Control Statements

Control Statement will alter a program sequence and it is used to make a decision

Type of Control Statements

1. **Simple if Statement**
2. **if…else**
3. **else…if ladder or Mutliple else if**
4. **Nested if**
5. **Switch statement.**
6. **Loops**

4.1 Simple if Statement

If statement, is used to check a condition and execute.

Syntax

```
if (condition)
{
      statement(s);
}
      normal statement;
```

Explanation

When the condition is **true** the **statement(s)** are executed and then control is passed to the statement immediately after the **if statement(s)**. If the condition is **false,** control is passed directly to the statement following the if statement, ie., **Normal statement will execute.**

Program

```
class IfDemo
{
      public static void main(String arg[] )
      {
            int a,b,c;
            a=Integer.parseInt(args[0]);
            b=Integer.parseInt(args[1]);

            if(a>b)
            {
                   big=a;
            }

            if(b>a)
            {
                  big=b;
            }
            System.out.println("Big= "+big);

}
```

OUTPUT

```
C:\>java IfDemo  10 20
```

```
Big=20
```

4.2 If..else statement

In an **if** statement, if the condition is **false**, control is passed to the statement following the **if** statement.

The **if-else** statement, allows you to explicitly specify one or more statements to be executed when the if condition is **false**.

Syntax

```
if(condition)
{
        true block statement(s)
}
else
{
        false block statement(s);
}
      statement-x;
```

Explanation

Here, If the condition is **true**, the **true block** statement(s) are executed and if the condition is **false**, the **false block** statement(s) are executed. Statement -x will execute always after the if ..else block

Program

```
class IfElseDemo
{
      public static void main(String arg[] )
      {
       int a =100 ,b=20,big=0;

            if(a>b)
            {
                  big=a;
            }

            else
            {
                  big=b;
            }
      System.out.println("Big = "+big);

      }
```

OUTPUT

```
C:\>java IfElseDemo  100 20

Big = 100
```

4.3 Else if ladder or Mutliple else if

The conditions are evaluated from the top to bottom with series of conditions.

Syntax

```
if(condition1)
{
       statement1
}
else if(condition2)
{
        statement2
}
else if(condition3)
{
        statement3
}
 ----------------------
 ----------------------
else
{
      default statement
}
 statement-x;
```

As soon as a **true** condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are **true**, the **final else** is executed. That is, if all other conditional tests fail, the last else statement is performed. If the final else is not present, no action takes place if all other conditions are false.

Program

```
class MultipleElseIfDemo
{
public static void main(String arg[])
{
int m1=50,m2=60,m3=70,m4=80,m5=91;
float per=0.0f;
int tot= m1+m2+m3+m4+m5;
per=tot/5.0f;
System.out.println("Total Marks = "+tot);
System.out.println("Percentage = "+per);
            if(per>=75)
            {
                   System.out.println("Distinction");
            }
            else if(per>=60)
            {
                  System.out.println("First Class");
            }
            else if(per>=50)
            {
                  System.out.println("Second Class");
            }
            else if(per>=40)
            {
                    System.out.println("Third Class");
            }
            else
            {
                    System.out.println("No Grade");
            }
 }
 }
```

OUTPUT

```
C:\>java MultipleElseIfDemo

Total Marks = 351
Percentage = 72.2
First Class
```

4.4 Nested if statements

Sometimes, while writing programs, it becomes necessary to check a second condition based on the result of the first. In such cases, the second if statement has to be nested within the first if statement. Such statement are called **Nested if statement**

Syntax

```
            if(condition1)
            {
             .............
             .............
                  if(condition2)
                  {
                         ............
                          ............
                  }
                  else
                  {
                        ..........
                        ...........
                  }
             }
             else
             {
               ..........
               ..........
             }
```

Here, **condition1** is evaluated first. If it is true, then **condition2** is evaluated otherwise control is passed to the last else statement and all the statements in between are skipped. Similarly result of **condition2** determines whether **condition3** is to be evaluated and which set of statements is to be executed.

