PARAMETERS IN JAVA

IF A SUBROUTINE IS A BLACK BOX, then a parameter is something that provides a mechanism for passing information from the outside world into the box. Parameters are part of the interface of a subroutine. They allow you to customize the behavior of a subroutine to adapt it to a particular situation.

As an analogy, consider a thermostat -- a black box whose task it is to keep your house at a certain temperature. The thermostat has a parameter, namely the dial that is used to set the desired temperature. The thermostat always performs the same task: maintaining a constant temperature. However, the exact task that it performs -- that is, **which** temperature it maintains -- is customized by the setting on its dial.

4.3.1 Using Parameters

As an example, let's go back to the "3N+1" problem that was discussed in Subsection 3.2.2. (Recall that a 3N+1 sequence is computed according to the rule, "if N is odd, multiply it by 3 and add 1; if N is even, divide it by 2; continue until N is equal to 1." For example, starting from N=3 we get the sequence: 3, 10, 5, 16, 8, 4, 2, 1.) Suppose that we want to write a subroutine to print out such sequences. The subroutine will always perform the same task: Print out a 3N+1 sequence. But the exact sequence it prints out depends on the starting value of N. So, the starting value of N would be a parameter to the subroutine. The subroutine could be written like this:

```
/**
 * This subroutine prints a 3N+1 sequence to standard output,
using
```

```
 * startingValue as the initial value of N.  It also prints the number
 *  of  terms  in  the  sequence.  The  value  of  the  parameter, startingValue,
 * must  be a positive integer.
 */

static void print3NSequence(int startingValue) {

   int N;      // One of the terms in the sequence.
   int count;  // The number of terms.

   N = startingValue;  // The first term is whatever value
                       //    is passed to the subroutine as
                       //    a parameter.

   count = 1; // We have one term, the starting value, so far.

   System.out.println("The 3N+1 sequence starting from " + N);
   System.out.println();
   System.out.println(N);  // print initial term of sequence

   while (N > 1) {
       if (N % 2 == 1)     // is N odd?
          N = 3 * N + 1;
       else
          N = N / 2;
       count++;   // count this term
       System.out.println(N);  // print this term
   }

   System.out.println();
   System.out.println("There were " + count + " terms in the sequence.");

}  // end print3NSequence
```

The parameter list of this subroutine, "`(int startingValue)`", specifies that the subroutine has one parameter, of type int. Within the body of the subroutine, the parameter name can be used in the same way as a variable name. However, the parameter gets its initial value from **outside** the subroutine. When the subroutine is called, a value must be provided for this parameter in the subroutine call statement. This value will be assigned to the parameter `startingValue` before the body of the subroutine is executed. For example, the subroutine could be called using the subroutine call statement "`print3NSequence(17);`". When the computer executes this statement, the computer first assigns the value 17 to `startingValue` and then executes the statements in the subroutine. This prints the 3N+1 sequence starting from 17. If `K` is a variable of type int, then when the computer executes the subroutine call statement "`print3NSequence(K);`", it will take the value of the variable `K`, assign that value to `startingValue`, and execute the body of the subroutine.

The class that contains `print3NSequence` can contain a `main()` routine (or other subroutines) that call `print3NSequence`. For example, here is a `main()` program that prints out 3N+1 sequences for various starting values specified by the user:

```
public static void main(String[] args) {
   System.out.println("This   program   will   print   out   3N+1
sequences");
   System.out.println("for starting values that you specify.");
   System.out.println();
   int K;  // Input from user; loop ends when K < 0.
   do {
      System.out.println("Enter a starting value.");
      System.out.print("To end the program, enter 0: ");
      K = TextIO.getInt();  // Get starting value from user.
      if (K > 0)   // Print sequence, but only if K is > 0.
         print3NSequence(K);
```

```
        } while (K > 0);   // Continue only if K > 0.
    } // end main
```

Remember that before you can use this program, the definitions of `main` and of `print3NSequence` must both be wrapped inside a class definition.

4.3.2 Formal and Actual Parameters

Note that the term "parameter" is used to refer to two different, but related, concepts. There are parameters that are used in the definitions of subroutines, such as `startingValue` in the above example. And there are parameters that are used in subroutine call statements, such as the `K` in the statement "`print3NSequence(K);`". Parameters in a subroutine definition are called formal parameters or dummy parameters. The parameters that are passed to a subroutine when it is called are called actual parameters or arguments. When a subroutine is called, the actual parameters in the subroutine call statement are evaluated and the values are assigned to the formal parameters in the subroutine's definition. Then the body of the subroutine is executed.

