BCA PART- III

PAPER – XIX

TOPIC: LOOPING IN JAVA

PREPARED BY: DR. KIRAN PANDEY

(SCHOOL OF COMPUTER SCIENCE)

EMAIL ID: kiranpandey.nou@gmail.com

Introduction

The process of repeatedly executing a block of statements is known as *looping*. The statements in the block may be executed any number of times, from zero to *infinite* number. If a loop continues forever, it is called *infinite loop*.

The flowcharts in Fig. 5.1 illustrate these structures. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the condition are not satisfied, then the body of the loop will not be executed. In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

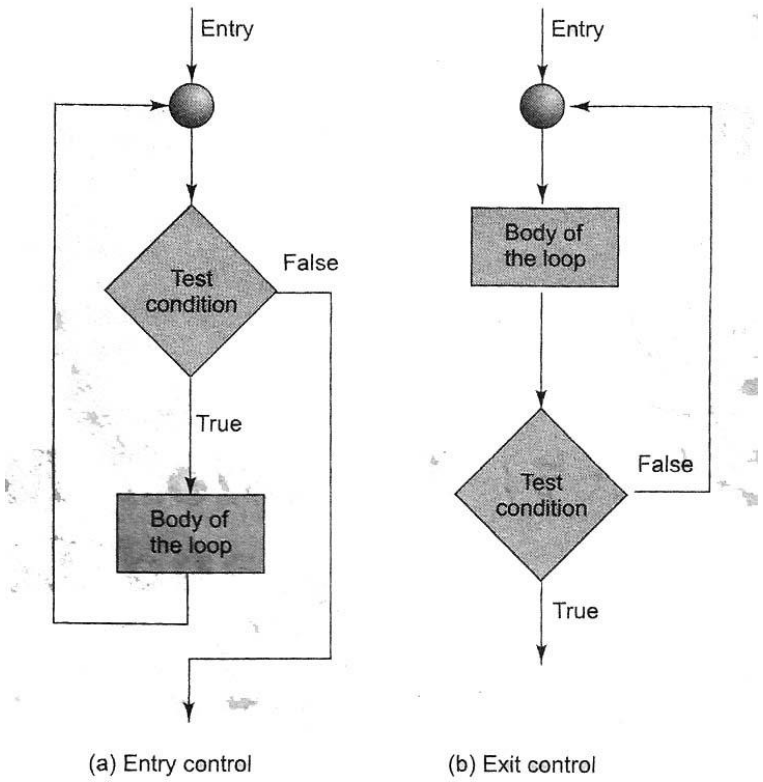


Fig. 5.1 Loop Control Structure

The test conditions should be carefully stated in order to perform the desired number of loop executions. It is assumed that the test condition will eventually transfer the control out of the loop. In case, due to some reason it does not do so, the control stets up an *infinite* loop and the body is executed over and over again.

A looping process, in general, would include the following four steps:

1. Setting and initialization of a counter.

2. Execution of the statements in the loop.

3. Test for a specified condition for execution of the loop.

4. Incrementing the counter.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met with.

The Java language provides for three constructs for performing loop operations. They are:

1. **while** construct

2. **do** construct

3. **for** construct

We shall discuss the features and applications of each of these constructs in this unit.

The While Statement

The simplest of all the looping structures in Java is the **while** statement. The basic format of the **while** statement is

```
Initialization;
While (test condition)
{
      Body for the loop
}
```

The **while** is an *entry-controlled* loop statement. The *test condition* is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only once statement.

Consider the following code segment;

```
.............
.............
sum = 0
n = 1;
while(n < = 10)
{
    sum = sum + n * n;
    n = n+1;
}
```

```
System.out.println("Sum = "+ sum);
.............
.............
```

The body of the loop is executed 10 times for n = 1, 2 ……., 10 each time adding the square of the value of n, which is incremented inside the loop. The test condition may also be written as n < 11; the result would be the same. Program 5.1 illustrates the use of the **while** for reading a string of characters from the keyboard. The loop terminates when c = '\ n', the newline character.