Program to find biggest of three numbers

```
class NestedIfDemo
{
      public static void main(String arg[] )
      {
            int a =67 ,b=89,c=45,big=0;
            if(a>b)
            {
                  if(a>c)
                        big=a;
                    else
                        big=c;
             }
             else
             {
                  if(b>c)
                        big=b;
                  else
                        big=c;
             }
      System.out.println("Big= "+big);
      }
}
```

OUTPUT

```
C:\>java NestedIfDemo  10 20

Big=89
```

4.5 Switch Statement

Multiple if statement are very useful but they can become complex and difficult to debug as the number of if statements increases. An alternative is to use switch statement. A **switch statement** is used to select one option out of many, based on the value of an expression.

Syntax

```
switch(expression)
{
      case constant1:
                        statement1
                        break;
      case constant2:
                        statement2
                        break;
                        -------------
                        -------------
      case constantn:
                        statement n
                        break;
      default:
                        default statement
                        break;
}
```

In switch statement, expression is compared with each of the constants **constant1**, **constant2**, **constant3**, and so on. When a match is found, the statements following the case are executed. The break statement is used to break out of the switch statement and pass control to the statement following the switch statement.

Program

```
class SwitchDemo
{
      public static void main(String arg[])
      {
            int n = Integer.parseInt(arg[0]);

            switch(n)
            {
            case 1:  System.out.println("The number is one");
                        break;
            case 2:  System.out.println ("The number is two");
                        break;
            case 3:  System.out.println ("The number is three");
                        break;
            default: System.out.println ("Number is not between 1 and 3");
                        break;
            }
            System.out.println ("\nEnd of program");

      }
}
```

OUTPUT

```
C:\> java SwitchDemo 2
The number is two
```

4.6 Loop Control Statements

Computers are best suited for applications where a sequence of steps has to be repeated several times. The program construct that is used to repeat such a sequence of steps is called as **loop.**

Every loop consists of condition that determines the number of times those statements are to be repeated.

There are three different loop statements in Java, they are

1. **while Loop**
2. **do-while Loop**
3. **for Loop**

4.6.1 While Loop

The while loop executes a set of code repeatedly until the condition returns false.

Syntax

```
while(condition)
{
      statement(s);
```

```
		}
```

This loop is executed as long as the condition is true. When the condition becomes false control is passed to the statement following the **while** statement.

Program

```
class WhileDemo
{
      public static void main(String arg[])
      {
       int n=Integer.parseInt(arg[0]);
       int i=1,s=0;

       while(i<=n)
       {
              s=s+i;
              i++;
       }
       System.out.printlnf("Sum of Series is "+s);
 }
}
```

OUTPUT

```
C:\>java WhileDemo  10

Sum of Series is 55
```

4.6.2 do-while loop

The **do-while** loop is very similar to the while loop. The difference is that in a do-while loop, the condition is evaluated at the end of the loop. That means the loops are executed atleast once even though if the condition false.

Syntax

```
do
{
  statement(s);

}while(condition);
```

Program:

```
class DoWhileDemo
{
      public static void main(String arg[])
      {
       int n=Integer.parseInt(arg[0]);
       int i=1,f=1;
      do
      {
              f=f*i;
              i++;
      } while(i<=n);
       System.out.printlnf("Product of Series is "+f);
 }
}
```

OUTPUT

```
C:\>java DoWhileDemo  5
Product of Series is 120
```

4.6.3 For loop

For loop allows us to specify three steps about the loop in a single line Ie.,

1. Initialization of a loop
2. Condition
3. Increment / Decrement

Syntax

```
for(initialization; condition; increment/decrement)
{
 statement(s)
}
```

Program

```
class ForDemo
{
      public static void main(String arg[])
      {
       int n=Integer.parseInt(arg[0]);
       int s=0;
```

```
            if(n==0)
            System.out.println(s);
            else
            for(int i= 1;i<=n;i++)
            {
              s=s+i;
            }
      System.out.printlnf("Sum of Series is "+s);
 }
}
```

OUTPUT
```
C:\>java ForDemo  10
    Sum of Series is 55
```

4.6.4 For-each Loop

The basic *for* loop was extended in Java 5 to make iteration over arrays and other collections more convenient. This newer *for* statement is called the *enhanced for* or *for-each* (because it is called this in other programming languages). I've also heard it called the *for-in* loop.

