CHAPTER 6

CONTROL STATEMENTS IN JAVA

A Java statement is the smallest unit that is a complete instruction in itself. Statements in Java generally contain expressions and end with a semi-colon. The two most commonly used statements in any programming language are as follows:

- **Sequential statements**: These are the statements which are executed one by one.
- **Control statements**: These are the statements that are executed randomly and repeatedly.

These are followed by Java also. Now, let us see some statements:

```
System.out.println("Hello");
x = y+z;
System.out.println(x);
```

These statements are executed by JVM one by one in a sequential manner. So they are called *sequential statements*. But this type of sequential execution is useful only to write simple programs. If we want to write better and complex programs, we need better control on the flow of execution. This is possible by using *control statements*.

Important Interview Question

What are control statements?

Control statements are the statements which alter the flow of execution and provide better control to the programmer on the flow of execution. They are useful to write better and complex programs.

The following control statements are available in Java:

- `if ...else` statement
- `do...while` loop
- `while` loop
- `for` loop
- `for-each` loop
- `switch` statement
- `break` statement
- `continue` statement

- `return` statement

> **Note**
>
> *A statement represents a single time execution from top to bottom and a loop represents repeated execution of several statements.*

if...else Statement

This statement is used to perform a task depending on whether a given condition is true or false. Here, *task* represents a single statement or a group of statements.

Syntax:

```
if(condition)
                    statements1;
            [else statements2;]
```

Here, the condition is written inside the small braces `()`. The statements written inside the square brackets `[ ]` represent optional part of the statement. It means that the part within `[ ]` can be omitted, if not required. This is the convention followed in all control statements.

By observing the syntax, we can understand that if the condition specified after `if` is true, then `statements1` will be executed. If the condition is false, then `statements2` will be executed. `statements1` and `statements2` represent either a single statement or more than one statement. If more than one statement is used, then they should be enclosed in angular brackets `{ }`.

Let us write a program to understand the use of `if...else` statement.

Program 1: Write a program to test if a number is positive or negative.

Here, actually three combinations arise: +ve, -ve, and neither +ve nor –ve, i.e. zero.

```
//To test if a number is +ve or -ve
class Demo
{
    public static void main(String[] args)
    {
 int num = -5;  //declare and initialize num to -5

      if(num == 0)
          System.out.println("It is zero");
      else if(num > 0)
          System.out.println(num+" is positive");
      else System.out.println(num+" is negative");
    }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
-5 is negative
```

`if...else` statement can be written in the following different variations as:

```
 if(condition1)
       statements1;
       else if(condition2)
           statements2;
           else if(condition3)
```

```
                    statements3;
                      else statements4;
```

Here, if condition1 is true, then statements1 is executed. If condition1 is false, then condition2 is tested. If condition2 is true, then statements2 is executed, otherwise condition3 is tested.

```
if(condition1)
      if(condition2)
            if(condition3)
                  statements1;
            else statements2;
       else statements3;
    else statements4;
```

Here, if condition1 is true, then condition2 is tested. If condition2 is also true, then condition3 is tested. If condition3 is true, then statements1 will be executed. If condition3 is false, then statements2 will be executed. If condition2 is false, then statements3 will be executed. If condition1 itself is false, then statements4 will be executed.

do...while Loop

This loop is used when there is a need to repeatedly execute a group of statements as long as a condition is true. If the condition is false, the repetition will be stopped and the flow of execution comes out of do...while loop.

Syntax:

```
do{
                    statements;
                }while(condition);
```

We need not use { and } braces, if there is only one statement inside the do...while loop.

