Chapter 3: Parameter Passing

Parameter Passing

Parameters are what you use to give information to a method. For any method that needs information, you should define parameters in its signature, like we did with the hypothetical setBirthDate() method in the previous chapter. It needed three pieces of information, all numbers: a month, a day, and a year. The method defined these as *int* types. Based on that signature, anyone who wants to use this method to set the birth date for a student must provide those three integer numbers as *arguments*. Arguments are the actual values you put into a parameter list when you use a method.

Parameters (and thus arguments) can be any type defined in Java. That includes all the primitive types, plus any other class or type defined in a program.

Variables, Constants, and Literals

I've used the term *variable* several times without really defining it, so I'll do that now. In Java, a variable is a value that can change. So an int variable could refer to a value of 0, then have its value changed to -5, and then to 1,000,000. (A variable can only hold one value at a time.)

If we use the Student class from earlier to create a variable, that variable would refer to an entire object that would contain all the data fields (instance variables) in the class that hold information about a student. In that case, then, one variable would refer to an object containing other variables.

We declare variables in Java with a type and a name, and sometimes a value and an *access modifier*. Instance variables have an access modifier to tell Java whether a variable is public (anyone can directly access it) or private (only its own class can access it). *Local variables* (defined and used locally in a method) don't have an access modifier because they're always private. But we can assign values to them in their declarations.

For example, here are declarations (again) for some instance variables from our Student class:

```
private String firstName;

private int studentID;

private Date birthDate;
```

In every Student object we create, these declarations will create a String object to hold a name, an int object to hold an ID number, and a Date object to hold a birth date.

You might see local variable declarations like these in a method dealing with coordinates in a graph:

```
int x = 0;

int y = 0;
```

In these declarations, you assign values at the same time you declare the variable instead of assigning values in a separate statement.

Variable names are normally nouns that describe what the variables contain. Java programming convention is that variables start with lowercase characters, and then we capitalize the first letter of each following word in the name, just like method names. Some conventional names are *firstName*, *birthDate*, and *x* (which isn't strictly descriptive, but it does have meaning when dealing with geometrical coordinates).

Constants represent values that don't change. They can be either *named constants*, which are often just called *constants*, or *literals*. A named constant refers to a value that doesn't change. For example, if I were to write a program that dealt with geometry and used the mathematical value π repeatedly, I could either type 3.1415927 every time I needed it, or I could set up a name for it. The name has several advantages. It's easier to remember, easier for another programmer to read and understand, and easier to use correctly. If I use a name, the compiler will insert the same value for that name every time. But if I enter the number manually every time and I accidentally type 3.1445927 once, for example, the compiler won't recognize that it's the wrong value.

Constant names, by convention, are all capital letters with underscores (_) between the words. For example, PI and MAX_SIZE are descriptive constant names.

You define named constants with the *final* keyword. We could define the two constant names above like this:

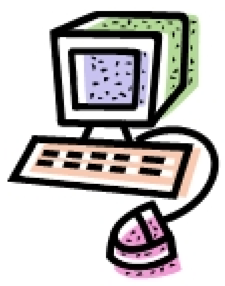
```
final double PI = 3.14159265358979;

final float MAX_SIZE = 1000;
```

Literals are constants, too, since their values can't change. But literals use their values *as* their names. For example, *5* is an integer literal, *2.4* is a floating-point literal, *Q* is a character literal, *false* is a Boolean literal, and *"this is a String literal"* is a String literal.

Console Input and Output

In Java, as in most programming languages, the *console* refers to the default input and output devices. The term goes back to early mainframe days when computers had a large typewriterlike device called the console that operators used to get messages and enter commands. For most of us today, the console is really two devices: our keyboard for input and our monitor for output. All console input and output is in text format.



