Chapter 3

Control Statements and Array

4.1 Introduction

In Java, program is a set of statements and which are executed sequentially in order in which they appear. In that statements, some calculation have need of executing with some conditions and for that we have to provide control to that statements. In other words, Control statements are used to provide the flow of execution with condition.

In this unit, we will learn the control structure in detail.

4.2 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. In Java control statements are categorized into *selection control* statements, *iteration control* statements and *jump control statements*.

4.2.1 Selection Statements: Java supports two selection statements: if and switch. These statements allow us to control the flow of program execution based on condition.

4.2.1.1 Statement: if statement performs a task depending on whether a condition is true or false.

Syntax: if (condition)

   statement1;

  else

   statement2;

Here, each statement may be a single statement or a compound statement enclosed in
curly braces (that is, a block). The condition is any expression that returns a boolean
value. The else clause is optional.

Program : Write a program to find biggest of three numbers.

```java
//Biggest of three numbers
class BiggestNo
{  public static void main(String args[])
  {  int a=5,b=7,c=6;
    if ( a > b && a>c)
      System.out.println ("a is big");
    else if ( b > c)
      System.out.println ("b is big");
    else
      System.out.println ("c is big");
  }
}
```

Output:


```
D:\JQR>javac BiggestNo.java

D:\JQR>java   BiggestNo
b is big

D:\JQR>_
```

4.2.1.2 Switch Statement:

When there are several options and we have to choose only one option from the available ones, we can use switch statement.

It is used for multiple-way selection that will branch to different code segments based on the value of a variable or an expression. The optional default label is used to specify the code segment to be executed when the value of the variable or expression cannot match any of the case values.

For version of Java prior to JDK 7, expression must be of **type byte, short, int, char** or an enumeration. Beginning with JDK 7, expression can also be of type String. Each value specified in the case statements must be a unique constant expression (such as literal value). Duplicate **case** value are not allowed.

Syntax: switch (expression)

   { case value1: //statement sequence

      break;

   case value2: //statement sequence

      break;

    ……………..

   case valueN: //statement sequence

      break;

   default: //default statement sequence

   }

Here, depending on the value of the expression, a particular corresponding case will be

executed.

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

//To display a color name depending on color value

class ColorDemo

```java
{ public static void main(String args[])

 { char color = 'r';

   switch (color)

   { case 'r': System.out.println ("red");    break;

     case 'g': System.out.println ("green");  break;

     case 'b': System.out.println ("blue");    break;

     case 'y': System.out.println ("yellow");  break;

     case 'w': System.out.println ("white");  break;

     default: System.out.println ("No Color Selected");

   }

 }

}
```

Output:

4.3 Iteration Statements: Java's iteration statements are for, while and do-while. These statements are used to repeat same set of instructions specified number of times called loops.

A loop repeatedly executes the same set of instructions until a termination condition is met.

4.3.1 while Loop: while loop repeats a group of statements as long as condition is true. Once the condition is false, the loop is terminated. In while

loop, the condition is tested first; if it is true, then only the statements are executed. while loop is called as entry control loop.
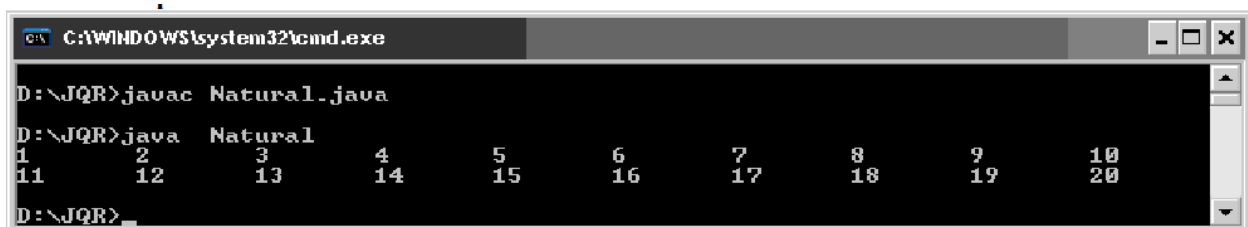
Syntax: while (condition)

```
{
  statements;
}
```

