loop, the *LoopCondition* is evaluated after the *Statement* is executed. This difference is important because there may be times when you want the *Statement* code to be executed at least once, regardless of the *LoopCondition*.

The *Statement* is executed initially, and from then on it is executed as long as the *LoopCondition* evaluates to true. As with the while loop, you must be careful with the do-while loop to avoid creating an infinite loop. An infinite loop occurs when the *LoopCondition* remains true indefinitely. The following example shows a very obvious infinite do-while loop:

```
do
    System.out.println("I'm stuck!");
while (true);
```

Because the *LoopCondition* is always true, the message “I’m Stuck!” is printed forever.

In some situations where a program repeatedly runs a block of code until exiting, it is okay to create an infinite while loop. In fact, such a loop is used to establish a game cycle in the example games throughout this book. However, the code within the infinite loop is specifically designed to allow the program to exit, which in effect overrides the loop and prevents the system from hanging.

**Construction**  
**Cue**



## Object-Oriented Programming and Java

Java is an object-oriented programming (OOP) language. Although you don’t need to be an OOP wizard to develop mobile games in Java, you do need to know some ground rules. First off, *objects* are software bundles of data and the procedures that act on that data. The procedures are also known as *methods*. The merger of data and methods provides a means of more accurately representing real-world objects in software.

To understand how software objects are beneficial, think about the common characteristics of all real-world objects. Lions, cars, and calculators all share two common characteristics: state and behavior. For example, the state of a lion includes color, weight, and whether the lion is tired or hungry. Lions also have certain behaviors, such as roaring, sleeping, and hunting. Similarly, the state of a car includes the current speed, the type of transmission, whether it is two-wheel or four-wheel drive, whether the lights are on, and the current gear, among other things. The behaviors for a car include turning, braking, and accelerating.

As with real-world objects, software objects also have these two common characteristics (state and behavior). To relate this back to programming terms, an object’s state is determined by its data; an object’s behavior is defined by its methods. By making this connection between real-world objects and software objects, you begin to see how objects help bridge the gap between the real world and the world of software inside your computer.

A *class* is a template or prototype that defines a type of object. A class is to an object what a blueprint is to a house. Many houses may be built from a single blueprint; the blueprint outlines the makeup of the houses. Classes work exactly the same way, except that they outline the makeup of objects.

In object-oriented terms, you would say that your house is a specific instance of the class of objects known as houses. All houses have states and behaviors in common that define them as houses. When builders start building a new neighborhood of houses, they typically build them all from a set of blueprints. It wouldn't be as efficient to create a new blueprint for every single house, especially when so many similarities are shared between each one. What it boils down to is that classes are software blueprints for objects.

What happens if you want an object that is very similar to one you already have, but that has a few extra characteristics? You just inherit a new class based on the class of the similar object. *Inheritance* is the process of creating a new class with the characteristics of an existing class, along with additional characteristics unique to the new class. Inheritance provides a powerful and natural mechanism for organizing and structuring programs.

When a class is based on another class, it inherits all the properties of that class, including the data and methods for the class. The class doing the inheriting is referred to as the *subclass* (or the *child class*), and the class providing the information to inherit is referred to as the *superclass* (or the *parent class*). Using the car example, child classes could be inherited from the car class for gas-powered cars and cars powered by electricity. Both new car classes share common "car" characteristics, but they also add a few characteristics of their own. The gas car would add, among other things, a fuel tank and a gas cap; the electric car might add a battery and a plug for recharging. Each subclass inherits state information (in the form of variable declarations) from the superclass.

## Working with Java Classes

In Java, all classes are subclassed from a standard superclass called `Object`. In other words, `Object` serves as the superclass for all Java classes, including the classes that make up the Java API as well as custom classes that you create.

### Declaring Classes

The syntax for declaring classes in Java follows:

```
class Identifier {  
    ClassBody  
}
```

*Identifier* specifies the name of the new class, which is by default derived from `Object`. The curly braces surround the body of the class, *ClassBody*. As an example, take a look at the class declaration for an `Alien` class, which could be used in a space game:

```
class Alien {
    Color color;
    int energy;
    int aggression;
}
```

The standard Java `Color` object isn't supported in the MIDP API, so you won't find it used in mobile Java games. However, for the purposes of learning the basics of Java, the `Color` class is useful in the context of the `Alien` example class. In a mobile game based on the MIDP API, you would probably use three separate integer values that represent the individual red, green, and blue components of a color.

