Module 2

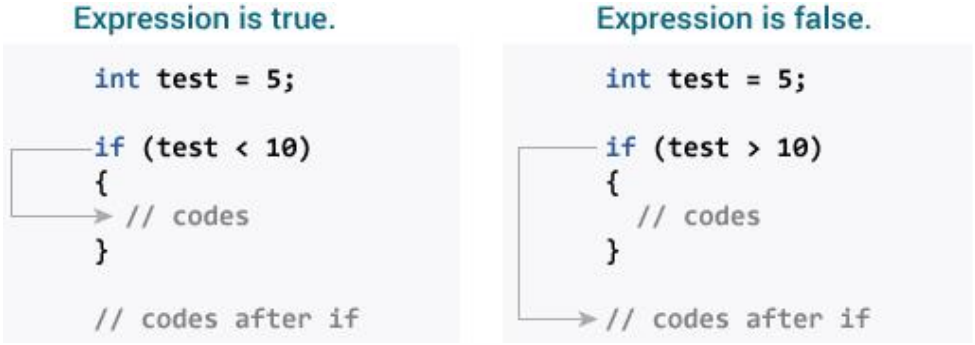
Java Basics-2

Control Statements

If statement

The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths.

How if statement works?



Here is the general form of the **if** statement:

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. The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

Nested ifs

A*nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. Consider the following example

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c;        // associated with this else
}
else a = d;          // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**).

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*. The **if** statements are executed from the top down.

It looks like this:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
..
.
else
statement;
```

```
// Demonstrate if-else-if statements.
class IfElse {
    public static void main(String args[]) {
    int month = 4; // April
    String season;
    if(month == 12 || month == 1 || month == 2)
    season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
    season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
    season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
    season = "Autumn";
    else
    season = "Bogus Month";
    System.out.println("April is in the " + season + ".");
    }
```

Here is the output produced by the program:

```
    April is in the Spring.
```

switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. It provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
switch (expression) {
case value1:
.
.
.
// statement sequence
```

```
        break;
        case value2:
        // statement sequence
        break;
        case valueN:
        // statement sequence
        break;
        default:
        // default statement sequence
        }
```

The *expression* must be of type **byte**, **short**, **int**, or **char.** The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. The **break** statement is used inside the **switch** to terminate a statement sequence.

```
// A simple example of the switch.
        class SampleSwitch {
        public static void main(String args[]) {
        for(int i=0; i<6; i++)
        switch(i) {
        case 0:
                System.out.println("i is zero.");
        break;
        case 1:
                System.out.println("i is one.");
        break;
        case 2:
                System.out.println("i is two.");
        break;
        case 3:
                System.out.println("i is three.");
        break;
        default:
                System.out.println("i is greater than 3.");
    }
}
```

Output

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Nested switch Statements

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**.

```
switch(count) {
case 1:
        switch(target) { // nested switch
        case 0:
                System.out.println("target is zero");
        break;
        case 1: // no conflicts with outer switch
                System.out.println("target is one");
        break;
        }
break;
case 2: // ...
```

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. A loop repeatedly executes the same set of instructions until a termination condition is met.

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. The general form:

```
while(condition) {

// body of loop
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with

Example 1: // Demonstrate the while loop.

```
class While {
public static void main(String args[]) {
        int n = 4;
        while(n > 0)
        {
        System.out.println("tick " + n);
        n--;
        }
    }}
```

Output

```
tick 4
tick 3
tick 2
tick 1
```

Example 2: // **Program to print Fibonacci series using while loop.**

```
public class FibonacciWhileExample {
public static void main(String[] args){
        int maxNumber = 10, previousNumber = 0, nextNumber = 1;
        System.out.print("Fibonacci Series of "+maxNumber+" numbers:");
        int i=1;
        while(i <= maxNumber)
        {
         System.out.print(previousNumber+" ");
         int sum = previousNumber + nextNumber;
         previousNumber = nextNumber;
         nextNumber = sum;
         i++;
        }
        }
    }
```

do-while

In case of while loop conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, there are times when we would like to test the termination expression at the end of the loop rather than at the beginning. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {

// body of loop

} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

Example: // Demonstrate the do-while loop.