Program 2: Write a program to display numbers from 1 to 10.

```
//To display numbers from 1 to 10
class Demo
{
    public static void main(String[] args)
  {
 int x;
       x = 1;    //starting number is 1

       do{
          System.out.println(x);
          x++;
       }while(x<=10);
   }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
1
2
3
4
5
```

```
6
7
8
9
10
```

In the preceding program, observe the loop:

```
do{
        System.out.println(x);  //display x value
        x++;  //increment x value by 1
    }while(x<=10);  //as long as x <=10
```

Here, already the value of x is 1, so it is displayed. Then x++ will increment the value of x by 1, hence the value of x becomes 2. Then the condition x<=10 is tested. As this condition is true, the flow of execution will go back and the value of x, i.e. 2 will be displayed. Then the x value is incremented and becomes 3. Since 3 is also <= 10, the flow goes back and displays x value. In this way, as long as x value does not exceed 10, the loop repeats and hence we can see the numbers from 1 to 10.

while Loop

The functioning of this loop is also similar to do...while loop. This loop repeats a group of statements as long as a condition is true. Once the condition is false, the loop is terminated.

Syntax:

```
while(condition)
        {
            statements;
        }
```


In a do...while loop, the statements are executed first and then the condition is tested. Whereas in a while loop, the condition is tested first; if it is true, then only the statements are executed.

Important Interview Question

Out of do...while *and* while *– which loop is efficient?*

In a do...while *loop, the statements are executed without testing the condition, the first time. From the second time only the condition is observed. This means that the programmer does not have control right from the beginning of its execution. In a* while *loop, the condition is tested first and then only the statements are executed. This means it provides better control right from the beginning. Hence,* while *loop is more efficient than* do...while *loop.*

Program 3: Let us rewrite the previous program (Program 2) using while loop.

```
//To display numbers from 1 to 10
class Demo
{
    public static void main(String[] args)
    {
  int x;
        x = 1;    //starting number is 1

        while(x<=10)
        {
            System.out.println(x);  //display x
            x++;  //increment x
```

```
            }
        }
    }
```

In this program, we can also rewrite the `while` loop as:

```
while(x<=10)
        System.out.println(x++);  //display x and increment it
```

for Loop

The `for` loop is also same as `do...while` or `while` loop, but it is more compact syntactically. The `for` loop executes a group of statements as long as a condition is true.

Syntax:

```
for ( expression1; expression2; expression3 )
                {
                    statements;
                }
```

To understand the preceding syntax, let us take an example:

```
for( int x = 1;  x <= 10; x++)
{
        System.out.println(x);
}
```

Please compare the expressions in the preceding `for` loop. The first expression represents an initialization expression (`int x = 1`). The second one is a conditional expression (`x<= 10`). As long as this condition is true, the statements inside `for` loop are executed. The third expression is a modifying expression (`x++`). It may increment or decrement the value of the variable.

Now, let us see how this `for` loop gets executed. `expression1` will be executed only once the first time. So, the value of `x` becomes 1. Then `expression2` (`x<=10`) will be evaluated. If it is true, then the statements inside the `for` loop will be executed. It means that the statement:

```
System.out.println(x);
```

is executed. So, the value of `x`, i.e.1 will be displayed. Then the third expression (`x++`) will be executed. Now, the `x` value becomes 2. Second expression is again executed, comparing the `x` value. Since `x<=10` are true, then the statement:

```
System.out.println(x);
```

will be executed again. So, the `x` value `2` will be displayed. The preceding statement repeatedly executes till the value of `x` reaches 10. As a result, `x` values from `1` to `10` will be displayed. From this discussion, we can understand that `expression1` is executed only once in the beginning of the `for` loop. Then `expression2` and `expression3` will be executed repeatedly as long as `expression2` is true.

`Do...while` or `while` loops are used when we do not know how many times we have to execute the statements. It just depends on the condition. The execution should continue till the condition is

false. But, the `for` loop is more suitable for situations where the statements should be executed a fixed number of times. Here, we know how many times exactly we want to execute.

Let us now execute `for` loop to see its effect.

