Section 4.3

Parameters

Subsections

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 can 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. But notice that there is nothing in the subroutine definition that gives a value to the parameter! The parameter gets its initial value from **outside** the subroutine. When the subroutine is called, a value must be provided for the parameter in the subroutine call statement. This value will be assigned to `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 the subroutine can be called by saying "`print3NSequence(K);`". When the computer executes this subroutine call statement, it takes the value of the variable `K`, assigns that value to `startingValue`, and then executes 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, String, or `double[]`. 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 computer automatically assigns values to the parameters before it starts executing the code inside the subroutine. 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 subroutine call statement'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. 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(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('A')` calls the subroutine with signature `println(char)`. 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 char, which is different from printing out a boolean, and so forth -- so that each of these operations requires a different subroutine.

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 subroutines defined as:

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

This is why in the TextIO class, the subroutines for reading different types are not all named `getln()`. In a given class, there can only be one routine that has the name `getln` with no parameters. So, the input routines in TextIO are distinguished by having different names, such as `getlnInt()` and `getlnDouble()`.

4.3.4 Subroutine Examples

Let's do a few examples of writing small subroutines to perform assigned tasks. Of course, this is only one side of programming with subroutines. The task performed by a subroutine is always a subtask in a larger program. The art of designing those programs -- of deciding how to break them up into subtasks -- is the other side of programming with subroutines. We'll return to the question of program design in Section 4.6.

As a first example, let's write a subroutine to compute and print out all the divisors of a given positive integer. The integer will be a parameter to the subroutine. Remember that the syntax of any subroutine is:

```
modifiers  return-type  subroutine-name  ( parameter-list ) {
    statements
}
```

Writing a subroutine always means filling out this format. In this case, the statement of the problem tells us that there is one parameter, of type int, and it tells us what the statements in the body of the subroutine should do. Since we are only working with static subroutines for now, we'll need to use `static` as a modifier. We could add an access modifier (`public` or `private`), but in the absence of any instructions, I'll leave it out. Since we are not told to return a value, the return type is `void`. Since no names are specified, we'll have to make up names for the formal parameter and for the subroutine itself. I'll use `N` for the parameter and `printDivisors` for the subroutine name. The subroutine will look like

```
static void printDivisors( int N ) {
    statements
}
```

and all we have left to do is to write the statements that make up the body of the routine. This is not difficult. Just remember that you have to write the body assuming that N already has a value! The algorithm is: "For each possible divisor D in the range from 1 to N, if D evenly divides N, then print D." Written in Java, this becomes:

```
/**
 * Print all the divisors of N.
 * We assume that N is a positive integer.
 */
static void printDivisors( int N ) {
    int D;   // One of the possible divisors of N.
    System.out.println("The divisors of " + N + " are:");
    for ( D = 1; D <= N; D++ ) {
       if ( N % D == 0 )  // Dose D evenly divide N?
          System.out.println(D);
    }
}
```

I've added a comment before the subroutine definition indicating the contract of the subroutine -- that is, what it does and what assumptions it makes. The contract includes the assumption that N is a positive integer. It is up to the caller of the subroutine to make sure that this assumption is satisfied.

As a second short example, consider the problem: Write a `private` subroutine named `printRow`. It should have a parameter `ch` of type char and a parameter `N` of type int. The subroutine should print out a line of text containing `N` copies of the character `ch`.

Here, we are told the name of the subroutine and the names of the two parameters, and we are told that the subroutine is `private`, so we don't have much choice about the first line of the subroutine definition. The task in this case is pretty simple, so the body of the subroutine is easy to write. The complete subroutine is given by

```
/**
 * Write one line of output containing N copies of the
 * character ch.  If N <= 0, an empty line is output.
 */
private static void printRow( char ch, int N ) {
    int i;  // Loop-control variable for counting off the copies.
    for ( i = 1; i <= N; i++ ) {
        System.out.print( ch );
    }
    System.out.println();
}
```

Note that in this case, the contract makes no assumption about `N`, but it makes it clear what will happen in all cases, including the unexpected case that `N < 0`.

Finally, let's do an example that shows how one subroutine can build on another. Let's write a subroutine that takes a String as a parameter. For each character in the string, it should print a line of output containing 25 copies of that character. It should use the `printRow()` subroutine to produce the output.