```
class DoWhile {
        public static void main(String args[]) {
                int n = 10;
                do {
                System.out.println("tick " + n);
                n--;
            } while(n > 0);
        }
 }
```

Output

```
tick 4
```

```
tick 3
tick 2
tick 1
tick 0
```

for loop

The simplest form of the **for** loop is shown here

```
for(initialization; testexpression; update)
  {
  //  statements
  }
```

How for loop works?

1. The initialization expression is executed only once.
2. Then, the test expression is evaluated. Here, test expression is a boolean expression.
3. If the test expression is evaluated to true,
   - Codes inside the body of for loop is executed.
   - Then the update expression is executed.
   - Again, the test expression is evaluated.
   - If the test expression is true, codes inside the body of for loop is executed and update expression is executed.
   - This process goes on until the test expression is evaluated to false.
4. If the test expression is evaluated to false, for loop terminates.

Example 1:// Program to print a line 10 times

```
class Loop {
        public static void main(String[] args) {
          for (int i = 1; i <= 10; ++i) {
          System.out.println("Line " + i);
           }
        }
    }
```

Example 2: //Program to check prime number using for loop

```
import java.utin.Scanner;
public class Prime {
public static void main(String[] args) {
Scanner s=new Scanner(System.in);
int num = s.nextInt();
boolean flag = false;
for(int i = 2; i <= num/2; ++i){
        if(num % i == 0)
        { flag = true;
          break; }
    }
if(!flag)
        System.out.println(num + " is a prime number.")
```

```
        else
                System.out.println(num + " is a prime number.")
        }
 }
```

Example 3 : Program to find factorial of a number based on user input using for loop

```
Import java.util.Scanner;
public class JavaExample {
        public static void main(String[] args) {
                Scanner s=new Scanner(System.in);
                 int number = s.nextInt();
                 long fact = 1;
                 for(int i = 1; i <= number; i++)
                {
                fact = fact * i;
                }
                System.out.println("Factorial of "+number+" is: "+fact);
            }
        }
```

Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop.

```
//Using Comma in Initialization
        class Comma{
                public static void main(String args[]) {
                int a, b;
                for(a=1, b=4; a<b; a++, b--)
                {
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                }
              }
        }
```

In this example, the initialization portion sets the values of both **a** and **b**. The two comma separated statements in the iteration portion are executed each time the loop repeats.

The program generates the following output:

```
a = 1
b = 4
a = 2
b = 3
```

The For-Each version of the Loop

The for-each style of **for** is also referred to as the *enhanced* **for** loop. The general form of the for-each version of the **for** is

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

Here, data*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.**

To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace. 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. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the for loop:

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

Example 1: // **Program to find the sum of the array elements using for each 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);
    }
}
```

Example 2: // **Program to search an element of the array using for each loop**

```
import java.util.Scanner;
class Search {
    public static void main(String args[]) {
int n, Key;
Scanner s=new Scanner(System.in);
System.out.println("Enter the size of the array");
n=s.nextInt();
int a[] = new int[n];
```

```
System.out.println("Enter the elements of the array");
for(int i=0;i<n;i++)
    a[i]=s.nextInt();
System.out.println("Enter Key to be searched");
Key=s.nextInt();
int found=0;
//use for-each style for to search nums for val
for(int x : a) {
if(x == Key)
 {
found = 1;
break;
    }
}
 if(found==1)
 System.out.println("Value found");
   else
 System.out.println("Value not found!");
            }
          }
```

Jump Statements

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto. The last two uses are explained here.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++)
{
if(i == 10) break;      // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

Using break as a form of goto (Labeled break)

The general form of the labeled **break** statement is

break *label*;

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of **break** executes, control is transferred out of the named block. The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means, for example, that you can use a labeled **break** statement to exit from a set of nested blocks. But you cannot use **break** to transfer control out of a block that does not enclose the **break** statement.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement.

