FOR LOOPS

The `for` statement makes a common type of while loop easier to write. Many while loops have the general form:

```
initialization
while ( continuation-condition ) {
    statements
    update
}
```

For example, consider this example, copied from an example in Section 3.2:

```
years = 0;  // initialize the variable years
while ( years < 5 ) {   // condition for continuing loop

    interest = principal * rate;    //
    principal += interest;          // do three statements
    System.out.println(principal);  //

    years++;   // update the value of the variable, years
}
```

This loop can be written as the following equivalent `for` statement:

```
for ( years = 0;  years < 5;  years++ ) {
   interest = principal * rate;
   principal += interest;
   System.out.println(principal);
}
```

The initialization, continuation condition, and updating have all been combined in the first line of the `for` loop. This keeps everything involved in the "control" of the loop in one place, which helps make the loop easier to read and understand. The `for` loop is executed in exactly the same way as the original code: The initialization part is

executed once, before the loop begins. The continuation condition is executed before each execution of the loop, and the loop ends when this condition is `false`. The update part is executed at the end of each execution of the loop, just before jumping back to check the condition.

The formal syntax of the `for` statement is as follows:

```
for ( initialization; continuation-condition; update )
     statement
```

or, using a block statement:

```
for ( initialization; continuation-condition; update ) {
     statements
}
```

The **continuation-condition** must be a boolean-valued expression. The **initialization** is usually a declaration or an assignment statement, but it can be any expression that would be allowed as a statement in a program. The **update** can be any expression statement, but is usually an increment, a decrement, or an assignment statement. Any of the three can be empty. If the continuation condition is empty, it is treated as if it were "`true`," so the loop will be repeated forever or until it ends for some other reason, such as a `break` statement. (Some people like to begin an infinite loop with "`for (;;)`" instead of "`while (true)`".)

Usually, the initialization part of a `for` statement assigns a value to some variable, and the update changes the value of that variable with an assignment statement or with an increment or decrement operation. The value of the variable is tested in the continuation condition, and the loop ends when this condition evaluates to `false`. A variable used in this way is called a loop control variable. In the `for` statement given above, the loop control variable is `years`.

Certainly, the most common type of `for` loop is the counting loop, where a loop control variable takes on all integer values between some minimum and some maximum value. A counting loop has the form

```
for ( variable = min;  variable <= max; variable++ ) {
     statements
}
```

where **min** and **max** are integer-valued expressions (usually constants). The **variable** takes on the values **min**, **min**+1, **min**+2, ..., **max**. The value of the loop control variable is often used in the body of the loop. The `for` loop at the beginning of this section is a counting loop in which the loop control variable, `years`, takes on the values 1, 2, 3, 4, 5. Here is an even simpler example, in which the numbers 1, 2, ..., 10 are displayed on standard output:

```
for ( N = 1 ;  N <= 10 ;  N++ )
   System.out.println( N );
```

For various reasons, Java programmers like to start counting at 0 instead of 1, and they tend to use a "<" in the condition, rather than a "<=". The following variation of the above loop prints out the ten numbers 0, 1, 2, ..., 9:

```
for ( N = 0 ;  N < 10 ;  N++ )
   System.out.println( N );
```

Using < instead of <= in the test, or vice versa, is a common source of off-by-one errors in programs. You should always stop and think, Do I want the final value to be processed or not?

It's easy to count down from 10 to 1 instead of counting up. Just start with 10, decrement the loop control variable instead of incrementing it, and continue as long as the variable is greater than or equal to one.

```
for ( N = 10 ;  N >= 1 ;  N-- )
   System.out.println( N );
```

Now, in fact, the official syntax of a `for` statement actually allows both the initialization part and the update part to consist of several expressions, separated by commas. So we can even count up from 1 to 10 and count down from 10 to 1 at the same time!

```
for ( i=1, j=10;  i <= 10;  i++, j-- ) {
   System.out.printf("%5d", i); // Output i in a 5-character
wide column.
   System.out.printf("%5d", j); // Output j in a 5-character
column
   System.out.println();       //     and end the line.
}
```

As a final introductory example, let's say that we want to use a `for` loop that prints out just the even numbers between 2 and 20, that is: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20. There are several ways to do this. Just to show how even a very simple problem can be solved in many ways, here are four different solutions (three of which would get full credit):

```
(1)   // There are 10 numbers to print.
      // Use a for loop to count 1, 2,
      // ..., 10.  The numbers we want
      // to print are 2*1, 2*2, ... 2*10.

      for (N = 1; N <= 10; N++) {
         System.out.println( 2*N );
      }


(2)   // Use a for loop that counts
      // 2, 4, ..., 20 directly by
      // adding 2 to N each time through
```

```
      // the loop.

      for (N = 2; N <= 20; N = N + 2) {
         System.out.println( N );
      }
```

(3)
```
      // Count off all the numbers
      // 2, 3, 4, ..., 19, 20, but
      // only print out the numbers
      // that are even.

      for (N = 2; N <= 20; N++) {
         if ( N % 2 == 0 ) // is N even?
            System.out.println( N );
      }
```