### **Construction Cue**



The state of the `Alien` object is defined by three data members, which represent the color, energy, and aggression of the alien. It's important to point out that the `Alien` class is inherently derived from `Object` even though the `Object` class isn't specifically mentioned. So far, the `Alien` class isn't all that useful; it needs some methods. The most basic syntax for declaring methods for a class follows:

```
ReturnType Identifier(Parameters) {
    MethodBody
}
```

*ReturnType* specifies the data type that the method returns, *Identifier* specifies the name of the method, and *Parameters* specifies the parameters to the method, if there are any. As with class bodies, the body of a method, *MethodBody*, is enclosed by curly braces. Following is a method declaration for the `morph()` method, which would be useful in the `Alien` class because some aliens might like to change shape:

```
void morph(int aggression) {
    if (aggression < 10) {
        // morph into a smaller size
    }
    else if (aggression < 20) {
        // morph into a medium size
    }
    else {
        // morph into a giant size
    }
}
```

The `morph()` method is passed an integer as the only parameter, `aggression`. This value is then used to determine the size to which the alien is morphing. As you can see, the alien morphs to smaller or larger sizes based on its aggression.

If you make the `morph()` method a member of the `Alien` class, it is readily apparent that the `aggression` parameter isn't necessary. The `aggression` parameter is already a member variable of `Alien`, to which all class methods have access. The `Alien` class, with the addition of the `morph()` method, looks like this:

```
class Alien {
    Color color;
    int energy;
    int aggression;

    void morph() {
        if (aggression < 10) {
            // morph into a smaller size
        }
        else if (aggression < 20) {
            // morph into a medium size
        }
        else {
            // morph into a giant size
        }
    }
}
```

## Deriving Classes

So far, the discussion of class declaration has been limited to creating new classes inherently derived from `Object`. Deriving all your classes from `Object` isn't a very good idea because you would have to redefine the data and methods for each class. The way you derive classes from classes other than `Object` is by using the `extends` keyword. The syntax for deriving a class using the `extends` keyword follows:

```
class Identifier extends SuperClass {
    ClassBody
}
```

*Identifier* refers to the name of the newly derived class, *SuperClass* refers to the name of the class from which you are deriving, and *ClassBody* is the new class body.

Let's use the `Alien` class introduced in the preceding section as the basis for a derivation example. What if you had an `Enemy` class that defined general information useful for all enemies? You would no doubt want to go back and derive the `Alien` class from the new `Enemy` class to take advantage of the standard enemy functionality provided by the `Enemy` class. Following is the `Enemy`-derived `Alien` class that uses the `extends` keyword:



```
class Alien extends Enemy {
    Color color;
    int energy;
    int aggression;

    void morph() {
        if (aggression < 10) {
            // morph into a smaller size
        }
        else if (aggression < 20) {
            // morph into a medium size
        }
        else {
            // morph into a giant size
        }
    }
}
```

This declaration assumes that the `Enemy` class declaration is readily available in the same package as `Alien`. In reality, you will likely derive from classes in a lot of different places. To derive a class from an external superclass, you must first use the `import` statement to import the superclass.

You'll get to packages a little later in this primer. For now, just think of a package as a group of related classes.

**Construction**  
**Cue**



If you had to import the `Enemy` class, you would do so like this:

```
import Enemy;
```

## Overriding Methods

At times it is useful to override methods in derived classes. For example, if the `Enemy` class had a `move()` method, you would want the movement to vary based on the type of enemy. Some types of enemies may fly around in specified patterns, whereas other enemies may crawl in a random fashion. To allow the `Alien` class to exhibit its own movement, you would override the `move()` method with a version specific to alien movement. The `Enemy` class would then look something like this:

```
class Enemy {
    ...
    void move() {
        // move the enemy
    }
}
```

Likewise, the Alien class with the overridden `move()` method would look something like this:

```
class Alien {
    Color color;
    int energy;
    int aggression;

    void move() {
        // move the alien
    }

    void morph() {
        if (aggression < 10) {
            // morph into a smaller size
        }
        else if (aggression < 20) {
            // morph into a medium size
        }
        else {
            // morph into a giant size
        }
    }
}
```

When you create an instance of the Alien class and call the `move()` method, the new `move()` method in Alien is executed rather than the original overridden `move()` method in Enemy. Method overriding is a simple yet powerful usage of object-oriented design.

## Overloading Methods

Another powerful object-oriented technique is method overloading. Method overloading enables you to specify different types of information (parameters) to send to a method. To overload a method, you declare another version with the same name but different parameters.