Example 1:// Using break as a civilized form of goto.

```
class Break {
    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.");
        }
      }
    }
```

Example 2:// Using break to exit from nested loops

```
class BreakLoop4 {
    public static void main(String args[]) {
        outer: for(int i=0; i<3; 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");
   }
class System.out.println("Loops complete.");
}
}
```

continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in

effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action.

```
// Demonstrate continue statement to print ODD numbers.
        class Continue {
                public static void main(String args[]) {
                for(int i=0; i<10; i++) {
                        if (i%2 == 0) continue;
                        System.out.print(i + " ");
                        }
                }
          }
```

Labeled continue Statement

The labeled *continue* statement is similar to the unlabelled *continue* statement in the sense that both resume the iteration. The difference with the labeled continue statement is that it resumes operation from the target label defined in the code. As soon as the labeled continue is encountered, it skips the remaining statements from the statement's body and any number of enclosing loops and jumps to the nest iteration of the enclosing labeled loop statements. Here is an example to illustrate the labeled continue statement.

```
public class LabeledContinueDemo {
  public static void main(String[] args) {
    start : for (int i = 0; i < 5; i++) {
                 System.out.println();
                  for (int j = 0; j < 10; j++) {
                    System.out.print("#");
                                if (j >= i)
                        continue start;
                  }
      System.out.println("This will never"  + " be printed");
    }
  }
}
```

return

The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

```
//Demonstrate return.
        class Return {
                public static void main(String args[]) {
                boolean t = true;
                System.out.println("Before the return.");
                if(t) return; // return to caller
                System.out.println("This won't execute.");
                  }
        }
```

Introducing Classes

Class

A mechanism that allows to combine data and operations on those data into a single unit is called a class. A class is a plan, a blueprint, a template, a logical construct. A class defines the state and behavior of its instances called objects. The data fields defined in the class are called instance variables or member variables because they are created when the objects are instantiated.

General form of a class:

```
class classname {
        type instance-variable1;
        type instance-variable2;

        // ...
        type instance-variableN;
        type methodname1(parameter-list) {
        // body of method
        }
        type methodname2(parameter-list) {
        // body of method
        }
        // ...
        type methodnameN(parameter-list) {
        // body of method
    }
}
```

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

Creating Objects

An object is essentially a block of memory that combines space to store all the instance variables. Creating an object is also referred as instantiating an object. Objects are created using the new operator. The new operator creates an object of the specified class and returns a reference of that object.

```
Rectangle rect1;   // declares an object
rect1=new Rectangle(); // instantiates the object
```

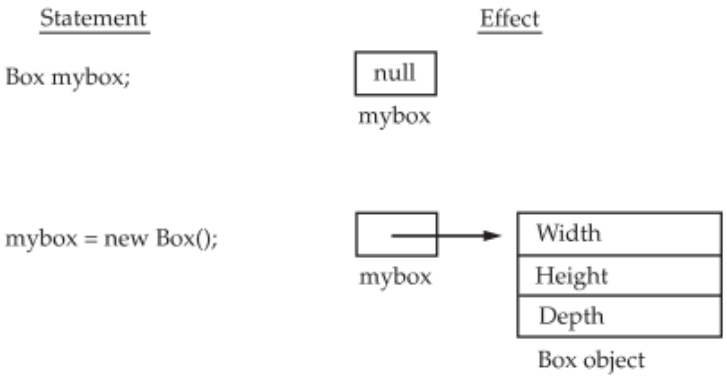
The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to it.

Consider the following example

// **This program declares two Box objects.**

```
class Box {
        double width;
        double height;
        double depth;
```

```
}
class BoxDemo2 {
public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        // assign different values to mybox2's instance variables
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);
        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
        }
}
```



The first line declares **mybox** as a reference to an object of type **Box**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**. The class name followed by parentheses(for eg., Box()) specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists.

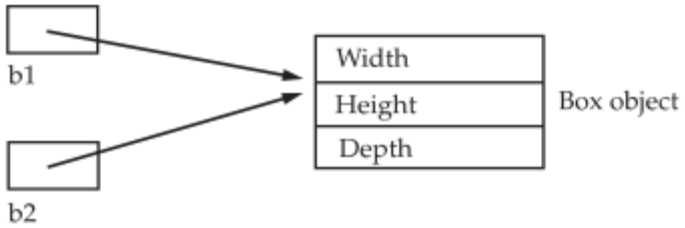
That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality.