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main(String args[])
  {  int i=21;
    while (i <= 20)
    {  System.out.print (i + "\t");
      i++;
    }
  }
}
```

Output:

4.3.2 do…while Loop: do…while loop repeats a group of statements as long as condition is

true. In do...while loop, the statements are executed first and then the condition is tested.

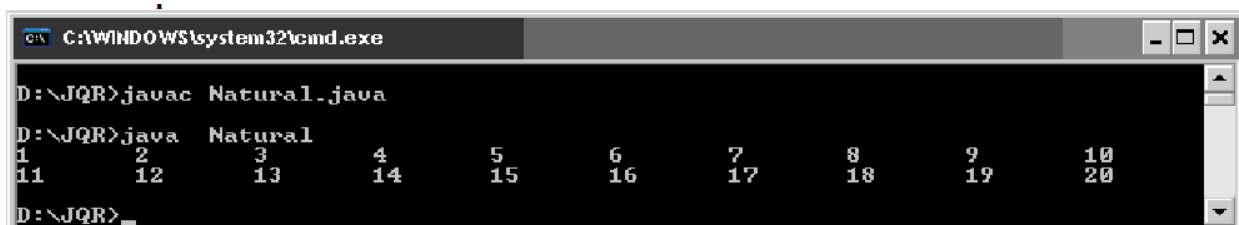do…while loop is also called as exit control loop.

Syntax: do

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

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main(String args[])
   {  int i=1;
      do
      {  System.out.print (i + "\t");
         i++;
      } while (i <= 20);
   }
}
```

Output:

4.3.3 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.

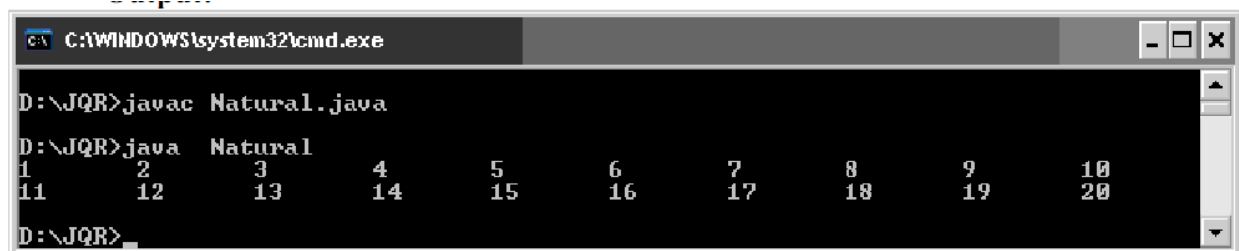**Syntax:** for (expression1; expression2; expression3)

```
{    statements;

}
```

Here, expression1 is used to initialize the variables, expression2 is used for condition
checking and expression3 is used for increment or decrement variable value.

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main(String args[])
  {  int i;
    for (i=1; i<=20; i++)
      System.out.print (i + "\t");
  }
}
```

Output:


```
D:\JQR>javac Natural.java

D:\JQR>java  Natural
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20

D:\JQR>
```

4.4 Jump Statements:
Java supports three jump statements: break, continue and return. These statements transfer control to another part of the program.

4.4.1 break:

- break can be used inside a loop to come out of it.
- break can be 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.

Syntax: break; (or) break label;//here label represents the name of the block.

 Program : Write a program to use break as a civilized form of goto.

```
//using break as a civilized form of goto
class BreakDemo
{  public static void main (String args[])
 {  boolean t = true;
   first:
{
 second:
{
     third:
{
       System.out.println ("Before the break");
       if (t) break second; // break out of second block
         System.out.println ("This won't execute");
      }
       System.out.println ("This won't execute");
     }
     System.out.println ("This is after second block");
   }
  }
 }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe                        _ □ ×

D:\JQR>javac BreakDemo.java

D:\JQR>java  BreakDemo
Before the break
This is after second block

D:\JQR>
```

4.4.2 continue: This statement is useful to continue the next repetition of a loop/ iteration. When continue is executed, subsequent statements inside the loop are not executed.

Syntax: continue;

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
```

```java
{  public static void main (String args[])
  {  int i=1;
    while (true)
    {  System.out.print (i + "\t");
      i++;
      if (i <= 20 )
        continue;
      else
        break;
    }
  }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe                                          _ □ ×

D:\JQR>javac Natural.java

D:\JQR>java  Natural
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20

D:\JQR>
```

Labelled loop:

We can give label to a block of statements with any valid Identifier. Following example shows the use of label, break and continue. One of the most common uses for a labeled break statement is to exit from nested loops. For example, in the following program, the outer loop executes only once:

```java
// Using break to exit from nested loops
class BreakLoop4 {
public static void main(String args[]) {
outer: for(int i=0; i<3; i++) {
System.out.print("Pass " + i + ": ");
for(int j=0; j<100; j++) {
if(j == 10) break outer; // exit both loops
System.out.print(j + " ");
}
}
```

```
    System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
    }
}
```

This program generates the following output:

Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

As you can see, when the inner loop breaks to the outer loop, both loops have been terminated.
Notice that this example labels the **for** statement, which has a block of code as its target.

Keep in mind that you cannot break to any label which is not defined for an enclosing block. For example, the following program is invalid and will not compile:

```
// This program contains an error.
class BreakErr
{
    public static void main(String args[])
    {
        one: for(int i=0; i<3; i++)
        {
            System.out.print("Pass " + i + ": ");
        }
        for(int j=0; j<100; j++)
        {
            if(j == 10) break one; // WRONG
            System.out.print(j + " ");
        }

    }
}
```

Since the loop labeled **one** does not enclose the **break** statement, it is not possible to transfer
control out of that block.