For example, the `move()` method for the Alien class could have two different versions: one for general movement and one for moving to a specific location. The general version is the one you've already defined: It moves the alien based on its current state. The declaration for this version follows:

```
void move() {
    // move the alien
}
```

To enable the alien to move to a specific location, you overload the `move()` method with a version that takes `x` and `y` parameters, which specify the location to move to. The overloaded version of `move()` follows:

```
void move(int x, int y) {  
    // move the alien to position x,y  
}
```

Notice that the only difference between the two methods is the parameter lists; the first `move()` method takes no parameters; the second `move()` method takes two integers.

You may be wondering how the compiler knows which method is being called in a program, when they both have the same name. The compiler keeps up with the parameters for each method along with the name. When a call to a method is encountered in a program, the compiler checks the name and the parameters to determine which overloaded method is being called. In this case, calls to the `move()` methods are easily distinguishable by the absence or presence of the `int` parameters.

## Access Modifiers

Access to variables and methods in Java classes is accomplished through access modifiers. Access modifiers define varying levels of access between class members and the outside world (other objects). Access modifiers are declared immediately before the type of a member variable or the return type of a method. There are four access modifiers: the default access modifier, `public`, `protected`, and `private`.

Access modifiers affect not only the visibility of class members, but also of classes themselves. However, class visibility is tightly linked with packages, which are covered later in this primer.

### The Default Access Modifier

The default access modifier specifies that only classes in the same package can have access to a class's variables and methods. Class members with default access have a visibility limited to other classes within the same package. There is no actual keyword for declaring the default access modifier; it is applied by default in the absence of an explicit access modifier. For example, the `Alien` class members all have default access because no access modifiers were specified. Examples of a default access member variable and method follow:

```
long length;  
void getLength() {  
    return length;  
}
```

Notice that neither the member variable nor the method supplies an access modifier, so they take on the default access modifier implicitly.

### **The public Access Modifier**

The public access modifier specifies that class variables and methods are accessible to anyone, both inside and outside the class. This means that public class members have global visibility and can be accessed by any other objects. Some examples of public member variables follow:

```
public int count;  
public boolean isActive;
```

### **The protected Access Modifier**

The protected access modifier specifies that class members are accessible only to methods in that class and subclasses of that class. This means that protected class members have visibility limited to subclasses. Examples of a protected variable and a protected method follow:

```
protected char middleInitial;  
protected char getMiddleInitial() {  
    return middleInitial;  
}
```

### **The private Access Modifier**

The private access modifier is the most restrictive; it specifies that class members are accessible only by the class in which they are defined. This means that no other class has access to private class members, even subclasses. Some examples of private member variables follow:

```
private String firstName;  
private double howBigIsIt;
```

### **The static Modifier**

At times you need a common variable or method for all objects of a particular class. The static modifier specifies that a variable or method is the same for all objects of a particular class. In other words, there is only one value assigned to the variable no matter how many objects of the class are created.

Typically, new variables are allocated for each instance of a class. When a variable is declared as being `static`, it is allocated only once regardless of how many objects are instantiated. The result is that all instantiated objects share the same instance of the static variable. Similarly, a static method is one whose implementation is exactly the same for all objects of a particular class. This means that static methods have access only to static variables.

Following are some examples of a static member variable and a static method:

```
static int refCount;
static int getRefCount() {
    return refCount;
}
```

A beneficial side effect of static members is that you can access them without having to create an instance of a class. Remember the `System.out.println()` method used earlier in this primer? `out` is a static member variable of the `System` class, which means that you can access it without having to actually instantiate a `System` object.

## The final Modifier

Another useful modifier in regard to controlling class member usage is the `final` modifier. The `final` modifier specifies that a variable has a constant value or that a method cannot be overridden in a subclass. To think of the `final` modifier literally, it means that a class member is the final version allowed for the class.

Following are some examples of `final` member variables:

```
final public int numDollars = 25;
final boolean amIBroke = false;
```

If you are coming from the world of C++, `final` variables may sound familiar. In fact, `final` variables in Java are very similar to `const` variables in C++; they must always be initialized at declaration and their value can't change any time afterward.

## Abstract Classes and Methods

An abstract class is a class that is partially implemented and whose purpose is solely as a design convenience. Abstract classes are made up of one or more abstract methods, which are methods that are declared but left bodiless (unimplemented).