A formal parameter must be a **name**, that is, a simple identifier. A formal parameter is very much like a variable, and -- like a variable -- it has a specified type such as int, boolean, or *String*. An actual parameter is a **value**, and so it can be specified by any expression, provided that the expression computes a value of the correct type. The type of the actual parameter must be one that could legally be assigned to the formal parameter with an assignment statement. For example, if the formal parameter is of type double, then it would be legal to pass an int as the actual parameter since ints can legally be assigned to doubles. When you call a subroutine, you must provide one

actual parameter for each formal parameter in the subroutine's definition. Consider, for example, a subroutine

```
static void doTask(int N, double x, boolean test) {
    // statements to perform the task go here
}
```

This subroutine might be called with the statement

```
doTask(17, Math.sqrt(z+1), z >= 10);
```

When the computer executes this statement, it has essentially the same effect as the block of statements:

```
{
  int N;          // Allocate memory locations for the formal
parameters.
  double x;
  boolean test;
  N = 17;                   // Assign 17 to the first formal
parameter, N.
  x = Math.sqrt(z+1);  // Compute Math.sqrt(z+1), and assign it
to
                       //    the second formal parameter, x.
  test = (z >= 10);     // Evaluate "z >= 10" and assign the
resulting
                       //        true/false value to the third
formal
                       //     parameter, test.
   // statements to perform the task go here
}
```

(There are a few technical differences between this and "`doTask(17,Math.sqrt(z+1),z>=10);`" -- besides the amount of typing -- because of questions about scope of variables and what happens when several variables or parameters have the same name.)

Beginning programming students often find parameters to be surprisingly confusing. Calling a subroutine that already exists is not a problem -- the idea of providing information to the subroutine in a parameter is clear enough. Writing the subroutine definition is another matter. A common beginner's mistake is to assign values to the formal parameters at the beginning of the subroutine, or to ask the user to input their values.**This represents a fundamental misunderstanding.** When the statements in the subroutine are executed, the formal parameters have already been assigned initial values! The values come from the subroutine call statement. Remember that a subroutine is not independent. It is called by some other routine, and it is the calling routine's responsibility to provide appropriate values for the parameters.

4.3.3 Overloading

In order to call a subroutine legally, you need to know its name, you need to know how many formal parameters it has, and you need to know the type of each parameter. This information is called the subroutine's signature. The signature of the subroutine `doTask`, used as an example above, can be expressed as: `doTask(int,double,boolean)`. Note that the signature does **not** include the names of the parameters; in fact, if you just want to **use** the subroutine, you don't even need to know what the formal parameter names are, so the names are not part of the interface.

Java is somewhat unusual in that it allows two different subroutines in the same class to have the same name, provided that their signatures are different. (The language C++ on which Java is based also has this feature.) When this happens, we say that the name of the subroutine is overloaded because it has several different meanings. The computer doesn't get the subroutines mixed up. It can tell which one you want to call

by the number and types of the actual parameters that you provide in the subroutine call statement. You have already seen overloading used with *System.out*. This object includes many different methods named `println`, for example. These methods all have different signatures, such as:

```
println(int)                println(double)
println(String)             println(char)
println(boolean)            println()
```

The computer knows which of these subroutines you want to use based on the type of the actual parameter that you provide. `System.out.println(17)` calls the subroutine with signature `println(int)`, while `System.out.println("Hello")` calls the subroutine with signature `println(String)`. Of course all these different subroutines are semantically related, which is why it is acceptable programming style to use the same name for them all. But as far as the computer is concerned, printing out an int is very different from printing out a *String*, which is different from printing out a boolean, and so forth -- so that each of these operations requires a different method.

Note, by the way, that the signature does **not** include the subroutine's return type. It is illegal to have two subroutines in the same class that have the same signature but that have different return types. For example, it would be a syntax error for a class to contain two methods defined as:

```
int    getln() { ... }
double getln() { ... }
```

So it should be no surprise that in the *TextIO* class, the methods for reading different types are not all named `getln()`. In a given class, there can only be one routine that has the name `getln` and has no parameters. So, the input routines in *TextIO* are

distinguished by having different names, such as `getlnInt()` and `getlnDouble()`.

Java 5.0 introduced another complication: It is possible to have a single subroutine that takes a variable number of actual parameters. You have already used subroutines that do this -- the formatted output routines `System.out.printf` and `TextIO.putf`. When you call these subroutines, the number of parameters in the subroutine call can be arbitrarily large, so it would be impossible to have different subroutines to handle each case. Unfortunately, writing the definition of such a subroutine requires some knowledge of arrays, which will not be covered until Chapter 7. When we get to that chapter, you'll learn how to write subroutines with a variable number of parameters. For now, we will ignore this complication.

Source : http://math.hws.edu/javanotes/c4/s3.html