Again, we get to choose a name for the subroutine and a name for the parameter. I'll call the subroutine `printRowsFromString` and the parameter `str`. The algorithm is pretty clear: For each position `i` in the string `str`, call `printRow(str.charAt(i),25)` to print one line of the output. So, we get:

```
/**
 * For each character in str, write a line of output
 * containing 25 copies of that character.
 */
private static void printRowsFromString( String str ) {
    int i;  // Loop-control variable for counting off the chars.
    for ( i = 0; i < str.length(); i++ ) {
        printRow( str.charAt(i), 25 );
    }
}
```

We could use `printRowsFromString` in a `main()` routine such as

```
public static void main(String[] args) {
    String inputLine;  // Line of text input by user.
    System.out.print("Enter a line of text: ");
    inputLine = TextIO.getln();
    System.out.println();
    printRowsFromString( inputLine );
}
```

Of course, the three routines, `main()`, `printRowsFromString()`, and `printRow()`, would have to be collected together inside the same class. The program is rather useless, but it does demonstrate the use of subroutines. You'll find the program in the file *RowsOfChars.java*, if you want to take a look.

4.3.5 Array Parameters

It's possible for the type of a parameter to be an array type. This means that an entire array of values can be passed to the subroutine as a single parameter. For example, we might want a subroutine to print all the values in an integer array in a neat format, separated by commas and enclosed in a pair of square brackets. To tell it which array to print, the subroutine would have a parameter of type `int[]`:

```
static void printValuesInList( int[] list ) {
    System.out.print('[');
    int i;
    for ( i = 0; i < list.length; i++ ) {
        if ( i > 0 )
            System.out.print(','); // No comma in front of list[0]
        System.out.print(list[i]);
    }
    System.out.println((']');
}
```

To use this subroutine, you need an actual array. Here is a legal, though not very realistic, code segment that creates an array just to pass it as an argument to the subroutine:

```
int[] numbers;
numbers = new int[3];
numbers[0] = 42;
numbers[1] = 17;
numbers[2] = 256;
printValuesInList( numbers );
```

The output produced by the last statement would be `[42,17,256]`.

4.3.6 Command-line Arguments

The `main` routine of a program has a parameter of type `String[]`. When the main routine is called, some actual array of String must be passed to `main()` as the value of the parameter. The system provides the actual parameter when it calls `main()`, so the values come from outside the program. Where do the strings in the array come from, and what do they mean? The strings in the array are command-line arguments from the command that was used to run the program. When using a command-line interface, the user types a command to tell the system to execute a program. The user can include extra input in this command, beyond the name of the program. This extra input becomes the command-line arguments. The system takes the command-line arguments, puts them into an array of strings, and passes that array to `main()`.

For example, if the name of the program is `myProg`, then the user can type "`java myProg`" to execute the program. In this case, there are no command-line arguments. But if the user types the command

```
java myProg one two three
```

then the command-line arguments are the strings "one", "two", and "three". The system puts these strings into an array of Strings and passes that array as a parameter to the `main()` routine. Here, for example, is a short program that simply prints out any command line arguments entered by the user:

```
public class CLDemo {

   public static void main(String[] args) {
      System.out.println("You entered " + args.length
                                  + " command-line arguments");
      if (args.length > 0) {
         System.out.println("They were:");
         int i;
         for ( i = 0; i < args.length; i++ )
            System.out.println("   " + args[i]);
      }
   } // end main()

} // end class CLDemo
```

Note that the parameter, `args`, can be an array of length zero. This just means that the user did not include any command-line arguments when running the program.

In practice, command-line arguments are often used to pass the names of files to a program. For example, consider the following program for making a copy of a text file. It does this by copying one line at a time from the original file to the copy, using TextIO. The function `TextIO.eof()` is a boolean-valued function that is `true` if the end of the file has been reached.

```
/**
 *  Requires two command line arguments, which must be file names.  The
 *  the first must be the name of an existing file.  The second is the name
 *  of a file to be created by the program.  The contents of the first file
 *  are copied into the second.  WARNING:  If the second file already
 *  exists when the program is run, its previous contents will be lost!
 *  This program only works for plain text files.
 */
public class CopyTextFile {

    public static void main( String[] args ) {
        if (args.length < 2 ) {
            System.out.println("Two command-line arguments are required!");
            System.exit(1);
        }
        TextIO.readFile( args[0] );   // Open the original file for reading.
        TextIO.writeFile( args[1] );  // Open the copy file for writing.
        int lineCount;  // Number of lines copied
```

```
            lineCount = 0;
            while ( TextIO.eof() == false ) {
                // Read one line from the original file and write it to the copy.
                String line;
                line = TextIO.getln();
                TextIO.putln(line);
                lineCount++;
            }
            System.out.printf( "%d lines copied from %s to %s%n",
                                    lineCount, args[0], args[1] );
        }

    }
```

Since most programs are run in a GUI environment these days, command-line arguments aren't as important as they used to be. But at least they provide a nice example of how array parameters can be used.