Program 5.1 Using while loop

```
class WhileTest
{
   public static void main(String args[ ])
   {
      StringBuffer string = new StringBuffer( );
      Char c;
      System.out.println("Enter & String ");
      try
      {
         While ( (c = (char)System.in.read()) != '\n')
         {
            string.append(c); // Append character
         }
      }
      catch (Exception e)
      {
         System.out.println("Error in input");
      }
      System.out.println(" You have entered ... ");
      system.out.println(string);
   }
}
```

Given below is the output of Program 5.1

```
Enter a string
Java is a true Object-Oriented Language
    You have entered
Java is a true Object-Oriented Language
```

The do Statement

The **while** loop construct that we have discussed in the previous section makes a test condition *before* the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement. This takers the form:

```
Initialization;
do
{
      Body of the loop
}
```

```
While (test condition);
```

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the *test condition* in the **while** statement is evaluated. If the condition is true, the program continues to evaluate the body hf the *loop* once again. This process continues as long as the *condition* is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

Since the *test condition* is evaluated at the bottom of the loop, the **do….while** construct provides an *exit-controlled* loop and therefore the body of the lop is always executed at least once.

Consider an example

```
.............
.............
i = 1;
sum = 0;
do
{
   sum = sum + 1;
   i = 1+2;
}
While(sum = 40 || i < 10);
.............
.............
```

The loop will be executed as long as one of the two relations is true. Program 5.2 the use of **do….while** loops for printing a multiplication table.

Program 5.2 Printing multiplication table using do……while loop

```
class
{
   public static void main(String args[])
   {
      int row, column, y;
      System.out.println("Multiplication Table \n");
      row = 1;
      do
      {
         column = 1;
         do
         {
            y = row * column;
            System.out.print(" " + y);
            Column = column + 1;
         }
         While (column <= 3);
         System.out.println("\n");
```

```
            row = row + 1;
        }
        While (tow <= 3);
    }
}
```

Program 5.2 uses two **do-while** loops in nested form and produces the following output:

```
Multiplication    Table
   1       2       3
   2       4       6
   3       6       9
```

The for Statement

The **for** loop is another *entry-controlled* loop that provides a more concise loop control structure. The general form of the **for** loop is

```
for (.initialization ; test condition ; increment)
{
      Body of the loop
}
```

The execution of the **for** statement is as follows:

1. *Initialization* of the *control variables* is done first, using assignment statements such as i = 1 and count = 0. The variables **i** and **count** are known as loop-control variables.
2. The value of the control variable is tested using the *test condition*. The test condition is a relational expression, such as i < 10 that determines when the loop will exit. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the **for** statement after evaluating the last statement in the loop. Now, the control variable is *incremented* using an assignment statement such as i = i + 1 and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test condition.

Consider the following segment of a program

```
for   (x = 0 ; x < = 9; x = x+1)
{
      System.out.println(x);
}
```

This **for** loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons. Note that thre is no semicolon at the end of the *increment* section, **x = x + 1**.

The **for** statement allows for negative *increments*. For example, the loop discussed above can be written as follows:

```
for   (x = 9; x > 0; x = x-1)
      System.out.println(x);
```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Since the conditional test is always performed at the beginning of the loop, the body of the loop may not be executed at all, if the condition fails at the start. For example,

```
for (x = 9; x < 9; x = x-1)
{
.............
.............
}
```

will never the executed because the test condition fails at the very beginning itself.

Let us consider the problem of sum of squares of integers discussed in Section 5.2. This problem can be coded using the **for** statement as follows:

```
.............
.............
sum = 0;
for  (n = 1; n <= 10; n = n+1)
{
     sum = sum + n*n;
}
.............
.............
```

The body of the loop

```
sum = sum + n*n
```

is executed 10 times for n = 1, 2, …….., 10 each time incrementing the **sum** by the square of the value of **n**.

One of the important points about the **for** loop is that all the three actions, namely *initialization, testing* and *incrementing*, are placed in the **for** statement itself, thus making them visible to the programmes and users, in one place. The **for** statement and its equivalent of **while** and **do** statements are shown in Table 5.1.

Table 5.1 Comparison of the Three Loops

for	*while*	*Do*

`for (n=1;n<=10;++n)`
`{`
`.............`
`.............`
`}`
	`n = 1`
	`while (n<=10)`
	`{`
	`.............`
	`.............`
	`n = n+1;`
	`}`
		`n = 1`
		`do`
		`{`
		`.............`
		`.............`
		`n = n+1`
		`}`
		`while (n<=10);`

Program 5.3 illustrates the use of **for** loop for computing and printing the "power of 2" table.