Program 4: Write a program to display the numbers from 1 to 10.

```
//To display numbers from 1 to 10
class Demo
{
    public static void main(String[] args)
  {
 for( int x = 1;  x <= 10; x++)
     {
         System.out.println(x);
     }
  }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
1
2
3
4
5
6
7
8
9
10
```

We can write the `for` loop without `expression1` or `expression2` or `expression3` or any two expressions or any three expressions and still we can get the same results. As an example, let us rewrite the preceding `for` loop without `expression1` as:

```
int x = 1;
for(;  x <= 10; x++)
{
     System.out.println(x);
}
```

Let us write the same `for` loop without `expression3` as:

```
int x = 1;
for(;  x <= 10; )
{
     System.out.println(x);
     x++;
}
```

Let us now eliminate the second expression also and write it as:

```
int x = 1;
for( ;  ; )
{
     System.out.println(x);
     x++;
}
```

Observe here, there is no condition in the loop that tells where to stop. So the preceding code executes without stoppage. It is called an infinite loop. Infinite loops are drawbacks in a program because when the user is caught in an infinite loop, he would be perplexed and could not understand how to come out of it. So, it is the duty of the programmer to see not to form the infinite loops. By chance, if the programmer got an infinite loop, he should break it when the condition is reached. For this purpose, `break` statement can be used.

Program 5: Write a program to display numbers from 1 to 10 using infinite `for` loop.

```
//To display numbers from 1 to 10
class Demo
{
    public static void main(String[] args)
  {
      int x = 1;
  for( ; ; )
    {
        System.out.println(x);
        x++;
        if(x > 10) break;   //if x value exceeds 10, then come out of the
loop.
    }
  }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
1
2
3
4
5
6
7
8
9
10
```


`break` is a statement that can be used to come out of a loop. It can be a `for`, `while` or `do...while` loop. The infinite loops can be formed by using not only `for` loop but also `while` or `do...while`, as shown here:

```
while(true)
{
    statements;
}

do{
    Statements;
}while(true);
```

There is another way of writing a `for` loop. We can use multiple initialization expressions and modifying expressions in the `for` loop, as shown here:

```
for(int i=1,j=5; i<=5; i++, j--)
      System.out.println(i+"\t"+j);
```

In the preceding loop, we used two initialization expressions (`i=1, j=5`) and two modifying expressions (`i++,j--`), but there is only one conditional expression (`i<=5`). This `for` loop displays `i`

values from 1 to 5 whereas j values will simultaneously change from 5 to 1. So, the output of executing the preceding for loop will be:

```
1    5
2    4
3    3
4    2
5    1
```

Nested for Loops

We can write a for loop within another for loop. Such loops are called *nested loops*.

```
for(int i=1; i<=3; i++)
{
      statements1;  //these are executed 3 times
}
```

The preceding for loop gets executed for 3 times by changing i values from 1 to 3. Let us take another for loop as:

```
for(int j=1; j<=4; j++)
{
       statements2;   //these are executed 4 times

}
```

This loop is executed 4 times by changing j values from 1 to 4. If we write this loop inside the preceding for loop, it looks like this:

```
for(int i=1; i<=3; i++)
{
   statements1;   //these are executed 3 times
   for(int j=1; j<=4; j++)
   {
        statements2;    //these are executed 12 times
   }
}
```

In this case, the execution starts from the outer for loop and hence i=1. Then statements1 will be executed once. Now, the execution enters second for loop and j value will be 1. Now statements2 will be executed once. After this, j value will be 2 and statements2 will be executed again. Like this, the inner for loop is executed 4 times, with j values changing from 1 to 4. This means statements2 will be executed 4 times.

When the inner for loop is completed, then the execution goes to the outer for loop and i value will be 2. This time, the execution again comes into the inner for loop and statements2 will be executed 4 times. Then the execution goes to outer for loop and i value will be 3. Again the inner for loop is executed 4 times. It means the i and j values will change like this:

```
i=1,  j=1,2,3,4
i=2,  j=1,2,3,4
i=3,  j=1,2,3,4
```

The preceding sequence represents that the outer for loop is executed totally 3 times and hence statements1 will be executed 3 times. The inner for loop is executed 4 times for each i value and hence statements2 will be executed 12 times.

In the same way, it is also possible to write a `while` loop or a `do...while` loop inside a `for` loop and vice versa. These are called *nested loops*.