4.3.7 Throwing Exceptions

I have been talking about the "contract" of a subroutine. The contract says what the subroutine will do, provided that the caller of the subroutine provides acceptable values for the subroutine's parameters. The question arises, though, what should the subroutine do when the caller violates the contract by providing bad parameter values?

We've already seen that some subroutines respond to bad parameter values by throwing exceptions. (See Section 3.7.) For example, the contract of the built-in subroutine `Double.parseDouble` says that the parameter should be a string representation of a number of type double; if this is true, then the subroutine will convert the string into the equivalent numeric value. If the caller violates the contract by passing an invalid string as the actual parameter, the subroutine responds by throwing an exception of type NumberFormatException.

Many subroutines throw IllegalArgumentExceptions in response to bad parameter values. You might want to do the same in your own subroutines. This can be done with a throw statement. An exception is an object, and in order to throw an exception, you must create an exception object. You won't officially learn how to do this until Chapter 5, but for now, you can use the following syntax for a `throw` statement that throws an IllegalArgumentException:

```
throw  new  IllegalArgumentException( error-message );
```

where **error-message** is a string that describes the error that has been detected. (The word "new" in this statement is what creates the object.) To use this statement in a subroutine, you would check whether the values of the parameters are legal. If not, you would throw the exception. For example, consider the `print3NSequence` subroutine from the beginning of this section. The parameter of `print3NSequence` is supposed to be a positive integer. We can modify the subroutine definition to make it throw an exception when this condition is violated:

```
static void print3NSequence(int startingValue) {

   if (startingValue <= 0)  // The contract is violated!
      throw new IllegalArgumentException( "Starting value must be positive." );
   .
   .  // (The rest of the subroutine is the same as before.)
   .
```

If the start value is bad, the computer executes the `throw` statement. This will immediately terminate the subroutine, without executing the rest of the body of the subroutine. Furthermore, the program as a

whole will crash unless the exception is "caught" and handled elsewhere in the program by a `try..catch` statement, as discussed in Section 3.7. For this to work, the subroutine call would have to be in the "try" part of the statement.

4.3.8 Global and Local Variables

I'll finish this section on parameters by noting that we now have three different sorts of variables that can be used inside a subroutine: local variables declared in the subroutine, formal parameter names, and static member variables that are declared outside the subroutine.

Local variables have no connection to the outside world; they are purely part of the internal working of the subroutine.

Parameters are used to "drop" values into the subroutine when it is called, but once the subroutine starts executing, parameters act much like local variables. Changes made inside a subroutine to a formal parameter have no effect on the rest of the program (at least if the type of the parameter is one of the primitive types -- things are more complicated in the case of arrays and objects, as we'll see later).

Things are different when a subroutine uses a variable that is defined outside the subroutine. That variable exists independently of the subroutine, and it is accessible to other parts of the program as well. Such a variable is said to be global to the subroutine, as opposed to the local variables defined inside the subroutine. A global variable can be used in the entire class in which it is defined and, if it not `private`, in other classes as well. Changes made to a global variable can have effects that extend outside the subroutine where the changes are made. You've seen how this works in the last example in the previous section, where the values of the global variables, `gamesPlayed` and `gamesWon`, are computed inside a subroutine and are used in the `main()` routine.

It's not always bad to use global variables in subroutines, but you should realize that the global variable then has to be considered part of the subroutine's interface. The subroutine uses the global variable to communicate with the rest of the program. This is a kind of sneaky, back-door communication that is less visible than communication done through parameters, and it risks violating the rule that the interface of a black box should be straightforward and easy to understand. So before you use a global variable in a subroutine, you should consider whether it's really necessary.

I don't advise you to take an absolute stand against using global variables inside subroutines. There is at least one good reason to do it: If you think of the class as a whole as being a kind of black box, it can be very reasonable to let the subroutines inside that box be a little sneaky about communicating with each other, if that will make the class as a whole look simpler from the outside.

[Previous Section | Next Section | Chapter Index | Main Index]