Program 5.3 Computing the 'power of 2' using for loop

```
Class ForTest
{
   public static void main(String args[ ])
   {
      long p;
```

```
      int n;
      double q;
      System.out.println("2 to power –n  n  2 to power n");
      p = 1;
      for (n = 0; n < 10; ++n)
      {
         if (n == 0)
            p = 1;
         else
            p = p * 2;
         q = 1.0 / (double)p;
         System.out.println(" " + q + " " + n " " + p);
      }
   }
}
```

Output of Program 5.3 would be;

```
2 to power–n        n          2 to power n
    1               0              2
    0.5             1              2
    0.25            2              4
    0.125           3              8
    0.625           4              16
    0.03125         5              32
    0.15625         6              64
    0.00390625      7              128
    0.00195313      8              256
    0.00195313      9              512
```

Additional Features of for Loop

The **for** loop has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the **for** statement. The statements

```
p = 1;
for (n=0; n<17; ++n)
```

can be rewritten as

```
for (p=1, n=0; n<17; ++n)"
```

Notice that the initialization section has two parts p = 1 and n = 1 separated by a *comma*.

Line the initialization section, the increment section may also have more than one part. For example, the loop

```
for (n=1, m=50; n<=m; n=n+1, m=m–1)
{
     ..............
     ..............
}
```

is perfectly valid. The multiple arguments in the increment section are separated by *commas*.

The third feature is that the test condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example that follows:

```
sum = 0;
for (i = 1, i < 20 && sum < 100; ++i)
{
      ..............
      ..............
}
```

The loop uses a compound test condition with the control variable I and external variable **sum**. The loop is executed as long as both the conditions **i** < 20 and sum < 100 are true. The sum is evaluated inside the loop.

It is also permissible to use expressions in the assignment statement of initialization and increment sections. For example, a statement of the type

```
for (x = (m+n)/2; x > 0; x = x/2)
```

is perfectly valid.

Another unique aspect of **for** loop is that one or more sections can be omitted, if necessary. Consider the following statements:

```
..............
..............
m = 5;
for ( ;m != 100 ; )
{
      System.out.println(m);
      m = m+5;
}
..............
..............
```

Both the initialization and increment sections are omitted in the **for** statement. The initialization has been done before the **for** statement and the control variable in incremented inside the loop. In such cases, the sections are left blank. However, the semicolons separating the sections must remain. If the test condition is not present, the **for** statement sets up an infinite loop.

We can set up time delay loops using the null statement as follows:

```
for (j = 10000; j > 0; j = j-1)
    ;
```

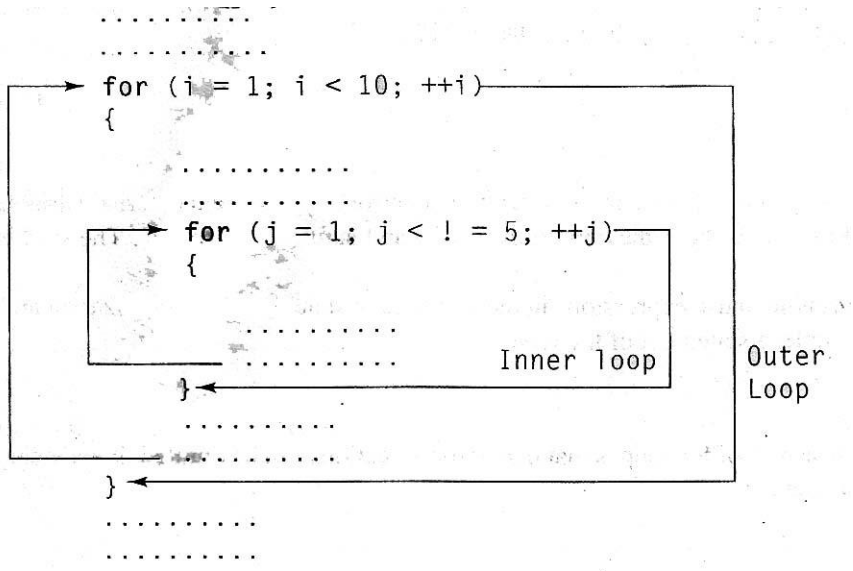
This loop is executed 1000 times without producing any output; it simply causes a time delay. Notice that the body of the loop contains only a semicolon, known as a *empty* statement. This can also be written as

```
for (j=1000; j > 0; j = j-1);
```

This implies that the compiler will not give an error message if we place a semicolon by mistake at the end of a **for** statement. The semicolon will be considered as an *empty* statement and the program may produce some nonsense.