(4)
```
      // Irritate the professor with
      // a solution that follows the
      // letter of this silly assignment
      // while making fun of it.

      for (N = 1; N <= 1; N++) {
         System.out.println("2 4 6 8 10 12 14 16 18 20");
      }
```

Perhaps it is worth stressing one more time that a `for` statement, like any statement, never occurs on its own in a real program. A statement must be inside the `main` routine of a program or inside some other subroutine. And that subroutine must be defined inside a class. I should also remind you that every variable must be declared before it can be used, and that includes the loop control variable in a `for` statement. In all the examples that you have seen so far in this section, the loop control variables should be declared to be of type int. It is not required that a loop control variable be an integer. Here, for example, is a `for`loop in which the

variable, ch, is of type char, using the fact that the ++ operator can be applied to characters as well as to numbers:

```
// Print out the alphabet on one line of output.
char ch;  // The loop control variable;
          //       one of the letters to be printed.
for ( ch = 'A';  ch <= 'Z';  ch++ )
    System.out.print(ch);
System.out.println();
```

Example: Counting Divisors

Let's look at a less trivial problem that can be solved with a for loop. If N and D are positive integers, we say that D is a divisor of N if the remainder when D is divided into N is zero. (Equivalently, we could say that N is an even multiple of D.) In terms of Java programming, D is a divisor of N if N % D is zero.

Let's write a program that inputs a positive integer, N, from the user and computes how many different divisors N has. The numbers that could possibly be divisors of N are 1, 2, ..., N. To compute the number of divisors of N, we can just test each possible divisor of N and count the ones that actually do divide N evenly. In pseudocode, the algorithm takes the form

```
Get a positive integer, N, from the user
Let divisorCount = 0
for each number, testDivisor, in the range from 1 to N:
    if testDivisor is a divisor of N:
        Count it by adding 1 to divisorCount
Output the count
```

This algorithm displays a common programming pattern that is used when some, but not all, of a sequence of items are to be processed. The general pattern is

```
for each item in the sequence:
   if the item passes the test:
       process it
```

The `for` loop in our divisor-counting algorithm can be translated into Java code as

```
for (testDivisor = 1; testDivisor <= N; testDivisor++) {
   if ( N % testDivisor == 0 )
      divisorCount++;
}
```

On a modern computer, this loop can be executed very quickly. It is not impossible to run it even for the largest legal int value, 2147483647. (If you wanted to run it for even larger values, you could use variables of type long rather than int.) However, it does take a significant amount of time for very large numbers. So when I implemented this algorithm, I decided to output a dot every time the computer has tested one million possible divisors. In the improved version of the program, there are two types of counting going on. We have to count the number of divisors and we also have to count the number of possible divisors that have been tested. So the program needs two counters. When the second counter reaches 1000000, the program outputs a '.' and resets the counter to zero so that we can start counting the next group of one million. Reverting to pseudocode, the algorithm now looks like

```
Get a positive integer, N, from the user
Let divisorCount = 0  // Number of divisors found.
Let numberTested = 0  // Number of possible divisors tested
                      //            since  the  last  period  was
output.
for each number, testDivisor, in the range from 1 to N:
    if testDivisor is a divisor of N:
        Count it by adding 1 to divisorCount
    Add 1 to numberTested
    if numberTested is 1000000:
        print out a '.'
```

```
        Reset numberTested to 0
Output the count
```

Finally, we can translate the algorithm into a complete Java program. Here it is, followed by an applet that simulates it:

```
/**
 * This program reads a positive integer from the user.
 * It counts how many divisors that number has, and
 * then it prints the result.
 */

public class CountDivisors {

   public static void main(String[] args) {

      int N;  // A positive integer entered by the user.
              // Divisors of this number will be counted.

      int testDivisor;  // A number between 1 and N that is a
                        // possible divisor of N.

      int divisorCount;   // Number of divisors of N that have
been found.

      int numberTested;   // Used to count how many possible
divisors
                          // of N have been tested.   When the
number
                          // reaches 1000000, a period is output
and
                          // the value of numberTested is reset
to zero.

      /* Get a positive integer from the user. */

      while (true) {
         System.out.print("Enter a positive integer: ");
```

```
            N = TextIO.getlnInt();
            if (N > 0)
               break;
            System.out.println("That number is not positive.
Please try again.");
         }

         /* Count the divisors, printing a "." after every 1000000
tests. */

         divisorCount = 0;
         numberTested = 0;

         for (testDivisor = 1; testDivisor <= N; testDivisor++) {
            if ( N % testDivisor == 0 )
               divisorCount++;
            numberTested++;
            if (numberTested == 1000000) {
               System.out.print('.');
               numberTested = 0;
            }
         }

         /* Display the result. */

         System.out.println();
         System.out.println("The number of divisors of " + N
                              + " is " + divisorCount);

      } // end main()

   } // end class CountDivisors
```

Source : http://math.hws.edu/javanotes/c3/s4.html