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.



Adding a Method to the Box class

```
// This program includes a method to compute the volume  inside the box class.
        class Box {
        double width;
        double height;
        double depth;
        // display volume of a box
                void volume() {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
              }
           }
        class BoxDemo3 {
        public static void main(String args[]) {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
        // assign values to mybox1's instance variables
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
        /* assign different values to mybox2's instance variables */
                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;
        // display volume of first box
                mybox 1.volume();
```

```
        // display volume of second box
                mybox2.volume();
        }
    }
```

Returning a value

```
// Now, volume() returns the volume of a box.
        class Box {
        double width;
        double height;
        double depth;
        // compute and return volume
                double volume() {
                return width * height * depth;
               }
        }
        class BoxDemo4 {
                public static void main(String args[]) {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;
                // assign values to mybox1's instance variables
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                /* assign different values to mybox2's
                instance variables */
                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;
                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
           }
        }
```

Adding methods that takes parameters

```
// This program uses a parameterized method.
        class Box {
        double width;
        double height;
        double depth;
        // compute and return volume
                double volume() {
```

```
            return width * height * depth;
        }
        // sets dimensions of box
            void setDim(double w, double h, double d) {
            width = w;
             height = h;
            depth = d;
          }
        }

        class BoxDemo5 {
        public static void main(String args[]) {
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            double vol;
            // initialize each box
            mybox1.setDim(10, 20, 15);
            mybox2.setDim(3, 6, 9);
            // get volume of first box
            vol = mybox1.volume();
            System.out.println("Volume is " + vol);
            // get volume of second box
            vol = mybox2.volume();
            System.out.println("Volume is " + vol);
          }
        }
```

Example: Program to print the default values of instance variables.

```
        class example{
          byte b;
          short s;
          int i;
          long l;
          float f;
          double d;
          char c;
          boolean boo;
        void putData()
        {
          System.out.println(“byte default value”+b);
          System.out.println(“short default value”+s);
          System.out.println(“int default value”+i);
          System.out.println(“long default value”+l);
          System.out.println(“float default value”+f);
          System.out.println(“double default value”+d);
          System.out.println(“character default value”+c);
          System.out.println(“boolean  default value”+boo);
        }
```

```
    public static void main(Stroing args[])
    {
      example ob=new example();
      ob.putData();
    }
}
```

The dot (.) operator

- It enables you to access instance variables of any objects within a class
- It enables you to store values in instance variables of an object
- It is used to call object methods

The new operator

The **'new'** operator in java is responsible for the creation of **new object** or we can say instance of a class. The dynamically allocation is just means that the memory is allocated at the run time of the program. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.

For example :

Before creating a new object of the class we create the 'reference' variable of the class as

Demo obj; // Where Demo is the class whose object we want to create.

If we use the new operator than the actual or physical creation of the object is occurred.

The statement is :

obj = new Demo();

This statement does two things :

1. It creates the actual or physical object in the heap with the variable 'obj' pointing to the object from the stack.
2. It calls the constructor of the "Demo" class for the initialization of the object.

Constructors

- A Constructor is a special method used to initialize the object of a class.
- Constructor is automatically invoked when the object of its class is created.
- Constructor name must be same as that of a class.
- Constructor does not have a return type not even void.
- The constructor can have default arguments.
- Constructor can not be inherited.
- Constructor overloading is possible.
- Constructor overriding is not possible.