Java's class library makes console input and output relatively painless. Output is simplest. It requires no declarations at all; you simply use a class (System) and object (out) that are available in every Java program. Java's System class provides several useful objects and methods to us, and the out object is one of them. The out object is an output stream that defaults to your monitor. It's easy to send text output to your monitor like this:

```
System.out.print("Text to display");
```

This statement calls the print() method of the out object in the System class to send text to the screen. It allows arguments that are strings or any of Java's primitive types, and it converts them to text and displays them on the screen.

Input from the console (keyboard) is not quite as simple, as it requires at least one declaration. The simplest input method uses Java's Scanner class, which has a number of methods to extract different types of data from an input text stream. Declaring and using the console input device as a Scanner object is easy:

```
Scanner input = new Scanner(System.in);

String s = input.nextLine();

int i = input.nextInt();

float f =  input.nextFloat();
```

The first line declares and creates a Scanner object using the default input stream. The other lines use several of the Scanner object's methods to extract different types of data from the input. You can find a complete list of its methods at Sun's Java API documentation Web site, which you'll find in this lesson's Supplementary Material.

Expressions

Java uses *expressions* to evaluate combinations of data items that we join with operators. You should already be familiar with arithmetic expressions and logical expressions.

Arithmetic expressions combine numeric values using standard operations like addition, subtraction, multiplication, and division. Those operators are +, -, *, and /, respectively. Remember that multiplication and division have higher priority than addition and subtraction, so they get evaluated first unless you use parentheses to specify a different priority. Here are a few arithmetic expressions as examples. You can assume that we've already declared and assigned values to all variables.

```
1. 2 + 2

2. PI * radius * radius

3. 3 + 4 * 5

4. (3 + 4) * 5

5. (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)
```

These are all valid expressions that give you numeric results. The first calculates a simple sum. The second gives the area of a circle. The third and fourth illustrate the priority of operations. The third does its multiplication first, then its addition, and gets a result of 23. Since the parentheses reorder the operations in the fourth example, the result is 35.

You might (or might not) recognize the last example from algebra as one of the roots of a quadratic equation. Don't worry if you didn't recognize it! It's in a different form than we're used to, and besides, algebra isn't a prerequisite for this course. I just included it to show that Java can handle expressions that are as complex as we need them to be. The *Math.sqrt()* call is to the square root method in the Math class, which gives us a lot of useful mathematical functions. You can find out more about them at the Java API Web site.

Logical Expressions

Logical expressions compare data values to get Boolean values. They also combine Boolean values and always give a Boolean result (either true or false). The comparison operators are *equals* (==), *not equals* (!=), *greater than* (>), *less than* (<), *greater than or equal to* (>=), and less than or equal to (<=). The logical operators are *and* (&&), *or* (||), and *not* (!).

Here are some examples of logical expressions:

```
1. 1 == 1

2. 1 == 2

3. a == b

4. b != c

5. c >= d

6. (d == e) && (e < f)

7. (f == g) || (g == h)

8. !(h == j)
```

Just to make sure we cover all the bases, let me discuss each of these. The first example (1 == 1) will always give a true result. The second (1 == 2) will always be false. The third (a == b) will be true if the variables a and b have the same value and false if they don't. The fourth (b != c) will be true if b and c don't have the same value and false if they do. The fifth (c >= d) will be true if c is at least as big as d and false if c is less than d.

The sixth one, (d == e) && (e < f), requires that two conditions be true if the expression is to be true. That's how the *and* (&&) operator works. For the *and* operator to return true, both of its operands must be true. If either operand is false, && returns false. So if d and e have the same value *and* e's value is less than f's, the expression will be true. Otherwise, it will be false.

The seventh one, (f == g) || (g == h), will be true if *either* of its comparisons is true. The *or* operator (||) needs only one of its operands to be true in order to return a true result. So if f and g have the same value *or* g and h have the same value, the expression will be true. It will be false only if the three of them contain three different values.

The last expression, !(h == j), uses the *not* operator (!) to reverse the result of its operand. So if h and j have the same value, the expression will be false, and it will be true if they have different values.

As always, if any of these raise questions in your mind, please ask me about them in the Discussion Area. That's one of my favorite parts of this course, since it lets me interact with you directly.