Program 6: Write a program to display stars in a triangular form—a single star in the first line, two stars in the second line, three stars in the third line, and so on.

```
//To display stars in triangular form - nested for loops
class Stars
{
  public static void main(String args[])
  {
     int r = 5; //we want 5 rows

     for(int i=1; i<=r; i++)  //i represents row number
    {

         for(int st=1; st<=i; st++)   //st represents no. of  stars
         {
            System.out.print(" * ");
         }
        System.out.println();
    }
  }
}
```

Output:

```
C:\> javac Stars.java
C:\> java Stars
 *
 *  *
 *  *  *
 *  *  *  *
 *  *  *  *  *
```


for-each Loop

This loop is specifically designed to handle the elements of a collection. Collection represents a group of elements. For example, we can take an array as a collection or any class in `java.util` package can be considered as a collection. The reason is that an array stores a group of elements like integer values or strings. Similarly, `java.util` package classes are developed to handle a group of objects.

Important Interview Question

What is a collection?

A collection represents a group of elements like integer values or objects. Examples for collections are arrays and `java.util` classes (Stack, LinkedList, Vector, etc.)

The `for-each` loop repeatedly executes a group of statements for each element of the collection. It executes as many times as there are number of elements in the collection.

Syntax:

```
for(var : collection)
                {
                    statements;
                }
```

Here, the var is attached to the collection. This var represents each element of the collection one by one. Suppose, the collection has got 5 elements then this loop will be executed 5 times and the var will represent these elements one by one.

Program 7: Write a program to see the use of for-each loop and retrieve the elements one by one from an array and display it.

```
//Using for-each loop - to display array elements
class Demo
{
  public static void main(String args[]) *
  {
       //declare an array with 5 elements
       int arr[] = {200, 19, -56, 44, 99};

       //use for each to retrieve elements from array
      for(int i : arr)
     {
          System.out.println(i);          //i represents each element of array
     }
  }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
200
19
-56
44
99
```


switch Statement

When there are several options and we have to choose only one option from the available ones, we can use switch statement. Depending on the selected option, a particular task can be performed. A task represents one or more statements.

Syntax:

```
switch(variable)
                {
                    case value1 : statements1;
                    case value2 : statements2;
                    case value3 : statements3;
                    :
                    case valuen : statementsn;
                    [default     : default_statements;]
                }
```

Here, depending on the value of the variable, a particular task (statements) will be executed. If the variable value is equal to value1, statements1 will be executed. If the variable value is value2, statements2 will be executed, and so on. If the variable value does not equal to value1, value2,...then none of the statements will be executed. In that case, default clause is executed and hence the *default_statements* are executed.

Let us take an example to understand the switch statement. In Program 8, we are taking a variable color, which is initialized to g. Depending on the color value, Red is displayed when the value is r; Green is displayed when the value is g; Blue is displayed when the value is b; and White is

displayed when the value is w. If color value is neither of the specified values—r, g, b, or w, then none of the statements are executed. As a result, the default statement is executed and it displays No color.

Program 8: Write a program for using the switch statement to execute a particular task depending on color value.

In this program, since color value is g, we expect that it displays Green as output.

```
//To display a color name depending on color value
class Demo
{
  public static void main(String args[])
 {
     char color = 'g';  //color is set to 'g'

     switch(color)
   {
     case 'r': System.out.println("Red");
     case 'g': System.out.println("Green");
     case 'b': System.out.println("Blue");
     case 'w': System.out.println("White");
     default : System.out.println("No color");
   }
 }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
Green
Blue
White
No color
```


The output of the program is not as expected. We expected that it would display Green, but it is displaying all colors starting from the Green color. What might be the reason? When color value is g, it has displayed Green and after that it has come down to execute the rest of the statements under it leading to the preceding output. The solution is to come out of the switch statement, after displaying Green. For this purpose, we can use break statement.

One of the uses of break is to terminate a loop and come out of it. Another use of break statement is to come out of the switch block. Now, let us rewrite the preceding program using the break statement.