Types of Constructors

1. Default constructor
2. Parameterized constructor
3. Copy constructor

Default Constructor : A constructor which does not contain any parameter is called default constructor.

```
class test
{  int a, b;
        test()
        {  a=10;
           B=20;
        }
        void putdata()
        { System.out.println("a="+a+"b="+b);
         }
}

class consructordemo
{
        public static void main(String args[])
        {
        Test ob=new test( );
        ob.putdata( );
        }
}
```

Parameterized Constructor: The constructor which contains parameters is called parameterized constructor.

```
class test
{  int a, b;
        test(int m, int n)
        {  a=m;
           b=n;
        }
        void putdata()
        { System.out.println("a="+a+"b="+b);
         }
}

class consructordemo
{
        public static void main(String args[])
        {
        test ob=new test(10,20 );
        ob.putdata( );
        }
}
```

Copy Constructor

```
// Program to demonstrate copy constructor and constructor overloading
class copy
{  int a, b;
        copy()
```

```
        {  a=0;
           b=0;
        }
        copy(int m, int n)
        { a=m;
          b=n;
        }
        copy(copy ob)
        {
          a=ob.a;
          b=ob.b;
         }
        void putdata()
        { System.out.println("a="+a+"b="+b);
         }
}

class Demo
{
        public static void main(String args[])
        {
        copy c1=new copy( );
        copy c2 = new copy(10,20);
        copy c3=new copy(c2);
        c2.putdata();
        c3.putdata();
        }
}
```

Differences between Constructors and methods

Constructors	**Methods**
Constructor should be of the same name as that of class	Method name should not be of the same name as that of class.
Constructor is used to initialize an object	Method is used to exhibits functionality of an object.
Constructors are invoked implicitly	methods are invoked explicitly
Constructor does not return any value	method must return a value
In case constructor is not present, a default constructor is provided by java compiler	In the case of a method, no default method is provided.
Constructor overriding is not possible	Method overriding is possible

The "this" keyword

There are two uses of this keyword. They are

1. **this can be used inside any method (except static) to refer to the *current* object**. That is, this is always a reference to the object on which the method was invoked.

```
class test1
{
  void create()
  {

        System.out.println(this);// prints memory refernce of ob1 object eg., test@a26fc
   }
 public static void main(String arg[])
 {
        test1 ob1=new test1();
        ob1.create();
}
}
```

2. **Instance variable hiding**. If there is a local variable in a method with same name as instance variable, then the local variable hides the instance variable. If we want to reflect the change made over to the instance variable, this can be achieved with the help of this keyword.

```
class test1
{
 int a;
 int b;
  test1(int a,int b)
  {
        this.a=a;
        this.b=b;
    }

        void show()
        {
          System.out.println("a="+a+"b="+b);
   }
        public static void main(String arg[])
         {
        test1 ob1=new test1(10,20);
        ob1.show();
        }
}
```

Garbage Collection

C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection.* Garbage Collection is process of reclaiming the runtime unused memory automatically.It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of

your program. Garbage Collector is a Daemon thread that keeps running in the background. Basically, it frees up the heap memory by destroying the unreachable objects.

Advantages of Garbage Collection

1. The manual memory management done by the programmers is time consuming and error prone. Hence automatic memory management is done.
2. Reusability of memory is achieved.

Disadvantages of Garbage Collection

1. The execution of the program is paused or stopped during the process of garbage collection.
2. Sometimes, situations like thrashing may occur due to garbage collection.

The finalize( ) method

If an object is holding some non-Java resource such as a file handle or character font, then we might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
To add a finalizer to a class, simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, specify those actions that must be performed before an object is destroyed. It is important to understand that **finalize( )** is only called just prior to garbage collection.

The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here

}
```

A Stack class

A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop.* To put an item on top of the stack, we will use push. To take an item off the stack, we will use pop.
Here is a class called **Stack** that implements a stack for integers:

```
// This class defines an integer stack that can hold 10 values.
class Stack {
        int stck[] = new int[10];
```

```
        int tos;
// Initialize top-of-stack
        Stack() {
                tos = -1;
        }
// Push an item onto the stack
        void push(int item) {
                if(tos==9)
        System.out.println("Stack is full.");
    else
        stck[++tos] = item;
        }
// Pop an item from the stack
        int pop() {
                if(tos < 0) {
                System.out.println("Stack underflow.");
                return 0;
                }
            else
                return stck[tos--];
          }
        }
class TestStack {
        public static void main(String args[]) {
                Stack mystack1 = new Stack();
// push some numbers onto the stack
                for(int i=0; i<10; i++) mystack1.push(i);
// pop those numbers off the stack
                System.out.println("Stack in mystack1:");
                for(int i=0; i<10; i++)
                System.out.println(mystack1.pop());
          }
        }
```

Nested classes

It is possible to define a class within another class such classes are called as known as nested classes. There are two types of nested classes

1) Static class 2) non-static class

1.Static nested class

A nested class can be made as static. The nested static class can be accessed without having an object of the outer class. A static nested class can not access non-static data members and methods. It can access static data members of outer class including private. Static nested classes are rarely used.