Use it in preference to the standard for loop if applicable

Series of values. The *for-each* loop is used to access each successive value in a collection of values.

General Form

The for-each and equivalent for statements have these forms. The two basic equivalent forms are given, depending one whether it is an array or an Iterable that is being traversed. In both cases an extra variable is required, an index for the array and an iterator for the collection.

For-each loop	Equivalent for loop

`for (type var : arr) {`
`    body-of-loop`
`}`
	`for (int i = 0; i < arr.length; i++) {`
	`    type var = arr[i];`
	`    body-of-loop`
	`}`

`for (type var : coll) {`
`    body-of-loop`
`}`
	`for (Iterator<type> iter = coll.iterator(); iter.hasNext(); ) {`
	`    type var = iter.next();`
	`    body-of-loop`
	`}`

Example - Adding all elements of an array

Here is a loop written as both a *for-each* loop and a basic *for* loop.

```
double[] ar = {1.2, 3.0, 0.8};
int sum = 0;
for (double d : ar) {  // d gets successively each value in ar.
    sum += d;
}
```

And here is the same loop using the basic *for*. It requires an extra iteration variable.

```
double[] ar = {1.2, 3.0, 0.8};
int sum = 0;
for (int i = 0; i < ar.length; i++) { // i indexes each element successively.
    sum += ar[i];
}
```

Where the *for-each* is appropriate

Altho the enhanced *for* loop can make code much clearer, it can't be used in some common situations.

- Only access. Elements can not be assigned to, eg, not to increment each element in a collection.
- Only single structure. It's not possible to traverse two structures at once, eg, to compare two arrays.
- Only single element. Use only for single element access, eg, not to compare successive elements.
- Only forward. It's possible to iterate only forward by single steps.
- At least Java 5. Don't use it if you need compatibility with versions before Java 5.

4.7 Break and Continue

The break keyword halts the execution of the current loop and forces control out to the loop. The term break refers to the act of breaking out of a block of code. In order to refer to a block by name, java has a label construct that assigns a name to every block.

Continue is similar to break, except that instead of halting the execution of the loop, it starts the next iteration. The following example demonstrates the usage of continue statement.

Example for Continue Statement

Program

```
//To print odd and even numbers in a separate column

class ContinueDemo
{
	public static void main(String arg[])
	{
		for(int i=0; i<10; i++)
		{
			System.out.print(i+" ");

			if(i % 2 == 0)
			{
				continue;
			}
	System.out.println();
		}//end of for loop
	}
}
```

OUTPUT

```
0 1
2 3
4 5
6 7
8 9
```

Example for Break Statement

Program

```
//Break with label name

class BreakDemo
{
      public static void main(String arg[])
{
boolean t=true;
a:    {
b:          {
c:                {
                  System.out.println("Before the break");
                        if(t)
                        {
                              break b;

                        }
                  System.out.println("This will not execute");
                  }
                  System.out.println("This will not execute");
            }
            System.out.println("this is after b");
      }
}
}
```

OUTPUT

```
Before the break
This is after b
```

4.8 Nested loop

When one loop statement is embedded within the another loop, it is called as a Nested Loop. In java, multiple **for** loops (and **while** loops are **do-while** loops) can be nested within each other.

Program

```
      //program for Nest for loop demo
      class NestDemo
      {
            public static void main(String arg[])
            {
                  int n;
                  n=Integer.parseInt(arg[0]);
                  for(int i=1;i<=n;i++)
                  {
                        for(int j=1;j<=i;j++)
                        {
                              System.out.print(j+"  ");
                        }
                        System.out.println();
                  }

            }
      }
```

OUTPUT

```
c:\> java NestDemo 5

      1
```

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

Note Methods `println()` and `print()`

▪ **`println()`**	cursor on next line after print the message.
▪ **`print()`**	cursor on same line after print the message.