Program 9: Write a program to come out of switch block, after executing a task.

In the following case, g becomes true and hence, JVM displays Green and then executes break, which terminates the switch block.

```
// To display a color name depending on color value
class Demo
{
  public static void main(String args[])
 {
     char color = 'g';  //color is set to 'g'

     switch(color)
   {
     case 'r': System.out.println("Red");
               break;
     case 'g': System.out.println("Green");
               break;
```

```
            case 'b': System.out.println("Blue");
                      break;
            case 'w': System.out.println("White");
                      break;
            default : System.out.println("No color");
        }
    }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
Green
```

Remember, we cannot use all the data types along with the `switch` statement. We can use `char`, `int`, `byte`, `short` types only. For example, the following `switch` statement is invalid:

```
String str= "Delhi";
switch(str)   //invalid - String cannot be used with switch.
```

`Switch` statements are mostly used in menu driven programs. A *menu driven* program is a program where a menu (a list of items) is displayed and the user can select an item from the list of items available in the menu. Depending on the choice of the user, a particular task is done.

```
switch(user_choice)
{
  case 1:  task1;
  case 2: task2;
  :
}
```


break Statement

The `break` statement can be used in 3 ways:

- `break` is used inside a loop to come out of it.
- `break` is used inside the `switch` block to come out of the `switch` block.
- `break` can be used in nested blocks to go to the end of a block. *Nested blocks* represent a block written within another block.

We have already observed the first two uses of `break` statement. Now, let us see the third use, where `break` statement is used inside the nested blocks.

```
break label;  //here label represents the name of the block.
```

The meaning of the preceding statement is *go to the end of the block*, whose name is given by the label. See the following example to understand this.

Program 10: Write a program to use a `break` statement to go to the end of a block.

Here we are using break to go to the end of the `Block2`.

```
//labeled break to go to end of a block
class Demo
{
  public static void main(String args[])
 {
```

```
        boolean x = true;

      bl1:{
         bl2:{
            bl3:{
               System.out.println("Block3");
               if(x) break bl2; //goto end of bl2
            } //end of bl3
            System.out.println("Block2");
         } //end of bl2
         System.out.println("Block1");
      } //end of bl1
      System.out.println("Out of all blocks");
    }
  }
```

Output:

```
C:\>javac Demo.java
C:\>java Demo
Block3
Block1
Out of all blocks
```

In the preceding program, `bl1`, `bl2`, `bl3` are names of the blocks starting with a `{` and ending with a `}`. First, `Block3` will be displayed as the control of execution enters that block. Then, it executes:

```
if(x) break bl2;
```

This represents going to the end of the block, named `bl2`. After the closing `}`, we got `Block1`, which is displayed and then the control jumps out and executes the last statement, so `Out of all blocks` will be displayed.

This type of break resembles `goto` statement in C/C++. The `goto` statement is useful to jump directly from one statement to another statement in the program, but `goto` is not available in Java due to its certain shortcomings:

- `goto` statements, more often than not, create infinite loops— which are drawbacks in a program.
- `goto` statements make documentation of a program difficult. Documenting a program means preserving a copy of the details of the development of a program. Documentation contains algorithm or logic of the program, flow chart representing the flow of execution and program print out as well as its output. A brief description of the program is done at the end of documentation representing the entire process of development of the program. When several `goto` statements are used, algorithms, flow charts, and explanation of the program will become unclear and hence the documentation becomes useless.
- `goto` statements are not a part of structured programming. The principles of structured programming do not include `goto` statements as part of elements of programming. This means, it is perfectly possible to write a program without ever using any `goto` statement.

Because of the preceding reasons, the use of `goto` statements has been banned in programming.

Important Interview Question

Why `goto` statements are not available in Java?

`goto` statements lead to confusion for a programmer. Especially, in a large program, if several `goto` statements are used, the programmer would be perplexed while understanding the flow from where to where the control is jumping.

continue Statement

`continue` is used inside a loop to repeat the next iteration of the loop. When `continue` is executed, subsequent statements in the loop are not executed and control of execution goes back to the next repetition of the loop.