As with the break statement, continue may specify a label to describe which enclosing loop to continue. Here is an example program that uses continue to print a triangular multiplication table for 0 through 9.

```
// Using continue with a label.
class ContinueLabel
{
public static void main(String args[])
{
outer: for (int i=0; i<10; i++) {
for(int j=0; j<10; j++) {
if(j > i) {
System.out.println();
continue outer;
}
System.out.print(" " + (i * j));
}
}
System.out.println();
}
}
```

The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**. Here is the output of this program:
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64

0 9 18 27 36 45 54 63 72 81

4.2.4. For-Each Style of the For Loop

Beginning with JDK 5, a second form of **for** was defined that implements a "for-each" style loop. A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword **foreach**, Java adds the for-each capability by enhancing the **for** statement

The general form of the for-each version of the **for** is shown here:

for(*type itr-var : collection*) *statement-block*

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by collection. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the collection have been obtained.

The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++)
  sum += nums[i];
```

To compute the sum, each element in **nums** is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the **nums** array by **i**, the loop control variable. The for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending

value, and manually index the array. For example, here is the preceding fragment rewritten using a for-each version of the **for**:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
sum += x;
```

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors. Here is an entire program that demonstrates the for-each version of the **for** just described:

```
// Use a for-each style for loop.
class ForEach {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values
for(int x : nums) {
System.out.println("Value is: " + x);
sum += x;
}
System.out.println("Summation: " + sum);
}
}
```

There is one important point to understand about the for-each style loop. Its iteration variable (i.e x in above example) is "read-only". An assignment to the iteration variable has no effect on the underlying array. In other words, you can't change the contents of the array by assigning the iteration variable a new value. For example, consider this program:

```java
// The for-each loop is essentially read-only.
class NoChange {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for(int x : nums)
{
System.out.print(x + " ");
x = x * 10; // no effect on nums
}
System.out.println();
for(int x : nums)
System.out.print(x + " ");
System.out.println();
}
}
```

For-Each Style Over Multidimensional Arrays
The enhanced version of the **for** also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array, because each iteration obtains the *next array*, not an individual element. Furthermore, the iteration variable in the **for** loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions. To understand the implications of this, consider the following program. It uses nested **for** loops to obtain the elements of a two-dimensional array in row order, from first to last.

```java
// Use for-each style for on a two-dimensional array.
class ForEach3 {
public static void main(String args[]) {
```

```java
int sum = 0;
int nums[][] = new int[3][5];
// give nums some values
for(int i = 0; i < 3; i++)
for(int j=0; j < 5; j++)
nums[i][j] = (i+1)*(j+1);
// use for-each for to display and sum the values
for(int x[] : nums) {
for(int y : x) {
System.out.println("Value is: " + y);
sum += y;
}
}

System.out.println("Summation: " + sum);
}
}
```

In the program, pay special attention to this line:

for(int x[] : nums) {

Notice how **x** is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**. The inner **for** loop then cycles through each of these arrays, displaying the values of each element.
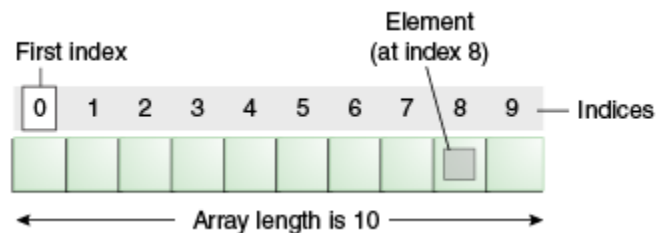
Using Arrays

This section describes basic array operations, including accessing array elements and declaring, creating, and copying arrays. But first, you should understand just what an array is.

What Is an Array?

An *array* is a group of variables of the same type referable by a common name. The type can be either a primitive datatype like int or an object type

like String. An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You have seen an example of arrays already, in the `main` method of the "Hello World!" application. This section discusses arrays in greater detail.

An array of 10 elements.

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

Creating an Array

Like any other variables, array must be declared and created in the memory before they are used. Creation of an array involves two steps:
1. Declaring the array
2. Creating memory locations

Declaration of Array
Array in Java may be declared in two forms:

Form 1:   <data type> arrayname[ ];
Form 2:   <data type> [ ] arrayname;

Examples:
int   list[ ];
int[ ]   num;

Note, here we do not enter the size of the array in the declaration.

Creation of Arrays

After declaring an array, we need to create it in the memory, Java allows us to create arrays using **new** operator only, as shown below:

arrayname = **new** datatype[size];

Examples:

list = new   int[10];
num = new int[5];

These lines create necessary memory locations for the array **num** and **list** and designated them as **int**. Now, the variable num refers to an array of 5 integers and list refers to an array of 10 elements.