Nesting of for Loops

Nesting of loops, that is, one **for** statement within another **for** statement, is allowed in Java. We have used this concept in Program 5.2. Similarly, **for** loops can be nested as follows:



The loops should be properly indented so as to enable the reader to easily determine which statements are contained within each **for** statement.

A program segment to print a multiplication table using **for** loops in shown below:

```
...........
...........
for (row = 1; row <= ROWMAX: ++row)
{
    for (column = 1; column <= COLMAX : ++column)
    {
          Y = row * column
          System.out.print(“ ” + Y);
    }
    System.out.println(“ ” );
}
...........
...........
```

The outer loop controls the rows while the inner one controls the columns.

The Enhanced for Loop

The enhanced **for** loop, also called *for each loop*, is an extended language feature introduced with the J2SE 5.0 release. This feature helps us to retrieve the array of elements efficiently rather than using array indexes. We can also use this feature to eliminate the interators in a for loop and to retrieve the elements from a collection. The enhanced for loop takes the following form:

```
for (Type Identifier : Expression)
{
    //statements;
}
```

Where, *Type* represents the data type or object used; *Identifier* refer to the name of a variable; and *Expression* is an instance of the java.lang.Iterable interface or an array.

For example, consider the following statements:

```
int nummary[3] = {56, 48, 79};
for (int k=0; k<=3; k++)
```

```
{
   if (nummary[k]>50 && numarray[k]<100)
   {
      System.out.println("The selected value is "+numarray[k]);
   }
}
```

Which is equivalent to the following code:

```
int numarray[3] = [56, 48, 79};
for (int k:numarray)
{
   if (k>50 && k<100)
   {
      System.out.println("The selected value is "+k);
   }
}
```

Thus, we can use the enhanced for loop to track the elements of an array efficiently. In the same manner, we can track the collection elements using the enhanced for loop as follows:

```
Stack samplestack = new Stack();
samplestack.push(new Integer(56));
samplestack.push(new Integer(48));
samplestack.push(new Integer(79));
for (Object obj : samplestack)
{
    System.out.println(obj);
}
```

Program 5.4 Use of enhanced for loop to retrieve the elements of arrays

```
import java.util.*;
class EnhanceForLoop
{
   public static void main(String args[])
   {
      System.out.println();
      String
      States[ ] = {"TamilNadu" , "AndhraPradesh", "UttarPradesh",
                   "Rajasthan"};
      for(int i=0; 1<states.length; i++)
      {
          System.out.println("Standard for-loop : state name : "+states[i]);
      }
      System.out.println();
          for(String i:states)    // enhanced for loop
      {
```

```
        System.out.println("Enhanced for-loop : state name : " + i);
    }
    System.out.println();
    ArrayList<String> cities = new ArrayList<String>();
        Cities.add("Delhi");
        Cities.add("Mumbai");
        Cities.add("Calcutta");
        Cities.add("Chennai");
        System.out.println();
    for(int i=0; i<cities.size(); i++)
    {
        System.out.println("Standard for-loop : city name : "+cities.get(i));
    }
    System.out.println();
        for(String city : cities)    // enhanced for loop
            System.put.println("Enhanced for-loop : city name : "+city);
  System.put.println();
  System.put.println("In Collections");
  System.put.println();
    printcollection(cities);
  }
  Public static<AnyType> void printcollection(Collection<AnyType< c)
  {
    for (AnyType val : c)
    System.put.println(val);
  }
{
```

The output of the above program is as follows:

```
Standard for-loop : state name : TamilNadu
Standard for-loop : state name : AndhraPradesh
Standard for-loop : state name : UttarPradesh
Standard for-loop : state name : Rajasthan

Enhanced for-loop : state name : TamilNadu
Enhanced for-loop : state name : AndhraPrades
Enhanced for-loop : state name : UttarPradesh
Enhanced for-loop : state name : Rajasthan

Standard for-loop : state name : Delhi
Standard for-loop : state name : Mumbai
Standard for-loop : state name : Calcutta
Standard for-loop : state name : Chennai

Enhanced for-loop : state name : Delhi
Enhanced for-loop : state name : Mumbai
Enhanced for-loop : state name : Calcutta
Enhanced for-loop : state name : Chennai
```

```
In Collections:

Delhi
Mumbai
Calcutta
Chennai
```

Jumps in Loops

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. For example, consider the case of searching for a particular name in a list containing, say, 100 names. A program loop written for reading and testing the names a 100 times must be terminated as soon as the desired name is found. Java permits a jump from one statement to the *end* or *beginning* of a loop as well as a *jump out of a loop*.