Syntax:

```
continue;
```

To understand the use of `continue`, let us write a program using a `for` loop to display the numbers in descending order from 10 to 1.

Program 11: Write a program using `for` loop to display the numbers in descending order.

```
//Numbers in descending order
class Demo
{
  public static void main(String args[])
  {
     for(int i=10; i>=1; i--)
     {
        System.out.print(i+"  ");
     }
  }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
10  9  8  7  6  5  4  3  2  1
```

In this program, `i` value starts at `10`. As long as `i` value is greater or equal to `1`, it is decremented by `1`. So we get numbers from 10 to 1 in descending order. Now, let's introduce `continue` in this program as:

```
for(int i=10; i>=1; i--)
    {
       if(i>5) continue;  //go back in the loop
       System.out.print(i+"  ");
    }
```

Here, we are redirecting the flow of execution back to the next iteration of the loop when `i>5`. So when `i` value changes from 10 to 6, `continue` statement will be executed and hence the subsequent statement:

```
System.out.print(i+"  ");
```

will not be executed. So the values of `i` from 10 to 6 will not be displayed, the output will be as follows:

```
5  4  3  2  1
```

The `continue` statement can be used along with a label, like `break` statement as:

```
continue label;  //here label represents name of the loop
```

In this case, the label after `continue` represents the name of the loop to where the flow of execution should jump. This is also called *labeled continue* statement.

Let us write a program using a `for` loop inside a `while` loop, which displays `i` and `j` values. It means we are using nested loops. In this program, we can use a label (name) to represent the `while` loop as `lp1` and another label to represent the `for` loop as `lp2`.

Program 12: Write a program for using nested loops (to display `i` and `j` values).

In this program, `i` values change from `1` to `3`. When `i` is `1`, then `j` values change from `1` to `5`; when `i` is `2`, `j` values change from `1` to `5`; and when `i` is `3`, `j` values change from `1` to `5` again.

```
//Using nested loops with labels.
class Demo
{
  public static void main(String args[])
 {
    int i=1,j;

   lp1: while(i<=3)
   {
     System.out.print(i); //i values change from 1 to 3
    lp2: for(j=1; j<=5; j++)
     {
        System.out.println("\t"+j); //j values from 1 to 5 for every i value
     }
     i++;
     System.out.println("-------------");
   }
  }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo

1       1
        2
        3
        4
        5
-------------
2       1
        2
        3
        4
        5
-------------
3       1
        2
        3
        4
        5
-------------
```

Now, in the same program, let us introduce a labeled `continue` statement as follows:

```
lp1: while(i<=3)
   {
     System.out.print(i); //i values change from 1 to 3
  lp2: for(j=1; j<=5; j++)
     {
        System.out.println("\t"+j);
        if (j==3) //j values change up to 3 only
```

```
            {
              i++;
              continue lp1; //go back to while loop
            }
        }
        i++;
        System.out.println("-------------");
```

Output:

```
1    1
     2
     3
2    1
     2
     3
3    1
     2
     3
```

Here, we have cut off the values, i.e. 4 and 5 of j, by comparing j value and redirecting the flow o execution to outer while loop, using the following statement:

```
if (j==3) //if j value is 3, then use continue to go back
        {
          i++;
          continue lp1; //go back to while loop
        }
```

return Statement

We know that a method is a function written inside a class. It contains a group of statements an performs a task or processing. It means a method is useful to perform certain calculations o processing of data in the program to yield expected results. Methods can accept the data fror outside for their processing and they can also return the results.

A method is executed when called from another method. The first method that is executed in a Jav program by the JVM is main() method and hence if we want to execute any other method, w should call it from main().

return statement is used in a method to come out of it to the calling method. For example, we ar calling a method by the name myMethod() from the main() method. If return is used insid myMethod(), then the flow of execution comes out of it and goes back to main(). While going bac from myMethod(), we can also return some value to the main() method. For this purpose, retur should be used as follows:

```
return  1;  //return 1 to calling method
return  x;  //return x value
return  (x+y); //calculate x+y and return that value
return  -5; //return -5
```

In the following example, we have taken a method myMethod() that accepts an integer valu calculates square value of it, and returns that value to the main() method which is the callir method.

