RETURN VALUES IN JAVA

A SUBROUTINE THAT RETURNS A VALUE is called a function. A given function can only return a value of a specified type, called the return type of the function. A function call generally occurs in a position where the computer is expecting to find a value, such as the right side of an assignment statement, as an actual parameter in a subroutine call, or in the middle of some larger expression. A boolean-valued function can even be used as the test condition in an `if`, `while`, `for` or `do..while` statement.

(It is also legal to use a function call as a stand-alone statement, just as if it were a regular subroutine. In this case, the computer ignores the value computed by the subroutine. Sometimes this makes sense. For example, the function `TextIO.getln()`, with a return type of *String*, reads and returns a line of input typed in by the user. Usually, the line that is returned is assigned to a variable to be used later in the program, as in the statement "`name = TextIO.getln();`". However, this function is also useful as a subroutine call statement "`TextIO.getln();`", which still reads all input up to and including the next carriage return. Since the return value is not assigned to a variable or used in an expression, it is simply discarded. So, the effect of the subroutine call is to read **and discard** some input. Sometimes, discarding unwanted input is exactly what you need to do.)

4.4.1 The return statement

You've already seen how functions such as `Math.sqrt()` and `TextIO.getInt()` can be used. What you haven't seen is how to write functions of your own. A function takes the same form as a regular subroutine, except that you have to specify the value that is to be returned by the subroutine. This is done with a return statement, which has the following syntax:

```
return  expression ;
```

Such a `return` statement can only occur inside the definition of a function, and the type of the **expression** must match the return type that was specified for the function. (More exactly, it must be legal to assign the expression to a variable whose type is specified by the return type.) When the computer executes this `return` statement, it evaluates the expression, terminates execution of the function, and uses the value of the expression as the returned value of the function.

For example, consider the function definition

```
static double pythagoras(double x, double y) {
      // Computes the length of the hypotenuse of a right
      // triangle, where the sides of the triangle are x and y.
    return  Math.sqrt( x*x + y*y );
}
```

Suppose the computer executes the statement "`totalLength = 17 + pythagoras(12,5);`". When it gets to the term `pythagoras(12,5)`, it assigns the actual parameters `12` and `5` to the formal parameters `x` and `y` in the function. In the body of the function, it evaluates `Math.sqrt(12.0*12.0 + 5.0*5.0)`, which works out to `13.0`. This value is "returned" by the function, so the `13.0` essentially replaces the function call in the assignment statement, which then has the same effect as the statement "`totalLength = 17+13.0`". The return value is added to `17`, and the result, 30.0, is stored in the variable, `totalLength`.

Note that a `return` statement does not have to be the last statement in the function definition. At any point in the function where you know the value that you want to return, you can return it. Returning a value will end the function immediately, skipping any subsequent statements in the function. However, it must be the case that the function definitely does return some value, no matter what path the execution of the function takes through the code.

You can use a `return` statement inside an ordinary subroutine, one with declared return type "`void`". Since a void subroutine does not return a value, the `return` statement does not include an expression; it simply takes the form "`return;`". The effect of this statement is to terminate execution of the subroutine and return control back to the point in the program from which the subroutine was called. This can be convenient if you want to terminate execution somewhere in the middle of the subroutine, but `return` statements are optional in non-function subroutines. In a function, on the other hand, a return statement, with expression, is always required.

4.4.2 Function Examples

Here is a very simple function that could be used in a program to compute 3N+1 sequences. (The 3N+1 sequence problem is one we've looked at several times already, including in the previous section.) Given one term in a 3N+1 sequence, this function computes the next term of the sequence:

```
static int nextN(int currentN) {
   if (currentN % 2 == 1)     // test if current N is odd
      return 3*currentN + 1;  // if so, return this value
   else
      return currentN / 2;    // if not, return this instead
```

```
}
```

This function has two `return` statements. Exactly one of the two `return` statements is executed to give the value of the function. Some people prefer to use a single `return` statement at the very end of the function when possible. This allows the reader to find the `return` statement easily. You might choose to write `nextN()` like this, for example:

```
static int nextN(int currentN) {
   int answer;  // answer will be the value returned
   if (currentN % 2 == 1)    // test if current N is odd
      answer = 3*currentN+1; // if so, this is the answer
   else
      answer = currentN / 2; // if not, this is the answer
   return answer;   // (Don't forget to return the answer!)
}
```

Here is a subroutine that uses this `nextN` function. In this case, the improvement from the version of this subroutine in Section 4.3 is not great, but if `nextN()` were a long function that performed a complex computation, then it would make a lot of sense to hide that complexity inside a function:

```
static void print3NSequence(int startingValue) {

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

   N  =  startingValue;       //  Start  the  sequence  with
startingValue.
   count = 1;

   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) {
       N = nextN( N );     // Compute next term, using the
function nextN.
       count++;          // Count this term.
       System.out.println(N);  // Print this term.
   }

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

}
```

Source : http://math.hws.edu/javanotes/c4/s4.html