It is possible to combine the two steps – declaration and creation- into one as shown below:

int num[ ] = new int[5];

Initialization of Arrays

The final step is to put values into the array created. This process is known as initialization. This is done using the array subscripts as shown below:

Example:
                num[0] = 35;
                num[1] = 40;
                . . . .
                num[4] = 19;

Unlike C, C++ Java protects array from overrun and underruns. Trying to access an array bound its boundaries will generate an error message.

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces., as shown below:

Example:
```
    int num[ ] = { 5, 10, 15, 20, 25};
```

Loops may be used to initialize large size arrays. Example

```
// Average an array of values.
class Average
{
  public static void main(String args[])
  {
    int nums[] = {10, 11, 12, 13, 14};
    double result = 0;
    int i;
    for(i=0; i<5; i++)
      result = result + nums[i];

      System.out.println("Average is " + result / 5);
  }
}
```

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays.

the following declares a two dimensional array variable called **twoD**.

```
int twoD[][] = new int[4][5];
```
This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**.

```
// Demonstrate a two-dimensional array.
class TwoDArray
 {
  public static void main(String args[])
{
  int twoD[][]= new int[4][5];
```

```
  int i, j, k = 0;
  for(i=0; i<4; i++)
  for(j=0; j<5; j++)
   {
     twoD[i][j] = k;
     k++;
   }
  for(i=0; i<4; i++)
   {
     for(j=0; j<5; j++)
       System.out.print(twoD[i][j] + " ");
       System.out.println();
   }
}
}
```
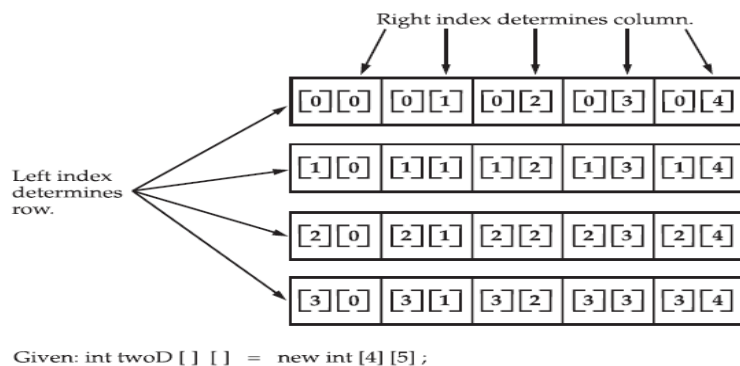
This program generates the following output:
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

When you allocate memory for a multidimensional array, you need only specify the
memory for the first (leftmost) dimension. You can allocate the remaining dimensions

Right index determines column.

Left index determines row.

Given: int twoD [ ] [ ] = new int [4] [5] ;

separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension.

Jagged Arrays

A jagged array is an array that contain a group of array within it. It means we can create an arrayin Java such that other array can become its elements. This is a unique feature in Java. A jagged array can store single dimensional(1 D) array or multi-dimensional (2D, 3D, etc.) arrays and also it can store of any size. Jagged arrays are also called 'irregular multidimensional arrays'. Jagged arrays are useful when dealing with a group of array of different size.

To create a jagged array that can store 4 1D arrays in which the size of the second dimension are unequal, we can write as:

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
```

```
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```


In the same way, we can also create a jagged array that contains 2D arrays. For example,
int arr[][][] = new int[3][][];
arr[0] = new int[2][3];
arr[1] = new int[2][2];
arr[2] = new int[3][2];

Three Dimensional Array (3D array)

We can consider a three dimensional array as a combination of several two dimensional arrays. This concept is useful when we want to handle group of elements belonging to another group. For example, a college has 3 departments: Electronics, computers and Civil. We want to represent the marks obtained

by the students of each department in 3 different subjects. We can write these marks as shown here:

Electronics department:
Student1 marks: 50, 55,60
Student2 marks: 60, 63, 67

Computers department:
Student1 marks: 70, 75,76
Student2 marks: 64, 67, 77

Civil department:
Student1 marks: 56, 57,76
Student2 marks: 77, 67, 87

To store all these marks, department-wise, we need a three dimensional array as shown here:

int arr[][][] = { {{50, 55,60},{64, 67, 77}},
        {{70, 75,76},{64, 67, 77}},
        {{56, 57,76},{77, 67, 87}} };

Here, three of square braces represent a 3D array and there are 3 rows representing the 3 departments. In each department, there are again two group representing the marks of two students. Observe how the {end} curly braces are used to represent each group.

arrayname.length

if we want to know the size of any array, we can use the property 'length' of an array. arrayname.length returns an integer number which represent the size of an array.