The `Enemy` class discussed earlier is an ideal candidate to become an abstract class. You would probably never want to actually create an `Enemy` object because it is too general. However, the `Enemy` class serves a very logical purpose as a superclass for more-specific enemy classes, such as the `Alien` class. To turn the `Enemy` class into an abstract class, you use the `abstract` keyword, like this:

```
abstract class Enemy {  
    abstract void move();  
    abstract void move(int x, int y);  
}
```

Notice the usage of the `abstract` keyword before the class declaration for `Enemy`. This tells the compiler that the `Enemy` class is abstract. Also notice that both `move()` methods are declared as being abstract. Because it isn't clear how to move a generic enemy, the `move()` methods in `Enemy` have been left unimplemented (abstract).

You should be aware of a few limitations to using `abstract`. First, you can't make constructors abstract. (You'll learn about constructors in the next section covering object creation.) Second, you can't make static methods abstract. This limitation stems from the fact that static methods are declared for all classes, so there is no way to provide a derived implementation for an abstract static method. Finally, you aren't allowed to make private methods abstract. At first, this limitation may seem a little picky, but think about what it means. When you derive a class from a superclass with abstract methods, you must override and implement all the abstract methods or you won't be able to instantiate your new class, and it will remain abstract itself. Now consider that derived classes can't see private members of their superclass, methods included. This results in you not being able to override and implement private abstract methods from the superclass, which means that you can't implement (non-abstract) classes from it. If you were limited to deriving only new abstract classes, you couldn't accomplish much!

## Creating Java Objects

Although most of the design work in object-oriented programming is creating classes, you don't really benefit from that work until you create instances (objects) of those classes. To use a class in a program, you must first create an instance of it.

## The Constructor

Before getting into the details of how to create an object, there is an important method you need to know about: the constructor. When you create an object, you typically want to initialize its member variables. The constructor is a special method you can implement in all your classes; it allows you to initialize variables and perform any other operations when an object is created from the class. The constructor is always given the same name as the class.

Following is the code for two constructors for the hypothetical Alien class:

```
public Alien() {  
    color = Color.green;  
    energy = 100;  
    aggression = 15;  
}  
  
public Alien(Color c, int e, int a) {  
    color = c;  
    energy = e;  
    aggression = a;  
}
```

The Alien class uses method overloading to provide two different constructors. The first constructor takes no parameters and initializes the member variables to default values. The second constructor takes the color, energy, and aggression of the alien and initializes the member variables with them. As well as containing the new constructors, this version of Alien uses access modifiers to explicitly assign access levels to each member variable and method. This is a good habit to get into.

## The new Operator

To create an instance of a class, you declare an object variable and use the new operator. When dealing with objects, a declaration merely states what type of object a variable is to represent. The object isn't actually created until the new operator is used. Following are two examples that use the new operator to create instances of the Alien class:

```
Alien anAlien = new Alien();  
Alien anotherAlien;  
anotherAlien = new Alien(Color.red, 56, 24);
```

In the first example, the variable `anAlien` is declared and the object is created through use of the `new` operator with an assignment directly in the declaration. In the second example, the variable `anotherAlien` is declared first; the object is created and assigned in a separate statement.

## Object Destruction

When an object falls out of scope, it is removed from memory, or deleted. Similar to the constructor that is called when an object is created, Java provides the ability to define a destructor that is called when an object is deleted. Unlike the constructor, which takes on the name of the class, the destructor is called `finalize()`. The `finalize()` method provides a place to perform chores related to the cleanup of the object, and is defined as follows:

```
void finalize() {  
    // cleanup  
}
```

It is worth noting that the `finalize()` method is not guaranteed to be called by Java as soon as an object falls out of scope. The reason for this is that Java deletes objects as part of its system garbage collection, which occurs at inconsistent intervals. Because an object isn't actually deleted until Java performs a garbage collection, the `finalize()` method for the object isn't called until then either. Knowing this, it's safe to say that you shouldn't rely on the `finalize()` method for anything that is time critical. In general, you will rarely need to place code in the `finalize()` method simply because the Java runtime system does a pretty good job of cleaning up after objects on its own.

## Packages

Java provides a powerful means of grouping related classes and interfaces together in a single unit: packages. (You learn about interfaces a little later in this primer.) Put simply, packages are groups of related classes and interfaces. Packages provide a convenient mechanism for managing a large group of classes and interfaces, while avoiding potential naming conflicts. The Java API itself is implemented as a group of packages.

As an example, the `Alien` and `Enemy` classes developed earlier in this primer would fit nicely into an `Enemy` package—along with any other enemy objects. By placing classes into a package, you also allow them to benefit from the default access modifier, which provides classes in the same package with access to each other's class information.