In the main() method, we are calling myMethod() and passing 10 to it as follows:

```
Demo.myMethod(10);
```

Here, we are not using an object to call the method. Since it is a static method, we can call it as `classname.methodname()`, i.e. `Demo.myMethod(10)`.

To receive the result returned from the method, we have taken an `int` type variable `res` as follows:

```
int res = Demo.myMethod(10);
```

And we have written `myMethod()` as:

```
static int myMethod(int num)
 {
    return num*num; //return square value
 }
```

Here, we declared the method as `static` and then `int` before the method name represents the type of value returned by the method. After the method name, we wrote `int num`, which is useful to receive the integer number into the method. This is nothing but the value `10` passed to the method at the time of calling it from `main()`. Now to calculate square value and return it, we used:

```
return num*num;
```

Thus, `return` statement is used to return some value from a method. Figure 6.1 will help to comprehend this.

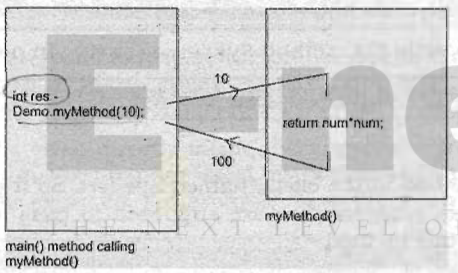

Figure 6.1 Calling a method and getting back result

Program 13: Write a program to return a value from a method.

```
//Calling a method and returning the result from the method.
class Demo
{
  public static void main(String args[])
 {
    //call myMethod() and catch the result into res.
    //since myMethod() is static, we can call it using classname.methodname()
    int res = Demo.myMethod(10);
    //display the result now
    System.out.println("Result= "+ res);
 }

 //this method calculates square value and returns it to main().
 static int myMethod(int num)
 {
    return num*num; //return square value
 }

}
```

Output:

```
C:\> javac Demo.java
```

```
C:\> java Demo
Result = 100
```

If we use `return` statement inside the `main()` method, then the entire program (or application) will be terminated and we come out to the system prompt. Let us try to understand this better with the help of an example.

Program 14: Write a program to demonstrate that the `return` statement in `main()` method terminates the application.

```
//return inside main()
class Demo
{
  public static void main(String args[])
  {
     int x = 1;
     System.out.println("Before return");
     if(x == 1) return;  //terminate the application
     System.out.println("After return");
  }
}
```

Output:

```
C:\> javac Demo.java
C:\> java Demo
Before return
```

In this program, we can also write the method `System.exit(0);` in place of the statement:

```
if(x==1) return;
```

to terminate the application.

Here, `exit(0)` is a static method in the class, named `System`. So it can be called `System.exit(0)` When calling `exit()` method, programmers are supposed to pass an `int` value to this method– generally `0` or `1` is passed to this method.

Important Interview Question

What is the difference between `return` *and* `System.exit(0)` *?*

`return` *statement is used inside a method to come out of it.* `System.exit(0)` *is used in any method to come out of the program.*

While calling `exit()` method of `System` class, we can give either `0` or `1` to it as `exit(0)` or `exit(1)` Both will terminate the program, but `0` or `1` indicates the reason for termination. `exit(0` represents normal termination while `exit(1)` represents termination due to some error in th program.

Important Interview Question

What is the difference between `System.exit(0)` *and* `System.exit(1)` *?*

`System.exit(0)` *terminates the program normally. Whereas* `System.exit(1)` *terminates the program because of some error encountered in the program.*

Conclusion

Control statements are very useful to a programmer for writing better and complex programs, sin they are designed to implement any sort of logic. Whatever the programmer wishes to do in h program, he can do it with the help of control statements. This is the reason we see the contr statements in almost all the languages and not just in Java.