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

Section 4.4

Return Values

Subsections

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.

Note that a return inside a loop will end the loop as well as the subroutine that contains it. Similarly, a return in a switch statement breaks out of the switch statement as well as the subroutine. So, you will sometimes use return in contexts where you are used to seeing a break.

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 the 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.");

   }
```

Here are a few more examples of functions. The first one computes a letter grade corresponding to a given numerical grade, on a typical grading scale:

```
/**
 * Returns the letter grade corresponding to the numerical
 * grade that is passed to this function as a parameter.
 */
static char letterGrade(int numGrade) {
   if (numGrade >= 90)
      return 'A';   // 90 or above gets an A
   else if (numGrade >= 80)
      return 'B';   // 80 to 89 gets a B
   else if (numGrade >= 65)
      return 'C';   // 65 to 79 gets a C
   else if (numGrade >= 50)
      return 'D';   // 50 to 64 gets a D
   else
      return 'F';   // anything else gets an F

}  // end of function letterGrade
```

The type of the return value of `letterGrade()` is char. Functions can return values of any type at all. Here's a function whose return value is of type boolean. It demonstrates some interesting programming points, so you should read the comments:

```
/**
 * This function returns true if N is a prime number.  A prime number
 * is an integer greater than 1 that is not divisible by any positive
 * integer, except itself and 1.  If N has any divisor, D, in the range
 * 1 < D < N, then it has a divisor in the range 2 to Math.sqrt(N), namely
 * either D itself or N/D.  So we only test possible divisors from 2 to
 * Math.sqrt(N).
 */
static boolean isPrime(int N) {

   int divisor;  // A number we will test to see whether it evenly divides N.

   if (N <= 1)
      return false;  // No number <= 1 is a prime.

   int maxToTry;  // The largest divisor that we need to test.

   maxToTry = (int)Math.sqrt(N);
        // We will try to divide N by numbers between 2 and maxToTry.
```

```
            // If N is not evenly divisible by any of these numbers, then
            // N is prime.  (Note that since Math.sqrt(N) is defined to
            // return a value of type double, the value must be typecast
            // to type int before it can be assigned to maxToTry.)

        for (divisor = 2; divisor <= maxToTry; divisor++) {
            if ( N % divisor == 0 )  // Test if divisor evenly divides N.
               return false;         // If so, we know N is not prime.
                                     // No need to continue testing!
        }

        // If we get to this point, N must be prime.  Otherwise,
        // the function would already have been terminated by
        // a return statement in the previous loop.

        return true;  // Yes, N is prime.

    }  // end of function isPrime
```

Finally, here is a function with return type String. This function has a String as parameter. The returned value is a reversed copy of the parameter. For example, the reverse of "Hello World" is "dlroW olleH". The algorithm for computing the reverse of a string, `str`, is to start with an empty string and then to append each character from `str`, starting from the last character of `str` and working backwards to the first:

```
static String reverse(String str) {
   String copy;  // The reversed copy.
   int i;        // One of the positions in str,
                 //       from str.length() - 1 down to 0.
   copy = "";    // Start with an empty string.
   for ( i = str.length() - 1;  i >= 0;  i-- ) {
            // Append i-th char of str to copy.
      copy = copy + str.charAt(i);
   }
   return copy;
}
```

A palindrome is a string that reads the same backwards and forwards, such as "radar". The `reverse()` function could be used to check whether a string, `word`, is a palindrome by testing `"if (word.equals(reverse(word)))"`.

By the way, a typical beginner's error in writing functions is to print out the answer, instead of returning it. **This represents a fundamental misunderstanding.** The task of a function is to compute a value and return it to the point in the program where the function was called. That's where the value is used. Maybe it will be printed out. Maybe it will be assigned to a variable. Maybe it will be used in an expression. But it's not for the function to decide.

4.4.3 3N+1 Revisited

I'll finish this section with a complete new version of the 3N+1 program. This will give me a chance to show the function `nextN()`, which was defined above, used in a complete program. I'll also take the opportunity to improve the program by getting it to print the terms of the sequence in columns, with five terms on each line. This will make the output more presentable. The idea is this: Keep track of how many terms have been printed on the current line; when that number gets up to 5, start a new line of output. To make the terms line up into neat columns, I use formatted output.

```
/**
 * A program that computes and displays several 3N+1 sequences.  Starting
```

```
 * values for the sequences are input by the user.  Terms in the sequence
 * are printed in columns, with five terms on each line of output.
 * After a sequence has been displayed, the number of terms in that
 * sequence is reported to the user.
 */
public class ThreeN2 {


   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;   // Starting point for sequence, specified by the user.
      do {
         System.out.println("Enter a starting value;");
         System.out.print("To end the program, enter 0: ");
         K = TextIO.getlnInt();  // 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


   /**
    * print3NSequence 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 found.
      int onLine;  // The number of terms that have been output
                   //     so far on the current line.

      N = startingValue;   // Start the sequence with startingValue;
      count = 1;           // We have one term so far.

      System.out.println("The 3N+1 sequence starting from " + N);
      System.out.println();
      System.out.printf("%8d", N);  // Print initial term, using 8 characters.
      onLine = 1;        // There's now 1 term on current output line.

      while (N > 1) {
          N = nextN(N);  // compute next term
          count++;   // count this term
          if (onLine == 5) {  // If current output line is full
             System.out.println();  // ...then output a carriage return
             onLine = 0;      // ...and note that there are no terms
                              //               on the new line.
          }
          System.out.printf("%8d", N);  // Print this term in an 8-char column.
          onLine++;   // Add 1 to the number of terms on this line.
      }

      System.out.println();  // end current line of output
      System.out.println();  // and then add a blank line
      System.out.println("There were " + count + " terms in the sequence.");

   }  // end of print3NSequence
```

```
    /**
     * nextN computes and returns the next term in a 3N+1 sequence,
     * given that the current term is currentN.
     */
    static int nextN(int currentN) {
        if (currentN % 2 == 1)
           return 3 * currentN + 1;
        else
           return currentN / 2;
    }  // end of nextN()


} // end of class ThreeN2
```

You should read this program carefully and try to understand how it works.

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