```
class example
{
  //static class
```

```
        static class X
        {
          static String str=" Inside static class";
        }
        public static void main(String args[])
        {
         X.str="Inside class example";
        System.out.println("String stored in str"+X.str);
        }
}
```

2.Non-static nested class or Inner class

An inner class is a non static nested class. It has access to all the data members and the members of its outer class and it can refer them directly without using an object. Inner classes are a security mechanism in java**.** An inner class can be made private.

```
class outer
{
 int x=10;
        class inner
        {
          int y=20;
         void display() {
         System.out.println(x);
         System.out.println(y);
         }
        }
        void show() {
         System.out.println(x);
         // System.out.println(y); member of inner class is not accessible by outer class
         inner ob=new inner();
         ob.display();
         }
class demo{
public static void main(String args[])
        {
         outer ob1=new outer();
         ob1.show();
        }
}
```

Static keyword

1. It is used to make block of code as static
2. It is used for a static variable
3. It can be used for a method
4. A class can be made static if it is a nested class. A nested class can be accessed without having an object of outer class.

Static Block

Static block is mostly used to change the default values of static variables. This block gets executed when the class is loaded in the memory (before the execution of the main method). A class can have multiple static blocks. They will execute in the same sequence in which they have been written in the program.

Example:
```
class test1
{
        static int num;
        static{
                num=40;
                System.out.println("Static Block");
                }
        public static void main(String args[])
        {
                System.out.println(" Value of num="+num);
                }
}
```

Ouput

Static Block
Value of num=40

Using static keyword for instance variables

- If all the objects of a class want to share a common instance variable, then we have to make the instance variable as static
- Local variables can not be declared as static
- Memory allocation for static variables in done during class loading
- Static variable gets life as soon as class is loaded into JVM and it is accessible through out the class.
- A static variable belongs to the class not to the object (instance).

Example:
```
class staticdata{
        int a;
        static int b;
        void getData(int m)
        {
          a=m;
          b++;
        }
        void putData()
        {
                System.out.println( a);
                System.out.println( b);
        }
    }
class staticeg {
        public static void main(String args[])
```

```
        {
                staticdata ob1=new staticdata();
                staticdata ob2=new staticdata();
                ob1.getData(10); ob2.getData(20);
                ob1.putData(); ob2.putData();
        }
    }
```

Static method

- It is a method which belongs to the class not to the object (instance).
- A static method can access only static data.
- A static method can access only other static method and can't invoke a non static method.
- A static method can be directly invoked through the class name and dose not need any object
- A static method can't refer to "this" or "super" in anyway.
- Please note, the main method is static because it must be accessible for an application to run before any instantiation takes place.

Example:

```
class staticmethod{
        int a;    static int b;
        void getData(int m)
        {
          a=m;    b++;
        }
        Static void printData()
        {
                //System.out.println( a); can not access a since it is non static
                System.out.println( b);
        }

        void putData()
        {
                System.out.println( a);
                System.out.println( b);
        }
    }
class staticeg {
        public static void main(String args[])
        {
                staticmethod.printData();
                staticmethod ob1=new staticmethod();
                staticmethod ob2=new staticmethod();
                ob1.getData(10); ob2.getData(20);
                ob1.putData(); ob2.putData();
        }
    }
```

Static class

A nested class can be made as static. The nested static class can be accessed without having an object of the outer class.

```
class example
{
  //static class
        static class X
        {
           static String str=" Inside static class";
        }
        public static void main(String args[])
        {
          X.str="Inside class example";
        System.out.println("String stored in str"+X.str);
        }
}
```