Jumping Out of a Loop

An early exit from a loop can be accomplished by using the **break** statement. We have already seen the use of the **break** in the **switch** statement. This statement can also be used within **while**, **do**, or **for** loops as illustrated in Fig. 5.2.

When the **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop.

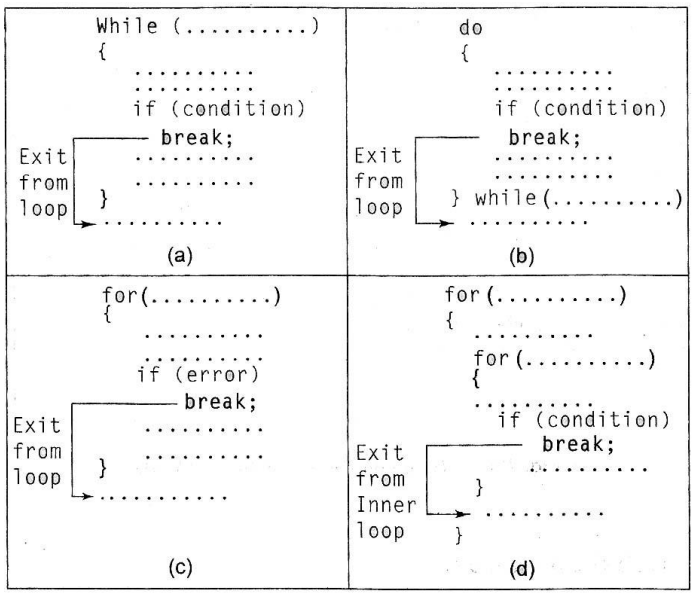


Fig. 5.2 Exiting a loop with break statement

Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category. On reading

the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

Like the **break** statement, Java supports another similar statement called the **continue** statement. However, unlike the **break** which causes the loop to be terminated, the **continue** as the name implies, causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler. "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the **continue** statement is simply

```
Continue;
```

The use of the **continue** statement in loop is illustrated in Fig. 5.3. In **while** and **do** loops, **continue** causes the control to go directly to the *test condition* and then to continue the iteration process. In the case of **for** loop, the *increment* section of the loop is executed before the *test condition* is evaluated.

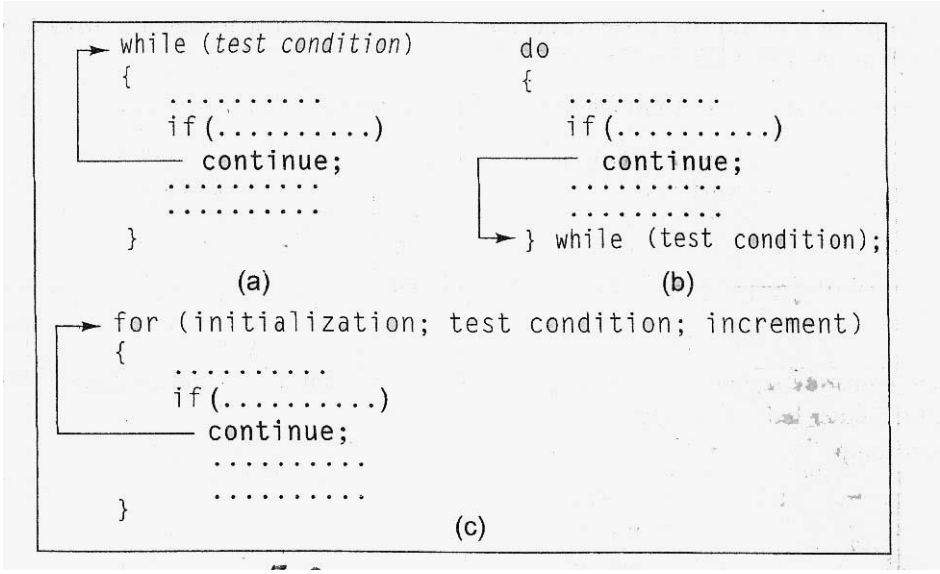


Fig. 5.3 Bypassing and continuing in loops

Labelled Loops

In Java, we can give a label to a block of statements. A label is any valid Java variable name. To give a label to a loop, place it before the loop with a colon at the end. Example:

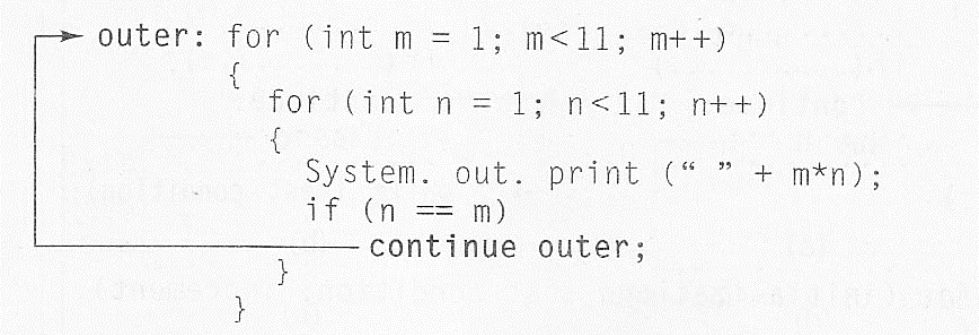
```
loop 1:     for (.............)
            {
                  .............
                  .............
            }
            .............
```

A block of statements can be labelled as shown below:

```
block1:     {
                        .............
                        .............
            block2:     {
                              .............
                              .............
                        }
                        .............
```

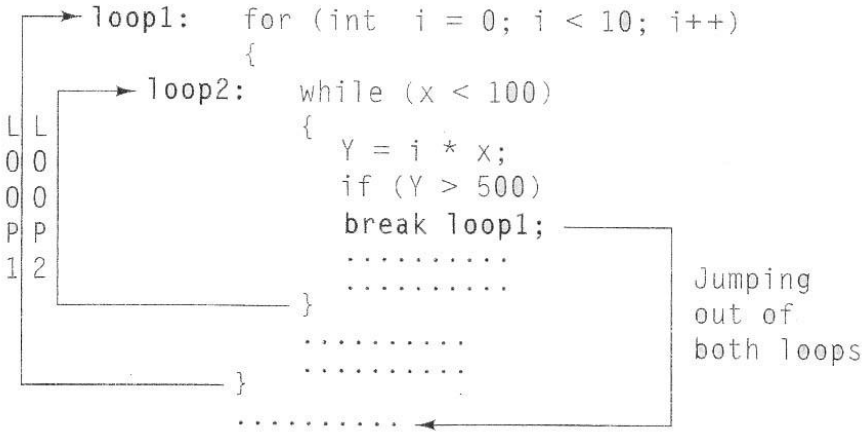
```
            .....
        }
```

We have seen that a simple **break** statement causes the control to jump outside the nearest loop and a simple **continue** statement restarts the current loop. If we want to jump outside a nested loops or to continue a loop that is outside the current one, then we may have to sue the labelled **break** and labelled **continue** statement. Example



Here, the **continue** statement terminates the inner loop when n = m and continues with the next iteration of the outer loop (counting m).

Another example:



Here, the label **loop1** labels the outer loop and therefore the statement

```
break loop1;
```

causes the execution to break out of both the loops. Program 5.4 illustrates the use of **break** and **continue** statements.

Program 5.5 Use of continue and break statements

```
class ContinueBreak
{
   public static void main(String args[])
   {
      LOOP1 : for (int i = 1; i < 100; i++)
              {
                  System.out.println(" ");
                  if (I >= 10) break;
                  for (int j = 1; j < 100; j++)
                  {
                      System.out.print (" * ");
                           if (j == i)
                                continue LOOP1;
```

```
            }
        }

}
```

Program 7.5 produces the following output:

```
*
*     *
*     *     *
*     *     *     *
*     *     *     *     *
*     *     *     *     *     *
*     *     *     *     *     *     *
*     *     *     *     *     *     *     *
*     *     *     *     *     *     *     *     *
```