## Declaring Packages

The syntax for the package statement follows:

```
package Identifier;
```

This statement must be placed at the beginning of a compilation unit (a single source file), before any class declarations. Every class located in a compilation unit with a package statement is considered part of that package. You can still spread classes out among separate compilation units; just be sure to include a package statement in each.

Packages can be nested within other packages. When this is done, the Java interpreter expects the directory structure containing the executable classes to match the package hierarchy.

## Importing Packages

When it comes time to use classes outside the package in which you are working, you must use the `import` statement. The `import` statement enables you to import classes from other packages into a compilation unit. You can import individual classes or entire packages of classes at the same time if you want. The syntax for the `import` statement follows:

```
import Identifier;
```

*Identifier* is the name of the class or package of classes you are importing. Going back to the `Alien` class as an example, the `color` member variable is an instance of the `Color` object, which is part of the Java AWT (Abstract Windowing Toolkit) class library. For the compiler to understand this member variable type, you must import the `Color` class. You can do this with either of the following statements:

```
import java.awt.Color;  
import java.awt.*;
```

The Abstract Windowing Toolkit is not part of the MIDP API, and therefore isn't available for use in mobile Java games. Instead, you use a more limited user interface API known as the UI API, which is located in the standard MIDP `javax.microedition.lcdui` package.

**Construction Cue**



The first statement imports the specific class `Color`, which is located in the `java.awt` package. The second statement imports all the classes in the `java.awt` package. Note that the following statement doesn't work:

```
import java.*;
```

This statement doesn't work because you can't import nested packages with the `*` specification. This works only when importing all the classes in a particular package, which is still very useful, especially during early development stages of a project.

### **Construction Cue**



Many Java developers frown on the practice of importing an entire package via the `*` wildcard because it doesn't make clear exactly which classes you are using in a package. I've used the `*` wildcard to import entire packages throughout the examples in this book to keep the code simpler; feel free to import classes one at a time in your own code to be more exacting and organized.

There is one other way to import objects from other packages: explicit package referencing. By explicitly referencing the package name each time you use an object, you can avoid using an `import` statement. Using this technique, the declaration of the `color` member variable in `Alien` would look like this:

```
java.awt.Color color;
```

Explicitly referencing the package name for an external class is generally not required; it usually serves only to clutter up the classname and can make the code harder to read. The exception to this rule is when two packages have classes with the same name. In this case, you are required to explicitly use the package name with the classnames.

## **Class Visibility**

Earlier in this primer, you learned about access modifiers, which affect the visibility of classes and class members. Because class member visibility is determined relative to classes, you're probably wondering what visibility means for a class.

Class visibility is determined relative to packages.

For example, a `public` class is visible to classes in other packages. Actually, `public` is the only explicit access modifier allowed for classes. Without the `public` access modifier, classes default to being visible to other classes in a package but not visible to classes outside the package.

## **Interfaces**

An interface is a prototype for a class and is useful from a logical design perspective. This description of an interface may sound vaguely familiar...Remember abstract classes?

Earlier in this primer, you learned that an abstract class is a class that has been left partially unimplemented because they use abstract methods, which are themselves unimplemented. Interfaces are abstract classes that are left completely unimplemented. “Completely unimplemented” in this case means that no methods in the class have been implemented. Additionally, interface member data is limited to `static final` variables, which means that they are constant.

The benefits of using interfaces are much the same as the benefits of using abstract classes. Interfaces provide a means whereby you can define the protocols for a class without worrying about the implementation details. This seemingly simple benefit can make large projects much easier to manage; after interfaces have been designed, you can develop classes without worrying about communication among classes.

## Declaring Interfaces

The syntax for creating interfaces follows:

```
interface Identifier {  
    InterfaceBody  
}
```

*Identifier* is the name of the interface and *InterfaceBody* refers to the abstract methods and `static final` variables that make up the interface. Because it is given that all the methods in an interface are abstract, it isn't necessary to use the `abstract` keyword.

## Implementing Interfaces

Because an interface is a prototype, or template, for a class, you must implement an interface to arrive at a usable class. Implementing an interface is similar to deriving from a class, except that you are required to implement all methods defined in the interface. To implement an interface, you use the `implements` keyword. The syntax for implementing a class from an interface follows:

```
class Identifier implements Interface {  
    ClassBody  
}
```

*Identifier* refers to the name of the new class, *Interface* is the name of the interface you are implementing, and *ClassBody* is the new class body.

